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

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

Karen Devine\textsuperscript{*}  
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++ \textit{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 Gra\textsuperscript{ki} 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 Gra\textsuperscript{ki} library interfaces and link their codes with the MPI-based libraries. The second uses the Unix \texttt{mmap} function to allow separate Chapel and Gra\textsuperscript{ki} 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 \texttt{mmap}.

KEYWORDS

Chapel, MPI, interoperability, \texttt{mmap}, container

1 INTRODUCTION

We present two methods that allow data to be \textit{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) \cite{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 Gra\textsuperscript{ki} \cite{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.

\textsuperscript{*}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, NumberOfProcesses). 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, understanding 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
with Chapel’s
ki. The
ki (in C++) to describe the matrix that is passed to Grafi. The
RowMatrix class directly uses our shared, distributed edge lists
for answer queries about the matrix and perform matrix operations.

The most important method of RowMatrix is sparse-matrix vector
multiplication (SpMV): \( y = aX + b\mathbf{g} \) for matrix \( A \), vectors \( x \) and \( y \),
and constants \( a \) and \( b \). In parallel with distributed data, SpMV re-
quires 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 Trili-
nos’ 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

<table>
<thead>
<tr>
<th>Locale 0</th>
<th>Locale 1</th>
<th>Locale 2</th>
<th>Locale 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15</td>
<td>31</td>
<td>48</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>32</td>
<td>48</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>47</td>
<td>63</td>
</tr>
<tr>
<td>47</td>
<td>48</td>
<td>48</td>
<td>63</td>
</tr>
</tbody>
</table>

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.

3 METHODOLOGY

In the general use case that we wish to support, graph analysts load
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 reformattng 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 Grafi library. Grafi
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 Grafi. 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): \( y = aX + b\mathbf{g} \) for matrix \( A \), vectors \( x \) and \( y \),
and constants \( a \) and \( b \). In parallel with distributed data, SpMV re-
quires 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 Trili-
nos’ 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

Chapel Executable linked to glue code and MPI-based libraries

Create Chapel Distributed arrays (e.g., edge lists)

Filter data (e.g., calculated largest connected component)

Call external C function with Chapel array pointers

Call C++ functions from C function

Use pointers to build structures needed by MPI-based library (e.g., RowMatrix)

Call MPI-based library (e.g., Grafi)

Compute in MPI-based library (e.g., compute hitting times in Grafi)

Continue Chapel execution

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, multi-
tplying each edge value for edge \( (source\(_k\), target\(_k\)) \) with the ap-
propriate, 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 Grafi. 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 Grafi. 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.
Executable. Our next approach reduces this burden for Chapel users.

They also need to determine how to link the Gra
tki, Chapel users must understand how to
to directly call to Gra
tki from hanging in collective MPI communications; all locales
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 Gra
tki as in our first method. Upon completion, it resets the semaphore,
indicating to the Chapel program that the Gra
tki 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 Fig-
ure 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 exam-
ple, 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 al-
located 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 func-
tion 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
ew 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 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
Gra
tki 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
dges 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 Gra
tki, Chapel users must understand how to
call C functions from Chapel. They must ensure that their Chapel
installation, Gra
tki library, and Trilinos library were built all with
compatible compilers and MPI libraries. They also need to deter-
mine how to link the Gra
tki 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 indepen-
dent, 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 Gra
tki as in our first method. Upon completion, it resets the semaphore,
indicating to the Chapel program that the Gra
tki 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 Fig-
ure 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 exam-
ple, 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 al-
located 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 func-
tion 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
ew 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
Integrating Chapel programs and MPI-Based Libraries for High-performance Graph Analysis

CHIUW22, June 10, 2022, Virtual

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 create the source, target, and component arrays using the backing files, collectively create a distributed `RowMatrix` wrapping the data, and call Grafi ki’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 Grafi ki even easier to use by Chapel programmers, we have containerized our C++ glue program with Grafi ki, and shown that a Chapel program and the Grafi ki container can share memory through the same `mmap` array mechanism. This containerization allows Grafi ki developers to provide tools that are fully encapsulated, sparing Chapel users from having to compile the Grafi ki executable.

Our Docker container encapsulates our C++ glue program, Grafi ki and all of the libraries on which Grafi ki 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 the Grafi ki 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 Grafi 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 Grafi. 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 Grafi, which computes hitting times within the largest component. While vertices that are not in the largest component remain in the edge lists passed to Grafi, 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 Grafi directly, only one MPI rank (and, thus, one locale) can be used. Method Two, in which separate processes are used for Grafi and Chapel, allows more MPI parallelism, as the MPI parallelism is not tied to Chapel’s shared-memory implementation; we can run Grafi and Chapel with up to 28 ranks (locales).
We tested the performance of our methods with small and large graphs from the SuiteSparse matrix collection [4]. The small graph, \texttt{bsst29.mtx}, has 28 strongly connected components and a largest component with 13.8K vertices and 620K edges. The large graph, \texttt{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 Grafički 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 Grafički’s hitting time algorithm using both integration methods and the small graph \texttt{bsst29.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 Grafički, 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 \texttt{GAP-Kron.mtx} graph; execution times are in Table 2. Again, we see that Grafički’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

```plaintext
// 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.

Figure 7: Method Two: Chapel shares matrix data with the separate C++ executable that calls Grafički via a small metadata file containing per-locale backing files name and sizes.
Table 1: Runtime of Chapel connected-component label propagation and Grafiiki hitting times on Kahuna and Mutrino using Method One (Direct calls from Chapel to Grafiiki, Section 3.1) and Method Two (Separate Chapel and Grafiiki executables, Section 3.2) with small graph bcsstk29.mtx

<table>
<thead>
<tr>
<th>Platform</th>
<th>Number of Locales</th>
<th>Chapel LabelPropagation Time per iteration (seconds)</th>
<th>Grafiiki Hitting Times Time per iteration (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>One</td>
<td>Two</td>
</tr>
<tr>
<td>Kahuna</td>
<td>1</td>
<td>1.24</td>
<td>1.22</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>NA</td>
<td>1.91</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>NA</td>
<td>1.56</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>NA</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>NA</td>
<td>0.65</td>
</tr>
<tr>
<td>Mutrino</td>
<td>1</td>
<td>0.82</td>
<td>0.89</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.65</td>
<td>0.77</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.44</td>
<td>0.47</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.22</td>
<td>0.29</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.11</td>
<td>0.15</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Platform</th>
<th>Number of Locales</th>
<th>Chapel LabelPropagation Time per iteration (seconds)</th>
<th>Grafiiki Hitting Times Time per iteration (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>One</td>
<td>Two</td>
</tr>
<tr>
<td>Kahuna</td>
<td>1</td>
<td>8626</td>
<td>7476</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>NA</td>
<td>9103</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>NA</td>
<td>3749</td>
</tr>
<tr>
<td>Mutrino</td>
<td>4</td>
<td>OOM</td>
<td>10333</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1274</td>
<td>3079</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>320</td>
<td>824</td>
</tr>
</tbody>
</table>

Grafiiki linear; Grafiiki linear solve times for both approaches are nearly identical. Thus, the degradation 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 Grafiiki 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 Grafiiki. 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 Grafiiki. 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