Extending Chapel to support fabric-attached memory

Amitha C
HPC Business Group
Hewlett Packard Enterprise
Bangalore, India
amitha.c@hpe.com

Bradford L. Chamberlain
HPC Business Group
Hewlett Packard Enterprise
Seattle, USA
bradford.chamberlain@hpe.com

Clarete Riana Crasta
HPC Business Group
Hewlett Packard Enterprise
New York, USA
clarete.riana@hpe.com

Sharad Singhal
Hewlett Packard Labs
Hewlett Packard Enterprise
Milpitas, USA
sharad.singhal@hpe.com

***Abstract*—Fabric-Attached Memory (FAM) is of increasing interest in HPC clusters because it enables fast access to large datasets required in High Performance Data Analytics (HPDA) and Exploratory Data Analytics (EDA). Most approaches to handling FAM force programmers either to use low-level APIs, which are difficult to program, or to rely upon abstractions from file systems or key-value stores, which make accessing FAM less attractive than other levels in the memory model due to the overhead they bring. The Chapel language is designed to allow HPC programmers to use high-level programming constructs that are easy to use, while delegating the task of managing data and compute partitioning across the cluster to the Chapel compiler and runtime. In this paper, we describe an approach to integrate FAM access within the Chapel language, thereby simplifying the task of programming and using FAM across distributed Chapel tasks.**

***Index Terms*—fabric-attached memory, Chapel, FAM distributed arrays, high performance computing**

I. Introduction

Increasingly, HPC clusters are being used for applications in high performance data analytics where the working set is too large to fit in compute node memory. Motivated by the emergence of storage class memory (SCM), research is focused on developing programming frameworks that allow applications to address fabric-attached memory (FAM) [1]. As shown in Figure 1, FAM architectures support external fabric-attached memory accessible to all compute nodes over a high-bandwidth low-latency network. By significantly reducing the latency to persistence, these architectures allow higher performance for applications that require large working sets.

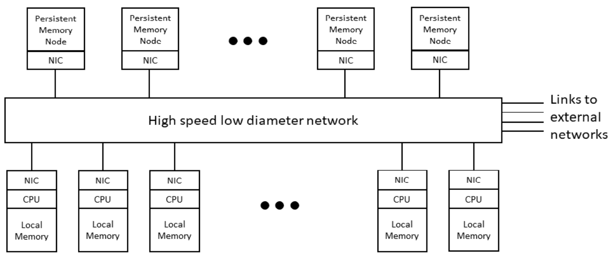

Fig. 1. FAM Architecture

Unfortunately, there are few means available to program FAM within HPC architectures. HPC programmers generally write code that is aware of low-level details of the underlying hardware and cluster topology in order to achieve the highest possible performance. However, this makes programs hard to write, understand, and maintain over time. Programming models thus frequently take a "library-based" approach, where much of the complexity is contained within libraries (e.g., MPI or SHMEM, or possibly higher-level libraries implemented on top of them) and abstracted away from the application writer.

The Chapel language [2], [3] takes a different approach—it is a language designed for programmer productivity targeted at high performance computing; the programmer simply defines large data sets and uses language constructs to indicate parallelism in the program. Chapel targets traditional clusters and supercomputers, in which data is co-located on compute nodes (or Chapel *locales*—portions of the target parallel architecture that have processing and storage capabilities). Chapel provides high-level abstractions that allow programmers to exploit locality by controlling the affinity of both data and tasks to abstract units of processing and storage capabilities. The Chapel compiler and runtime take care of distributing data and compute tasks across the cluster, communicating between them as necessary.

Like most other languages, Chapel was designed at a time when memory resources were intimately tied to the CPU. While it handles distribution of data and tasks to the compute nodes in a cluster, it does not currently support abstractions for disaggregated memory.

In this paper, we describe extensions to the Chapel language that enable applications written in Chapel to take advantage of FAM architectures. We assume that in most applications FAM-resident data is used as distributed arrays; thus, rather than providing a general file-system interface to FAM, we incorporate FAM-resident data directly as distributed arrays in Chapel.

We start with details of our design in §II and discuss the high-level architecture of our implementation, followed by the details of how FAM-resident arrays are exposed to the programmer. We describe the various operations (array allocation, lookups, iteration, etc.) available to the programmer within the language. We discuss approaches we are currently investigating for supporting multi-dimensional arrays in §III and finish the paper by briefly mentioning alternative approaches (§IV), some related work (§V), and a summary (§VI).

II. Design

Our design adds support for accessing FAM-resident data to the Chapel libraries, runtime, and compiler with minimal language changes while ensuring the same level of abstraction and parallelism that currently exists in the language for data stored in compute node memory. A guiding philosophy in our work is to minimize changes within Chapel, while ensuring existing Chapel programs are not impacted. While there are a number of ways FAM can be exposed within Chapel (see §IV), our solution currently focuses on enabling distributed arrays that are resident in FAM.

Distributed arrays form an important aspect of large-scale programming in Chapel. While the programmer treats the array as a single large sequence of elements, Chapel actually distributes the elements of the array across nodes. The programmer can specify array distributions using Chapel language statements. Array distributions provide a *global view* of the array that allows programmers to operate on the array as if it was a local array, even though its data is distributed across locales internally. Since array indices are partitioned and distributed among locales, the operations on the distributed array are broken down into multiple tasks based on array partitioning, and are assigned to the respective locales. These tasks are then executed in parallel providing implicit data parallelism. This also supports the common case of having task executions occur close to the data being referenced in the task.

Chapel array distributions can be defined by users [4], [5]. The Chapel distribution abstracts the low-level implementation of the data distribution and its access details from the language. Our design goal is to provide FAM array semantics similar to other distributions and support common operations even though the underlying FAM memory is decoupled from the compute nodes.

A. Architecture

An important aspect of Chapel distributions is that the compiler does not make any assumptions about how data is distributed across locales or about how array indices and data are represented by internal data structures within the distribution. Our solution takes advantage of this level of abstraction and flexibility within Chapel to define a new distribution policy [4] under Chapel external modules as highlighted in Figure 2 for array data resident in FAM.

Figure 3 shows a FAM-resident array with the target nodes participating in reads/writes to FAM in parallel. Unlike other distributed arrays in Chapel which reside in DRAM, FAM arrays can outlive the application. To support this, the FAM distribution module requires FAM arrays to be named during array creation. The names enable the array to be accessed by other applications, or across different run instances of the same application through an array name lookup. The FAM distribution module converts these high-level array operations into FAM-specific accesses underneath. When a FAM array is created, the complete array of the desired size is allocated on FAM by the current locale. Each locale is then assigned a partition of the array upon which to operate. Parallel array operations such as `forall`, `reduce` or `scan` are internally divided into multiple tasks based on the partitioning, and executed in parallel by the target nodes. This design ensures that the semantics of a FAM array is as close as possible to that of an existing Chapel array as possible.

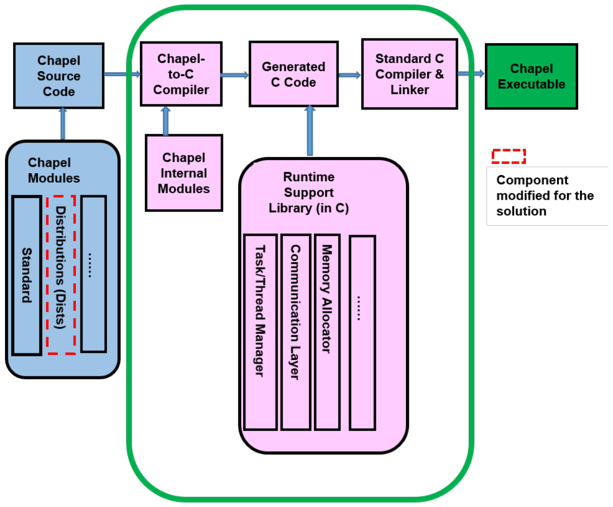

Fig. 2. Chapel Components

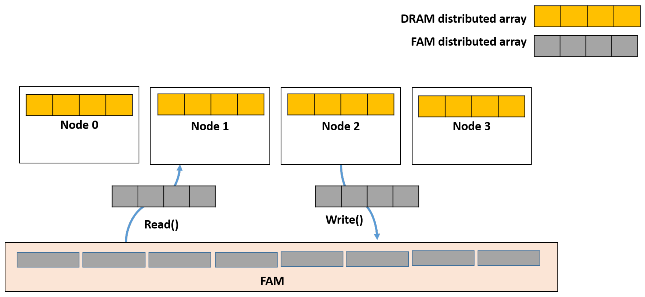

Fig. 3. DRAM and FAM Distributed Arrays

B. FAM distribution details

The FAM distribution is implemented using a separate Chapel distribution module as shown in Figure 2. The distribution module internally calls the OpenFAM library [6], [7] for access to FAM, although other libraries can also be plugged in. The module thus abstracts the details of how FAM is accessed away from the user.

Additionally, because FAM data can outlive the application, the module supports only named FAM array allocations. This enables the application to lookup and use an existing FAM array by specifying its name. Similar to other array distributions provided in Chapel, FAM arrays also support parallelism through domain partitioning, i.e. assigning the index ownership to participating nodes. Note that this "ownership" is purely for ease of partitioning the array, because the data in FAM is external to all nodes. However, the partitioning enables compute tasks to be distributed across the locales based on the ownership of the indices. Parallel array operations such as `forall`, `reduce` or `scan` are internally divided into multiple tasks based on the partitioning and executed in parallel by the target nodes.

The FAM distribution uses the same domain partitioning policy as the Chapel Block distribution [4]. When a FAM array is created, the complete array of the desired size is allocated on FAM by the current locale. Like other distributions, a FAM distribution is defined using a set of global descriptors (Chapel classes) to hold the information about the distribution instance such as index set, list of target nodes, and array names. Additionally, the FAM distribution also defines per-locale descriptors which store the locale-specific information required for FAM access.

User defined distributions support a feature called privatization that is used to avoid communication overheads when a node accesses distribution-specific information stored on a different node from where the array was declared. With privatization, every locale makes a local copy of global descriptors which hold metadata associated with the distribution instance. If any of the private copies are updated, a "re-privatization" is triggered to synchronize the local copies. The FAM distribution takes advantage of privatization to store immutable array names and locale-specific OpenFAM descriptors at all locales to reduce access overhead.

The FAM distribution in Chapel introduces a programmer productive way to store and retrieve data from FAM. The application need not understand the low-level programming API semantics required in OpenFAM. Listing 1 shows how an application can copy a large array from DRAM to FAM in parallel. The programmer only declares and allocates the array in FAM. On assigning the local array to the FAM array, the Chapel runtime creates tasks necessary to copy the contents from local array to FAM array. The number of parallel operations depend on the number of locales.

```
const Space = {1..1_000_000};
// Declare FAM distributed domain
const FamDomain = Space dmapped Fam();

var localArray: [Space] int;

// allocate new FAM distributed array
var FamArr: [FamDomain] int;
FamArr.allocate(name="records",
  auto_destroy=false);

// Copy a DRAM array to FAM.
FamArr = localArray;
```

Listing 1. Example of using FAM arrays in Chapel

As a comparison, Listing 2 shows the same copy from a local array to FAM, where the programmer has to invoke OpenFAM APIs [7]. There is no parallelism in this piece of code as these are APIs invoked directly from the application. In addition, the programmer explicitly has to handle exceptions that may occur in low-level access and deal with all of the detail required by the low-level API.

```
const Space = {1..1_000_000};
var localArray: [Space] int;

var fam_inst = new unmanaged fam();

try!{
 var fam_opts: fam_options =
 fam_inst.fam_set_options();
 fam_inst.fam_initialize
               ("CHPL", fam_opts);

 // allocate data on FAM
 const mode = 0o777;
 var regdes:fam_region_desc;
 var FamDesc: fam_desc;
 var size =
   Space.sizeAs(uint)*c_sizeof(int);

 regdes = fam_inst.fam_create_region(
  "MyRegion",size*2:size_t,mode:mode_t,1);
 FamDesc = fam_inst.fam_allocate(
  "MyArray",size:size_t,mode:mode_t,regdes);
 fam_inst.fam_put_blocking(
   localArray[Space.low],FamDesc,0,size);

} catch e: FamException {
 writeln(
 "OpenFAM error:",e.err:int,":",e.details);
 exit(1);
}
```

Listing 2. Example of accessing FAM using OpenFAM API

C. Array Operations

In this section, we describe some of the common operations that have been implemented for FAM arrays. Our goal is to provide a user experience similar to that of other existing distributions in Chapel. However, due to current limitations in Chapel's distribution interfaces and compiler to support FAM locations, there are some differences in the semantics of FAM array access, which we highlight in the discussion below.

1) Array allocation: New FAM array creation requires three Chapel statements: First, the Chapel "dmapped" keyword is used to define the distribution policy of the FAM domain and its index set. Next, the FAM array is declared with a data type specified for the array using the FAM domain. Unlike other distributed arrays, space on FAM is not allocated at the time of declaration. The array declaration only creates and sets up the necessary distribution descriptors. The corresponding space on FAM is allocated using an explicit allocate method called on the array object. The `allocate()` method requires a mandatory name and an optional flag to indicate whether FAM needs to be deallocated as part of the variable's deinitialization when the array goes out of the scope or the program exits. The default value for the `auto_destroy` flag is false.

```
const Space = {1..1_000_000};
const FamDomain = Space dmapped Fam();
// allocate new FAM distributed array
var FamArr: [FamDomain] int;
FamArr.allocate(name="records",
        auto_destroy=false);
```

2) Array lookup: Because FAM arrays can outlive the application, access to an existing array is supported using a `find()` method on the FAM array object as shown below. The `find()` method takes the array name as an argument. The internal metadata (e.g., global descriptors) is re-constituted upon a successful lookup.

```
const FamDomain = Space dmapped Fam();
var FamArr: [FamDomain] int;
FamArr.find(name="records");
```

3) Array destroy: The FAM array is either destroyed automatically as part of clean-up or by explicitly calling the `destroy()` method on the array object in the program. Automatic deallocation is controlled by the `auto_destroy` flag when the array variable goes out of scope. If `auto_destroy` is set to false, then only the internal distribution descriptors that hold FAM data references are deleted during the clean-up, and FAM memory is only released using an explicit request through `destroy()` method as shown below.

```
FamArr.allocate(name="records",
        auto_destroy=false);
...
FamArr.destroy();
```

4) Iteration: The FAM distribution supports both serial and parallel loops with zippering. Internally, the array iterator methods read the data from FAM to local DRAM and iterate over the local copy. After the iteration completes, if the array data is modified by the loop, then updated data is written back to FAM.

```
//Serial Loop on FAM Array
// assign 1 to array elements
for fa in FamArr do
        fa = 1;
// Update the array data in parallel
forall fa in FamArr do
        fa = fa *2;
// Copy elements from Block to FAM array
forall (fa,ba) in zip(FamArr,BlockArr) do
        fa = ba;
```

5) Reduce and Scan: Reduce is an operator that combines a set of values, where the operation is performed in parallel to produce a single result. The scan operation is similar to reduce, but it creates an array of results showing the partial results of the operation as applied to all elements up to that point.

```
// Calculate the average
var eltAvg =
        (+ reduce FamArr) / size**2;
// Calculate min and max valuea
var (maxVal,maxLoc) =
  maxloc reduce zip(FamArr,FamArr.domain);
var (minVal,minLoc) =
  minloc reduce zip(FamArr,FamArr.domain);
// Scan operation
writeln("Scan: Sum of all elements
        = ", (+ scan FamArr));
```

6) Array slicing and re-indexing: Chapel programmers can access just a subset of an array using a "slice" and perform operations on that sub-array. The FAM distribution supports applying common operations on FAM array slices as well as reindexing the FAM array with a new domain as shown below:

```
const Space = {1..10};
const Dfam = Space dmapped Fam();
var FamArr: [Dfam] int;
...
//Update elements with index 2 to 4
FamArr[2..4] = 500;

//Re-index the array with new domain D
const D = {6..15};
ref reA = FamArr.reindex(D);
reA[6] = 600;  // updates FamArr[1]

writeln("reA[10..15] = ",reA[10..15]);
```

7) Random indexed access: Random element access for an array is implemented using a DSI method called `dsiAccess()` in Chapel modules. When an individual indexed element is accessed in a Chapel program, the compiler in turn generates a call to the `dsiAccess()` method. The Chapel compiler expects the `dsiAccess()` interface to return a reference to the array element, which is later used for either load/store or communication get/put calls to access the array data in the runtime. Since `dsiAccess()` is designed to return the reference to DRAM-based memory, there are significant challenges in implementing the method for FAM access without making changes in the Chapel compiler.

To overcome the challenges, we have used an approach which represents the FAM access using a class wrapper object that is returned by `dsiAccess()`. In this approach, we create an instance of this class for every `dsiAccess()` request. The corresponding class object instance stores the required information for the FAM access and also overloads operators "this" and "=", which are called when an index element is accessed for read and write respectively.

Since Chapel currently supports "this" method with parentheses, indexed access to FAM-resident arrays require extra parentheses at the end as shown below. However the write semantics does not change for the FAM array element. Once Chapel adds support for user-defined implicit conversions, we expect these extra parenthesis to be unnecessary, as our class wrapper would then convert to the correct element type.

```
//Indexed element access
//write a value to 100th element
FamArr[100] = 100;
// read the value from 50th element
var x = FamArr[50]();
```

8) Bulk transfer: Bulk transfer refers to the array-to-array copy using a simple assignment operation. Typically the distributions implement bulk-transfer of the data using a parallel loop. As shown below, the FAM distribution also supports bulk-transfer to and from FAM and non-FAM arrays in parallel.

```
FamArr1.find(name="MyArray1");
FamArr2.allocate(name="MyArray2",
        auto_destroy=false);
...
// Bulk transfer
//  FAM to FAM copy
FamArr2 = FamArr1;
//  FAM <-> Block array
BlockArr = FamArr1;
...
```

```
FamArr2 = BlockArr;
```

Figure 4 shows preliminary results when a 25 GiB array is copied from FAM to a DRAM distributed array using the bulk transfer operation versus when it is directly copied using OpenFAM APIs in a manner similar to that shown in Listing 2. It can be seen that with bulk transfer, the throughput increases as the number of locales increase due to task parallelism. However, because the direct API is not parallelized, throughput drops as the number of locales is increased as a result of the communication overhead between locales.

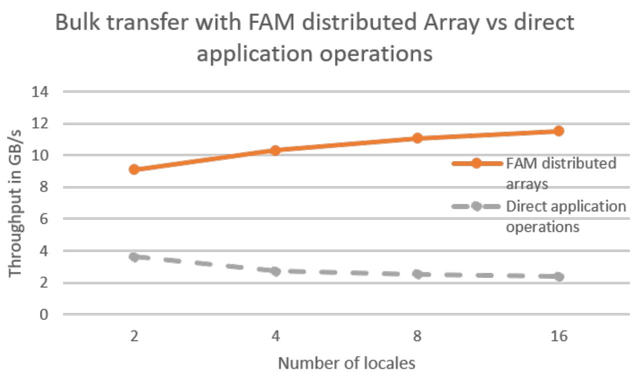

Fig. 4. Bulk transfer results

III. Multi-dimensional arrays

Similar to single dimensional arrays, multi-dimensional arrays are internally represented as a single contiguous array on FAM. But to support parallelism and task distribution, the multi-dimensional indices are partitioned and assigned to the participating compute locales. In the current design of the FAM distribution, the arrays are divided into blocks similar to Chapel's Block distribution [5]. For a multi-dimensional array, the block partition may result in non-contiguous elements per locale. This may lead to performance degradation when a locale tries to access its multi-dimensional block. To overcome this, we explored another approach for partitioning the multi-dimensional arrays, i.e., row partitioning. In the following sections we will discuss both block and row partitioning approaches, along with their advantages and disadvantages.

A. Block partitioning

Figure 5 shows an example of a 2D array, i.e. a 6 X 6 matrix which is partitioned into blocks across 4 different locales. Each block is a 2D array of 3 rows and 3 columns. When certain parallel operations like `forall`, `reduce` etc. are performed on the array, the operation is divided into multiple parallel tasks based on the ownership of the indices. Each node accesses only the indices/block that it owns and does the local computations on them. In this approach, all elements in a given partition are not contiguous for a multi-dimensional array. However, the data may be stored on FAM contiguously as shown in Figure 6. Hence to access a given block, multiple data access requests are involved. This means there are as many read/write requests as the number of rows in a partition from each locale. In this example, every locale has to perform 3 data access calls to access its elements from FAM. As the number of rows increases, the number of calls required by block partitioning also increases per locale, possibly leading to performance degradation when accessing data from FAM.

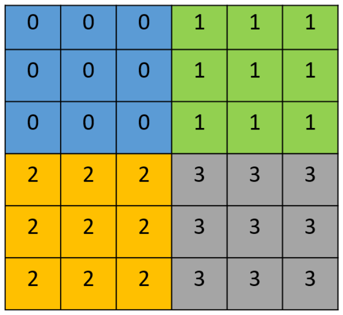

Fig. 5. 6x6 matrix block partitioned over four locales

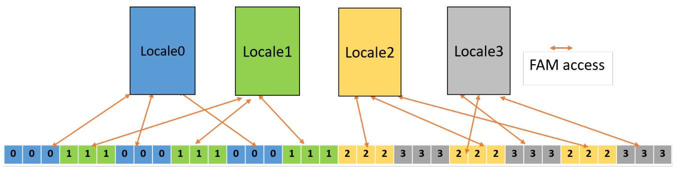

Fig. 6. Block partitioned data layout in FAM

B. Row partitioning

In this approach, a given 2D array is divided based on the number of rows and locales. For arrays greater than two dimensions, only the first dimension is partitioned based on the number of participating nodes. Figure 7 shows an example of a 2D array, i.e. a 6 X 6 matrix distributed across 4 nodes using row-partitioning. In this approach, each node gets a partition where all of its indices are contiguous. Since the data is also stored on FAM by row-major order as shown in Figure 8, a locale will need to perform only a single OpenFAM data access call to access its data elements. Hence the FAM data access in this approach has an advantage over the block access which requires multiple I/O requests. Since the partitioning is based on the number of rows and the number of locales, it can result in unused locales if the number of locales is more than the number of rows in a given array. However this scenario is highly unlikely to occur in real-world applications which mainly use FAM to store very large arrays. We are currently evaluating these options again with larger arrays and considering the most appropriate use case for the FAM distributed array in real-world chapel applications to decide on the right solution for data partitioning.

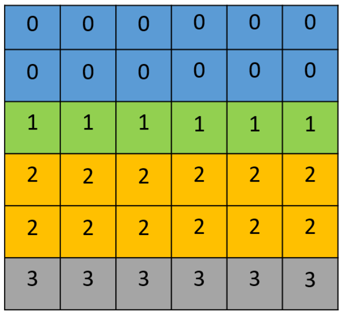

Fig. 7. 6x6 matrix row partitioned over four locales

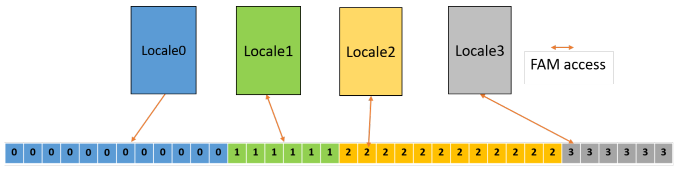

Fig. 8. 6x6 matrix row partitioned over four locales

IV. Discussion

In this paper, we described our approach to exposing FAM-resident data as distributed arrays within Chapel. However, the language provides many other ways in which FAM could be exposed to the programmer. We enumerate some of these approaches and discuss benefits (and limitations) associated with them.

A. Alternate approaches to presenting FAM in Chapel

We have also investigated a few other approaches for enabling FAM access from the Chapel runtime as discussed next.

1) FAM as a locale: This approach would add FAM as new top-level locality in the Chapel execution environment. The Chapel programming environment makes available the `Locales[]` array, which provides an abstraction of all the localities available in the current execution environment. Combined with the `on` statement, this presents a strong tool to the application to transfer control to a specific locality either to execute a task, or to have memory allocated from that locality. In all locality models currently supported by Chapel, memory is part of the top-level locales themselves. Unlike the currently supported top-level localities, FAM can be considered as a remote locality visible to the other compute localities on the network. Defining a new locality model that adds a new top-level virtual locality as the last locale in the `Locales[]` array allows the `on` statement to be re-written as shown below.

```
// last locale-id assigned to FAM
const FAM_INDEX: int = numLocales;
class C {
  var x: int;
}
var num: owned C?;
on Locales(FAM_INDEX) {
  // memory for num comes from FAM
  num = new C();
}
num!.x = 100;
```

While this model is intuitive, the Chapel locality model assumes homogeneous localities at the top-level. If FAM is conceptualized as top-level locality, then it would differ semantically from other localities, since it is a memory-only locality. This breaks fundamental assumptions in the Chapel runtime, and presents compatibility challenges and unknowns, making this approach unattractive.

2) FAM as a sub-locale: To retain top-level localities as homogeneous, it is also possible to introduce FAM as a virtual sub-locale beneath each top-level locale. The approach is inspired by the KNL [11] locality model (now obsolete) supported by Chapel that was implemented to support Intel's Xeon Phi [12] processors with high-bandwidth on-package memory (HBM). In the Chapel KNL locality model, the HBM is a sub-locale within the top-level processor locale and is made available to the Chapel programming environment as `on Locales(1).highBandwidthMemory()`. Based on this, it is possible to present FAM as a virtual sub-locale underneath each top-level locale. Like the support for HBM on KNL, new helper methods can be implemented for this FAM sub-locale, thus allowing applications to request memory from FAM via something like `on Locales(0).fabricAttachedMemory()`. Though from a programming environment, these sub-locales may appear as distinct sub-locales within the top-level locales, they would actually point to the same remote FAM. Unfortunately, although conceptually possible, implementation as a sublocale requires significant compiler changes to support external memory over the network (without load/store support) with compiler generated code.

3) FAM as an object: The idea here is to define a new class in the Chapel modules that represent FAM objects. The definition would be sufficiently generic so that it could hold whatever datatype that the user wants. The goal would be to abstract the FAM access semantics within this FAM class and allow the user to operate on FAM-resident data just like any other object. With this approach, helper methods implemented within the FAM class would handle all FAM access details through the communication layer. Allocating FAM data using this approach would appear as shown in the code below.

```
class C { } // user-defined class
var myFAMC = new FAM(C);
on Locales(FAM_INDEX) {
// memory for num comes from FAM
        num = new C();
}
num!.x = 100;
```

Here, `myFAMC` is an object that is allocated in local memory, which represents the actual data item residing on FAM. The initializer of FAM class would take care of allocating space on FAM corresponding to the user class C, and store the corresponding FAM address within `myFAMC` object. The definition of the FAM class would be generic enough to support any operation that is allowed for the user-defined class, so that any such operations can be forwarded to FAM.

We are currently exploring these approaches and working through the different cases necessary in preparation for identifying all compiler and runtime changes required in Chapel. While compiler support or changes to existing Chapel internal modules are needed in multiple places for our implementation, we have used alternate approaches to enable early adopters to experiment with FAM using Chapel. These approaches require changes in the semantics of array access when performing certain operations on FAM arrays. For example, extra parentheses are required to read an individual indexed array element from FAM, and allocation of FAM arrays is not implicit as it is for DRAM based arrays.

In addition, in existing distribution models, DRAM is allocated on the participating node. However, a FAM array resides outside all locales. Hence data on FAM arrays is always accessed over a network. Since every FAM array data access request involves a network call, the performance of FAM-based array access is not expected to be comparable to that of a DRAM-based array. However FAM arrays provide a low latency replacement for persistent secondary storage. Persistent FAM therefore offers the potential to significantly reduce the latency to persistence, thus allowing higher performance for applications that require large working sets.

Additional work involves supporting multi-dimensional arrays, enabling support for FAM from the Chapel runtime and compiler, and folding our changes back to the Chapel language. We also want to explore FAM usage in Arkouda [9], [10] and test the performance benefits to Arkouda or other Chapel applications that can take advantage of FAM.

V. Related work

Other competitive approaches to disaggregation represent data in FAM using file-system [8] or key-value store [13] abstractions. Academic research literature also has published efforts supporting transparent paging to fabric-attached memory [14]. While attractive from a programming perspective, these approaches often require kernel modules and introduce large paging overheads. FAM access to Chapel applications cluster-wide can also be enabled at the application level through external libraries such as OpenFAM, or DAOS [15], but application writers would be required to understand the APIs provided by those libraries, manage FAM and data distributions in FAM, and handle errors explicitly.

All of these approaches involve programming overhead, which contrasts with Chapel's philosophy of programmer productivity. Our solution currently leverages the OpenFAM library, but can be modified to leverage other libraries that provide access to FAM, while making FAM operations transparent to the programmer.

VI. Summary

There is increasing interest in using fabric-attached memory in HPC clusters, especially for high performance data analytics or exploratory data analytics workloads. The Chapel language provides an attractive way of writing high-performance computing applications, while hiding much of the complexity inherent in using lower-level HPC libraries. This paper describes an approach where FAM-resident data can be managed similar to the way Chapel treats DRAM-resident distributed arrays.

Our solution can be introduced into Chapel using an external module, and provides a programmer-productive way to store and retrieve data from FAM because it is directly built into the Chapel language. The application need not understand the low-level interfaces necessary to access FAM data, nor the distinct API semantics required by library-based approaches. Our design:

- Honors Chapel's programming philosophy of task parallelism.
- Abstracts FAM data allocation and accesses from the application.
- Keeps FAM array semantics close to that of existing Chapel distributions.

We have implemented our design in a proof-of-concept implementation and have identified a number of changes in the Chapel compiler and runtime that are necessary to support native access to FAM. We are discussing how to introduce our changes back to the Chapel community.

Acknowledgment

We would like to thank Sanish Suresh, Greg Titus, Elliot Ronaghan, Michael Ferguson, Shome Porno, Chinmay Ghosh and Dave Emberson for their contributions to this project.

References

[1] I. Peng, R. Pearce, and M. Gokhale, "On the Memory Underutilization: Exploring Disaggregated Memory on HPC Systems," in 2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Sep. 2020, pp. 183–190. doi: 10.1109/SBAC-PAD49847.2020.00034.

[2] "Chapel: Productive Parallel Programming." https://chapel-lang.org/ (accessed Apr. 01, 2022).

[3] Bradford L. Chamberlain. "Chapel". In: Programming Models for Parallel Computing. Ed. by Pavan Balaji. MIT Press, 2015. Chap. 6, pp. 129–159.

[4] B. L. Chamberlain, S. J. Deitz, D. Iten, and S.-E. Choi, "User-defined distributions and layouts in chapel: philosophy and framework," in Proceedings of the 2nd USENIX conference on Hot topics in parallelism, USA, Jun. 2010, p. 12.

[5] B. L. Chamberlain, S. Choi, S. J. Deitz, D. Iten, and V. Litvinov, "Authoring user-defined domain maps in chapel," 2011.

[6] K. Keeton, S. Singhal, and M. Raymond, "The OpenFAM API: A Programming Model for Disaggregated Persistent Memory," in OpenSHMEM and Related Technologies. OpenSHMEM in the Era of Extreme Heterogeneity, Cham, 2019, pp. 70–89. doi: 10.1007/978-3-030-04918-8_5.

[7] "OpenFAM: A library for programming Fabric-Attached Memory." https://openfam.github.io/index.html (accessed Aug. 29, 2021).

[8] "DAOS and Intel® OptaneTM Technology for High-Performance Storage," Intel. https://www.intel.com/content/www/us/en/high-performance-computing/daos-high-performance-storage-brief.html (accessed Apr. 01, 2022).

[9] Arkouda: NumPy-like arrays at massive scale backed by Chapel. Bears-R-Us, 2021. Accessed: Aug. 29, 2021. [Online]. Available: https://github.com/Bears-R-Us/arkouda

[10] M. Merrill, W. Reus, and T. Neumann, "Arkouda: interactive data exploration backed by Chapel," in Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop, New York, NY, USA, Jun. 2019, p. 28. doi: 10.1145/3329722.3330148.

[11] "Locale Models — Chapel Documentation 1.16." https://chapel-lang.org/docs/1.16/technotes/localeModels.html (accessed Apr. 01, 2022).

[12] A. Sodani, "Knights landing (KNL): 2nd Generation Intel® Xeon Phi processor," in 2015 IEEE Hot Chips 27 Symposium (HCS), Aug. 2015, pp. 1–24. doi: 10.1109/HOTCHIPS.2015.7477467.

[13] Y. Shan, S.-Y. Tsai, and Y. Zhang, "Distributed shared persistent memory," in Proceedings of the 2017 Symposium on Cloud Computing, New York, NY, USA, Sep. 2017, pp. 323–337. doi: 10.1145/3127479.3128610.

[14] "Rethinking software runtimes for disaggregated memory," Penn State. https://pennstate.pure.elsevier.com/en/publications/rethinking-software-runtimes-for-disaggregated-memory/fingerprints/ (accessed Apr. 01, 2022).

[15] daos-stack/daos. DAOS Storage Stack, 2020. Accessed: Aug. 27, 2020. [Online]. Available: https://github.com/daos-stack/daos