OpenFAM: Programming Disaggregated Memory

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

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

Mashood Abdulla K
HPC Business Group
Hewlett Packard Enterprise
Bangalore, India
mashood.abdulla@hpe.com

Faizan Barmawer
HPC Business Group
Hewlett Packard Enterprise
Bangalore, India
sfaizan@hpe.com

Gautham Bhat
HPC Business Group
Hewlett Packard Enterprise
Bangalore, India
gautham.bhat-k@hpe.com

Ramya Ahobala Rao
HPC Business Group
Hewlett Packard Enterprise
Bangalore, India
ramya.ahobala@hpe.com

Soumya P N
HPC Business Group
Hewlett Packard Enterprise
Bangalore, India
soumya.p.n@hpe.com

Rishi Kesh K Rajak
HPC Business Group
Hewlett Packard Enterprise
Bangalore, India
rishikesh.rajak@hpe.com

***Abstract*—HPC clusters are increasingly handling workloads where working data sets cannot be easily partitioned or are too large to fit into local node memory. In order to enable HPC workloads to access memory external to the node, HPE has defined a programming API (OpenFAM) for developing applications that use large-scale disaggregated memory. In this paper we describe an open-source reference implementation of OpenFAM that can be used on scale-up machines, traditional HPC clusters, as well as emerging disaggregated memory architectures. We demonstrate the efficiency of the implementation using micro-benchmarks.**

***Index Terms*—fabric attached memory, disaggregated memory, high performance computing, programming API, interleaved RDMA, multi-threading, contexts**

I. Introduction

High performance computing (HPC) clusters are usually optimized for workloads where the application can be partitioned and parallelized. Increasingly, these clusters are being used for applications in high performance interactive data analytics or machine learning [1] where data cannot be partitioned easily. In addition, frequently workloads require very large working sets, causing an imbalance in the compute-to-memory ratios within the clusters [2].

Emerging disaggregated memory architectures provide a new approach to handling large data sets by supporting fabric-attached memory (FAM) accessible to all compute nodes over a high-speed low-latency network. The architectures are motivated by the emergence of storage class memory (SCM) [3], which offers both persistence and higher memory density than DRAM, at latencies that are closer to DRAM speed than SSD. Currently, SCM can be provisioned directly as dense memory within compute nodes [4]. Industry efforts are underway using Compute Express Link (CXL) [5] to also enable SCM to be provisioned across nodes at a small scale. When coupled with cluster-wide interconnects such as Slingshot [6] or InfiniBand [7], FAM architectures enable data to be held in external memory accessible to all compute nodes, thus providing a new approach to handling large data sets. Because SCM can significantly reduce the latency to persistence when provisioned over a fabric, it enables higher performance for applications that require large working sets or multi-application workflows. In addition, because FAM represents a separate failure domain than compute nodes in a large cluster, FAM-based architectures can allow applications to continue running in spite of compute node failures [8], thus reducing down-time.

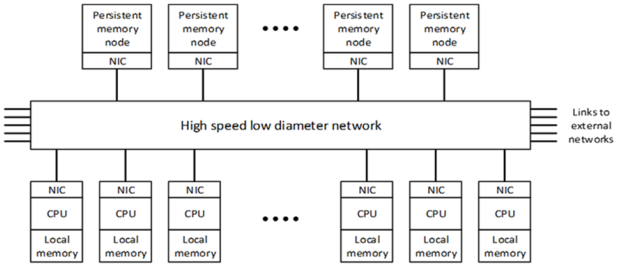

Fig. 1. Architecture of a cluster with FAM

Figure 1 shows a high level architecture for a cluster containing FAM. A high speed low diameter network connects the compute and FAM nodes. FAM nodes may be constructed out of standard servers with DRAM or SCM, may be provisioned as CXL-connected nodes, or may be light-weight nodes where the network interface cards (NICs) are linked directly to the memory using specialized memory controllers without an intervening general purpose CPU.

Note that the architecture permits current HPC applications to be run in the environment by using the compute nodes for processing, while treating the memory nodes as fast distributed storage by overlaying file-system abstractions on the memory nodes [9]. However, these abstractions reduce the potential benefits achievable by these architectures, because of the software overhead in the data paths within the file system. In contrast, the OpenFAM API [10] treats data in FAM as memory-resident, and provides memory management and lightweight data operations APIs patterned after OpenSHMEM [11]. OpenFAM provides the following benefits to the HPC

programmer over other programming models:

- The API is natural to the HPC programmers used to writing one-sided operations.
- It enables FAM allocation from distributed programs, as well as persisting allocations across programs, thus providing efficient HPC workflows to be built.
- It associates access permissions with individual allocations to restrict sharing as necessary, thus allowing user-level control of the visibility of FAM-resident data.
- The reference implementation is generic and supports both scale-up machines and scale-out clusters.

In this paper, we briefly review the OpenFAM API specification, and summarize extensions we are making to the API to add functionality and improve performance. We describe the architecture of a reference implementation [12] for OpenFAM, and provide a summary of its performance, which was described in more detail in [13]. Next, we focus on the API extensions we are making. These include support for thread safety, I/O contexts, data item interleaving, support for archival storage, and memory side operations. As part of our current work, we are porting the reference implementation to support HPE's Slingshot interconnect. We provide preliminary results from our implementation on Slingshot; we will report on a more comprehensive analysis in the future. We conclude the paper with some related work.

II. The OpenFAM API specification

We first provide a brief review of the OpenFAM API. A more detailed description of the API is present in [10], [14]. The API is targeted for use in a clustered environment (Figure 1) where each compute node runs a separate OS instance, but also has access to fabric-attached memory that is addressable using a global address space. The API assumes a two-level hierarchy for fabric-attached memory: ***Regions*** represent large data containers that have nonfunctional characteristics such as resilience or persistence associated with them. Each region is treated as a separate heap by memory managers, which can allocate ***data items*** within the region that are directly accessible by applications. Data items inherit the nonfunctional characteristics of the region within which they are allocated. Both regions and data items have access permissions associated with them to allow finer-grained access control, and can be named to enable different parts of the application (or different applications) to access a given region or data item as necessary. Rather than exposing the global address space directly to the applications, the OpenFAM API uses descriptors (opaque handles) to address FAM.

The methods in the API are grouped based on the following categories:

- *Initialization and finalization*: These operations include initialization, finalization, and aborting a running application.
- *Memory management*: These operations include region creation, destruction, and resizing, as well as data item allocation and deallocation.
- *Query and access control operations*: These operations include the ability to look up allocations by name, and change access permissions for data items or regions.
- *Data path operations*: Data path operations include blocking and non-blocking versions of get (copy data from FAM to local node memory), put (copy data from local node memory to FAM), and both strided and indexed gather and scatter operations. An additional API allows a copy to be made from one part of FAM into another part of FAM.
- *Atomics*: This group of operations include both fetching (e.g., `fetch_add()` or `compare_swap()`) and non-fetching (e.g., `set()`) operations on FAM, with memory side controllers ensuring atomicity in case the operation is performed concurrently by multiple processing elements (PEs).
- *Memory ordering and collectives*: This group includes `fence()` and `quiet()` with semantics similar to those defined in OpenSHMEM. Unlike OpenSHMEM, OpenFAM only defines a barrier operation; other collectives are not defined in the API. We are currently exploring additional collective operations (see Memory Side Operations later).
- *Memory mapping operations*: On scale-up systems or if supported by the underlying fabric [5], this set of APIs allow FAM to be mapped directly into the process address space and accessed by the CPU. Cache coherence is maintained among processors within a node (scale-up), but is not provided across nodes (scale-out) accessing FAM.

Most methods defined in the API follow a consistent pattern for providing byte-level access to FAM-resident data, where local memory is addressed using local pointers while FAM is addressed using a descriptor, a byte offset from the start of a data item, and a length field specifying the number of bytes at that offset. For example, the `get_blocking()` call is specified as

```
void fam_get_blocking(void *local,
    Fam_Descriptor *descriptor,
    uint64_t offset, uint64_t nbytes);
```

Here `local` represents the address of the destination buffer in the calling process, `descriptor` is the associated reference to the source FAM data item, and the operation is specifying that `nbytes` be copied starting at `offset` from the start of the data item in FAM to the local destination buffer. Other methods follow the same pattern. As extensions to the API, we have added the following capabilities:

- *Contexts and multi-threaded operations*: Data path operations can be partitioned into multiple contexts, and non-blocking data path and atomic operations can be independently tracked by context. Additionally, a `progress()` operation enables the application to check how many operations are pending within a context, thus enabling

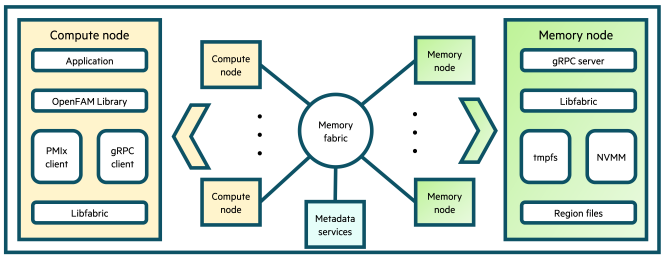

Fig. 2. Architecture of the reference implementation

more efficient use of processor threads when managing a large number of contexts.

- *Backup and restore*: These operations provide application controlled mechanisms to move data between FAM and an archival store.
- *Volatile and non-volatile regions*: Since memory nodes may contain both volatile (DRAM) and non-volatile (SCM) memory, we have extended the `create_region` API to allow the programmer to specify the type of memory hosting a given region.
- *Data interleaving*: The API has been extended to allow creation of regions where data items can be interleaved across memory servers.
- *Query region attributes*. The `fam_stat()` operation has been extended to allow the programmer to query additional attributes (e.g., number of memory servers for a region, or whether the region is interleaved or not).

III. Implementation Architecture

The overall architecture of the OpenFAM reference implementation [15] is shown in Figure 2. Examples demonstrating the use of the API as well as example applications (SpMV and PageRank) are available at [16]. The implementation assumes the architecture shown in Figure 1, where FAM is provided to compute nodes over a high speed RDMA network, and is implemented using memory servers, which serve allocations to applications running on the compute nodes. Applications are compiled with the OpenFAM library, and are deployed across the compute nodes as processing elements (PEs) using a workload manager such as SLURM [17]. The PEs treat the memory within the compute nodes as "private", while considering memory served by memory servers as "global." Once allocated by a PE, all other PEs within the application (or within other applications) can access data items from the memory servers using RDMA.

The OpenFAM implementation includes a *client library* that is linked to the PEs, a *memory management service* that runs on the memory nodes and serves memory to the PEs over RDMA, and two additional services (the *client interface service* and the *metadata management service*) that manage cluster configuration information and metadata associated with allocations respectively.

The OpenFAM client library. The client library exposes the OpenFAM API to the application, and is used by the PEs to access FAM using libfabric [18]. In addition, the client library includes a PMIx client [19] to communicate with the workload manager, and a gRPC client [20] to communicate with the OpenFAM metadata services.

The memory management service. The allocated memory is served to the PEs from memory servers. Space within a given region can span multiple memory servers and FAM can be horizontally scaled further in the cluster by incrementally adding memory nodes. Memory nodes host NVMM [21] and libfabric to support RDMA in a fabric-agnostic manner. NVMM is responsible for creation of heaps, as well as the allocation and deallocation of data items within those heaps. Each memory server uses memory-mapped files (using tmpfs or a different in-memory file system) for creating large memory regions and allows the application to allocate data items within those regions. Regions can span multiple memory nodes. The PEs interact directly with memory servers via libfabric using RDMA for data path operations such as `get`, `put`, or atomics using the topology details available from the client interface service. The implementation currently supports several fabric interconnects including Ethernet, InfiniBand, and Omnipath. For data path operations, upon validation of permissions, FAM is mapped from the memory server, registered onto libfabric and the key shared with PEs. The PEs then access FAM in the memory servers directly using libfabric. In addition to serving FAM to the clients, the memory management service also supports RDMA operations among memory servers for operations such as `fam_copy()`.

The client interface service. The client interface service (CIS) provides a layer of abstraction between PEs and metadata and memory services. All PEs interact with the CIS for region and data item allocation, lookup and other metadata and memory operations. The CIS stores cluster information such as addresses for nodes hosting other services, as well as memory node information. This service minimizes the burden on the OpenFAM client to track and maintain cluster-wide configuration information.

The metadata management service. Region, data item, and memory server metadata information is hosted in the metadata management server. Data item names and permissions are tracked using the metadata service using a key-value store (KVS). In the current implementation, the radixtree module [22] provides the KVS service. This service also serves as a resource manager. It provides a list of memory servers used for hosting regions. It also identifies memory servers where data items are allocated. The service coordinates allocations across memory servers to enable regions and data items to span memory nodes. Our initial design uses hash-based addresses for selecting memory servers when regions or data items are created. In the future, we can also enable other user-defined selection (e.g. data locality or affinity) policies. Depending on configuration parameters, the client interface service and the metadata management service can be co-located with the memory management service or run as separate executables.

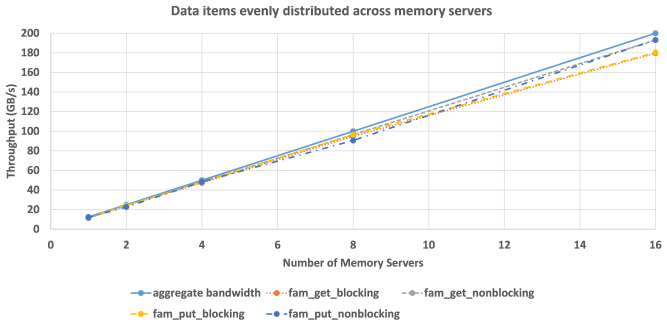

Fig. 3. Throughput achieved for data path operations with an evenly distributed workload

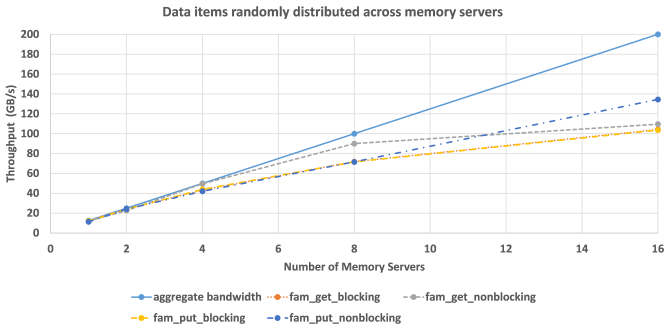

Fig. 4. Throughput achieved for data path operations with an unevenly distributed workload

IV. Implementation Performance

We have evaluated the implementation on both scale-up and scale-out architectures. In the scale-up mode, the implementation bypasses all RDMA operations, and relies on memory-to-memory copies to manage the movement of data between the memory server and the PEs. It also supports direct mapping of FAM into the process address space. In this paper, however, we concentrate on the scale-out architecture, since that is more representative of large HPC clusters.

Detailed measurements of API-level performance of OpenFAM were provided in [13]. In this paper, we briefly review those results, and present additional results obtained as a result of optimizations and additional functionality introduced into the implementation since then. The reader is referred to [13] for more a more complete evaluation of the API including scatter-gather, FAM atomics, and metadata operations. As mentioned in [13], the implementation can be compiled to enable profiling. When turned on, each API logs the time taken within itself. The logs are then used to compute averages across multiple invocations of the call, which are presented in this section.

A. Data path measurements

All data path measurements used 48 nodes from a 96-node InfiniBand cluster interconnected using 12.5 GB/s link bandwidth configured in a fat-tree. Each node had 40 Xeon Gold 6248 cores (80 hyper threaded cores) with 128 GB memory running RHEL 8.3. For data path (e.g., get, put, gather, and scatter) operations, tests were run using 34 nodes. Metadata services were hosted on two nodes, and 16 nodes hosted PEs (one per node). The remaining 16 nodes were used as memory servers, and the number of memory servers (1, 2, 4, 8, and 16) was varied within the tests. A single region was configured to span all memory servers, so data items could be distributed across the memory servers.

Figure 3 shows throughput obtained when the workload is evenly distributed across memory servers. It also shows the aggregate bandwidth available (12.5 GB/s x number of memory servers) as a reference. It is clear that the total throughput scales linearly with memory servers and is close to the aggregate bandwidth. For 16 servers, throughput ranges from 179.6 GB/s for `fam_get_blocking` to 193.1 GB/s for `fam_put_nonblocking`.

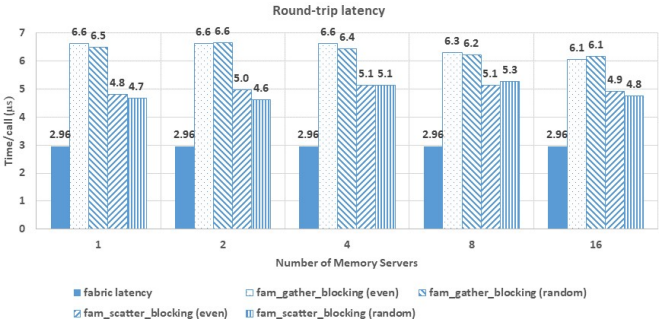

Fig. 5. Observed latency for blocking get calls using short messages

Figure 4 shows the throughput when the workload is randomly placed across memory servers. Unlike the even configuration, we see a significant drop in aggregate throughput as memory servers are added, with a maximum throughput of 134.5 GB/s for `fam_put_nonblocking` with 16 memory servers. The primary reason for the drop in throughput is the imbalance in workload, resulting in in-cast at the network interfaces on memory servers.

The round trip latency obtained with small (256 byte) data items with 16 concurrent PEs is shown in Figure 5. The Figure includes an estimate of round-trip fabric latency (obtained using the Linux `ibv_rc_pingpong` utility [23] with 256 byte messages between two servers). Comparisons show that the OpenFAM software stack incurs an additional end-to-end round trip overhead (primarily within libfabric) of slightly less than 1 µs when a single PE is accessing a single memory server.

The reader is referred to [13] for additional details on performance of other data path operations such as gather/scatter and atomics operations, as well as metadata operations such as region creation and destruction, data item allocation and deallocation, and `fam_lookup()`. In this paper, we now provide details about additional evaluations we have performed since the results in [13] were obtained, as well as new features we are implementing in OpenFAM.

B. FAM contexts and threading

The original OpenFAM API [10] assumes a basic threading and context framework for invoking the data path APIs. All data path APIs use a single communication context, thereby restricting the grouping of I/O operations to a single context group. Hence a `quiet()` operation invoked by the application enforces completion of all non-blocking operations invoked from the PE. This context model does not allow the application to optimally utilize available CPU and network resources when PEs are multi-threaded.

In many applications, fabric operations can be grouped into smaller parts, where the application can make independent progress within each subgroup as data becomes available, thus allowing finer-grained overlap of compute and communication within the application [24], [25]. The application can now create multiple contexts, and track non-blocking calls from each context separately; a `quiet()` invoked from a given context only enforces completion of operations invoked within that context. To support contexts, we have added the following APIs to OpenFAM:

```
// create a new context
fam_context* fam->fam_context_open();
// close the context
void fam->fam_context_close(fam_context*);
```

Once a context has been created, it can be used to invoke data path operations as in the original API. A `quiet()` operation invoked from the context only waits for completion of I/O operations invoked from that context. Since the application can create many more contexts than the number of cores available to the PE, the PE can check if a `quiet()` call is expected to block or not by checking the number of pending I/O calls from that context using:

```
uint64_t  fam_progress();
```

In addition, while large (e.g., 4 MiB) data transfers can achieve close to fabric bandwidth (Figure 3), for smaller sized transfers, a single thread is unable to drive enough traffic to the network, resulting in underutilized network bandwidth. Figure 6 shows the throughput achieved when a PE issues blocking calls using a 64 KiB message size using different number of threads. In this experiment, the processor contains 40 cores (80 hyperthreads). It is clear that to maximize network throughput, about 8 threads are necessary. Performance drops again once we exceed 32 threads because of context-switching between hyperthreads or because of interference from other processes (e.g., OS and daemon processes) running on the same node.

Ensuring that I/O operations invoked concurrently from multiple threads do not interfere with one another was left to the application in the earlier implementation of OpenFAM. However, this makes the application more prone to errors and data races. Therefore we have also added an option to handle concurrent accesses to OpenFAM from the application in a thread-safe manner.

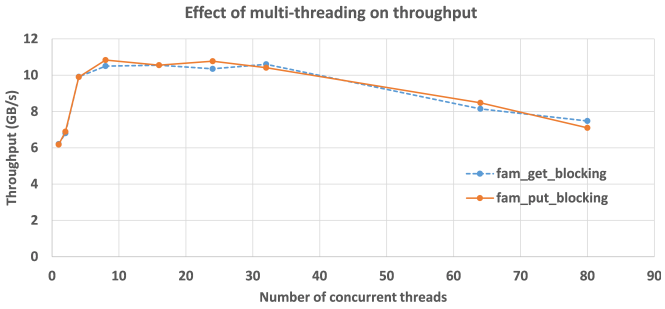

Fig. 6. Throughput achieved for data path operations with an unevenly distributed workload

C. Data item interleaving

If the data placement and distribution across memory servers is skewed due to large data sets stored on a few memory servers, it can easily create performance bottlenecks. For example, in case of a sparse matrix vector multiplication (SpMV) application using the compressed sparse row (CSR) form, the input vector is common for all PEs running on the client nodes. Since every PE needs to read the input vector, the network interface at the memory server becomes the performance bottleneck if the entire vector is located on it. Our earlier results (Figure 4) clearly showed this behavior. We hypothesized in [13] that performance in the random workload placement could be improved by enabling interleaving across memory servers at the data item level. We have since implemented data item interleaving in OpenFAM, where the application programmer can select at a region level whether the region is interleaved or not.

Most storage solutions that interleave data use a centralized controller that coordinates the read/writes across the distribution. Within OpenFAM, the application can choose if a region should be interleaved or not when it is created. For interleaved regions, the implementation sends an allocate request from a client node to each memory server hosting that region, which in turn allocates a chunk of memory (stripe) corresponding to the portion of the data item resident in that memory server. Each allocated stripe is then registered with libfabric, and the memory server returns metadata consisting of the registration key and virtual address of the stripe hosted at the server. The metadata from all memory servers is placed in the data item descriptor.

The information in the descriptor enables the OpenFAM client to directly manage RDMA operations for interleaved data items without additional calls to the metadata management service. The client can directly locate memory servers that hold the data, partition the request into multiple chunks, and dispatch them in parallel. Thus, interleaving can be handled from the client without additional intervention or coordination required from a central controller.

Figure 7 compares performance with interleaving enabled and disabled for a randomly distributed workload. In the experiment, 12 PEs access data items concurrently when the data item is interleaved across the number of memory servers

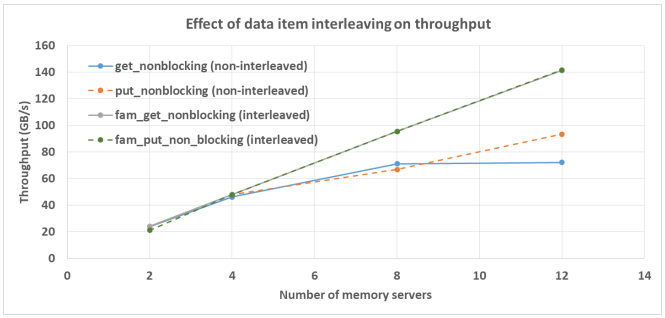

Fig. 7. Throughput for large data transfers with interleaving on or off

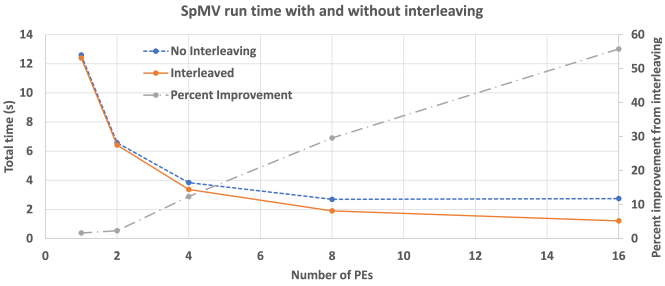

Fig. 8. Execution time for a sparse matrix vector multiplication kernel with and without interleaving

shown on the x-axis. The transfer size is 1 GiB, and interleave size is 128 KiB. It is clear that once data is interleaved, performance in the random distribution becomes much closer to the uniform workload distribution, since each PE can fetch data in parallel from multiple memory servers.

As a second test, we also tested application level performance using a sparse matrix vector multiplication kernel. In this test the data is stored in compressed sparse row (CSR) format in FAM. Each PE reads in the dense vector from FAM, then reads the matrix in groups of rows, and writes the resulting part of the result vector back to FAM. In the experiment a 2^{27} x 2^{27} sparse matrix was generated with edge factor 4, and saved in CSR format in FAM in a region spanning 8 memory servers. The interleave size is 128 KiB during the experiment. Each PE first reads the dense vector from FAM, then processes the matrix multiplication by fetching rows in groups of 2^{20} rows at a time, computing that fraction of the result vector, and writing the corresponding part of the result back to FAM. Figure 8 shows the total execution time as a comparison based on interleaving.

We observe that as the number of PEs is increased from 1 to 16, the total time taken is (approximately) inversely proportional to the number of PEs. This is expected because the dominant part of the application run-time is the computation time, which is reduced as the number of PEs is increased. However interleaving improves performance because it reduces in-cast at the memory servers by allowing data to be retrieved in parallel. Figure 9 shows time broken down separately by compute and communication time. The compute time is identical in both cases (left axis in the Figure).

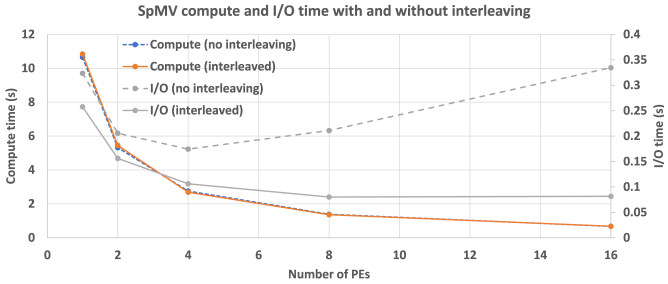

Fig. 9. Breakdown of compute and communication time for SpMV

However we observe that without interleaving, the I/O time drops until 4 PEs are used, but then starts increasing again. With interleaving, this behavior (although still present) does not appear until 16 PEs, indicating that the system is more scalable.

Additional experiments suggest a complex relationship that impacts the read/write performance of the application based on the data item size, interleave size, and data access patterns. Some regions may contain data items which are not accessed in parallel or have data items whose size is less than the interleave size. For such regions interleaving can be an overhead. For larger data items where separate parts of the data item are accessed in parallel, interleaving can provide significant benefits. To allow interleaving to be tuned, the implementation leaves parameters configurable at the region level, and provides flexibility to choose interleaving sizes for each region based on application needs.

D. Initial results on Slingshot

We are currently qualifying OpenFAM using a Slingshot network. Unlike the InfiniBand cluster, the Slingshot cluster offers a 25 GB/s link bandwidth. As a result, it is possible that some of the bottlenecks that were not visible in our code become visible with the larger link bandwidth available with Slingshot. Initial results from tests performed using a small Slingshot-based cluster are shown in Figure 10. Only `fam_get()` and `fam_put()` results are currently measured as we port our code. The experiment uses a single PE and a single memory server connected using a single Rosetta switch.

Figure 10 shows the throughput obtained as a function of transfer size ranging from 256 bytes to 64 MiB. In each case, the value shown is the average value obtained from 10,000 transfers. We observe that for short messages (less than 1 MiB), a single thread is not sufficient to drive the NIC to saturation, as also observed in Figure 6. Once the message sizes exceed 1 MiB, the PE can achieve close to full bandwidth (25 GB/s), with 64 MiB transfers reaching 23.9 GB/s for `get_blocking`, and 24.1 GB/s for `put_blocking` respectively. We need further experimentation with multi-threaded PEs (or multiple PEs per node) to validate performance improvement for smaller messages when more cores are driving the traffic.

In addition, we measured round trip latency using short messages (256 bytes), and obtained 3.2

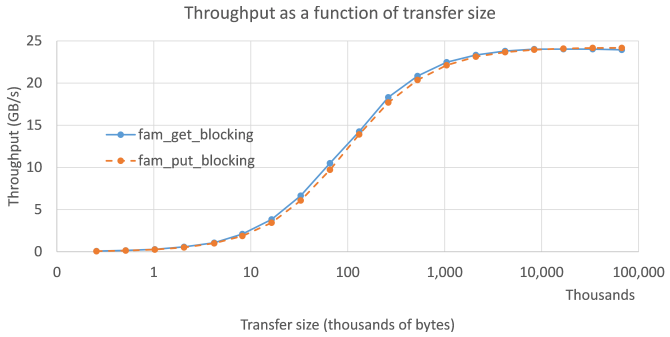

Fig. 10. Throughput obtained between a PE and a memory server over Slingshot

µs for `fam_get_blocking()` and 3.7 µs for `fam_put_blocking()`. As a comparison, the round trip latency for 256 byte messages using `fi_pingpong` Linux utility was measured at 2.6 µs. Thus the overhead in the software stack is about 1 µs, consistent with earlier results [13] obtained on the InfiniBand cluster.

V. Additional APIs

In addition to interleaving and contexts, we are introducing the following capabilities in OpenFAM.

A. Support for data retention and archival

Regions in OpenFAM are persistent, i.e., data is retained within an allocation until it is explicitly deallocated by an application. Thus, if a data item is not deallocated, it can be discovered and used by a different application either concurrently with the application that creates it, or after the original application terminates.

However, memory servers can contain both DRAM or SCM. Data in DRAM is volatile and does not survive power failure, while data in SCM is non-volatile, and may survive power failures. During region creation, the programmer can specify the region as being volatile or non-volatile. All data items within a volatile region are allocated in DRAM, while data items in non-volatile regions are allocated in SCM.

In addition, data can be copied from FAM to an archival store under application control. All movement of data between the archival store and FAM can be controlled by the application using new backup and restore APIs within the OpenFAM client.

```
void *fam_backup(Fam_Descriptor *src,
    char *backupName,
    Fam_Backup_Options backupOptions);

void *fam_restore(char *backupName,
    Fam_Descriptor *dest);

Fam_Descriptor *fam_restore(char *backupName,
    Fam_Region_Descriptor *destRegion,
    char *dataitemName, mode_t accessPermissions);
```

The application can use the `fam_backup()` API to backup a data item to a system-defined archival store (e.g., Lustre) using a `backupName`, and providing additional options (e.g., retention periods). If the application is aware of the backup size, it can use the first variant of `fam_restore()` to restore that backup to a predefined data item allocated by the application. If the application does not have pre-defined FAM space, the second variant of `fam_restore()` allows the application to restore the data item in a different region, with the OpenFAM library allocating the required data item.

B. Memory side operations

Given that in the near-term, it is anticipated that FAM will be provided using commodity servers, which have CPUs available in them, it becomes possible to use those CPUs for providing server-side operations. We are currently investigating whether applications would benefit if certain reduction operations are passed on to the memory server. For example, in many instances, communications costs in sparse matrix multiplication [26] can be reduced by doing operations in blocks, which require reduction operations on the output vector. If only one-sided operations are available, consolidation requires an element-by-element `fam_add` as a reduction. Currently, the only one-sided operation available in the API for this reduction is a `fam_add_atomic`, which incurs a significant overhead. If reduction operations were available at the memory server, substantial improvement in fabric traffic may be possible.

Note that while memory-side operations may reduce the network traffic, they increase the CPU usage in the memory servers, and may cause interference among independent applications that happen to share data on the memory servers. Additional impact arises from the use of RPC calls (as opposed to one-sided RDMA calls) between the PEs and the memory servers. Finally, with data item interleaving, the reduction operations also have to be split among memory servers. It is thus desirable to perform server-side operations only if there is a minimum number of operations to each memory server, and the performance overhead incurred in RPC can be concealed. It may thus be beneficial to aggregate multiple operations at the client push them to memory servers after a sufficient number of these operations are available [27].

To evaluate server-side operations, we are experimenting with the following APIs:

```
fam_queue_operation(FAM_QUEUE_OP op, void *local,
    Fam_Descriptor *descriptor, uint64_t  nElements,
    uint64_t  *elementIndex, uint64_t elementSize);

fam_aggregate_flush(Fam_Descriptor *desc);
```

Here, each `fam_queue_operation()` queues a set of elements to be added to the data item represented by `descriptor`. `FAM_QUEUE_OP` denotes operations such as `OP_ADD_INDEXED`, `OP_SCATTER` or `OP_GATHER`. The parameter `local` stores a pointer to an appropriately sized area of local memory, `nElements` stores number of elements to put, `elementIndex` is a local array containing element

indexes in FAM and `elementSize` is the size of each element to operate on.

These APIs combine memory server-side reduction operations with client-side message aggregation. When the application calls a `fam_queue_operation()`, the operation is queued at the PE. Once the queue is full, a sufficient number of operations are queued to PE, or an explicit `fam_aggregate_flush()` call is made, the data is pushed to memory servers for further processing. At the memory servers, the buffer is temporarily stored in FAM, and background threads at the memory server process each request. We will report on performance for this API in the future.

VI. Related Work

The OpenFAM API borrows ideas from Partitioned Global Address Space (PGAS) programming models that use one-sided operations, such as OpenSHMEM. Unlike OpenSHMEM, which assumes that the global heap is served by the compute nodes, OpenFAM relies on disaggregated memory. It differs from the Storage Networking Industry Association (SNIA) NVM Programming Model and Intel's Persistent Memory Development Kit (PMDK), which primarily address locally attached memory or replicate memory over RDMA. Unlike Intel's DAOS [4], OpenFAM exposes FAM using memory abstractions.

AsymNVM [28] provides another solution for fabric-attached persistent memory. It shares NVM devices (i.e., back-end nodes) among multiple servers (i.e., front-end nodes) and provides recoverable persistent data structures. The focus is on providing a framework where high-performance data structures can be built using FAM, and the framework focuses on data structure updates; crash consistency and replication; and data management.

AIFM [29] considers APIs that enable application developers to directly allocate fabric attached memory, and provides a runtime that handles swapping objects in and out, prefetching, and memory evacuation. Unlike OpenFAM, which exposes fabric-attached memory in scale-out environments, AIFM is targeted at providing access to fabric-attached memory using transparent caching and transfers at individual object-level instead of using virtual memory abstraction of pages.

Striping or interleaving data across servers/disks in storage context is a common technique used for achieving parallel reads/writes. In most cases, these operations are mediated by the storage controller or by a main server. When striping data across disks [30], the operations are mediated by the storage controller which controls a set of disks. In one of the implementations of multiple memory server architecture [31], the read/write request from the client to the main server goes through a tcp/udp call. The main server then does spliced RDMA to auxiliary servers and returns the data back to the client through tcp packets. Our approach uses direct RDMA from the client to the intended memory servers and has advantages over this approach as there is no mediation required and no additional tcp/udp calls necessary to achieve data interleaving. The other approach is interleaving over tcp [32], which eliminates the advantages of RDMA.

The Compute Express Link group [5] has defined the notion of memory that can be pooled using the CXL 2.0 specification. CXL 2.0 allows multiple hosts to partition a CXL-connected memory pool into logical devices, but each logical device is owned by a single host, and is accessible only by that host. CXL therefore does not support multi-host access available using OpenFAM. Although the group is working on CXL 3.0 to enable multi-host access, there are no public specifications available at this time.

Other hardware solutions include Active Memory(AM) or Processing-In-Memory(PIM) [33], which have used AM/PIM as smart memory controllers [34] or as co-processors [35]. Some of these early hardware solutions had limited adoption as integration of the logic into the hardware is costly [36]. Each is targeted to a specific use case and caters to the specific requirement like matrix multiplication [37], MapReduce [38], or graph processing [39], etc. Memory-side processing is limited by the ability to integrate multiple levels of logic into the hardware. Since we are building our solution in software for a distributed system, we have flexibility with the functionality we can provide with the memory-side operations and we also have the ability to modify and optimize the solutions in software for having compute closer to memory. Once we understand which functions provide the most benefit to use cases of interest, we can optimize the memory server by pushing functions into hardware.

VII. Conclusion and Next Steps

In this paper, we describe our progress on OpenFAM, an API for programming fabric-attached memory in HPC environments and its reference implementation. We provide early micro-benchmark results showing the scalability of the implementation. Patterned after one-sided libraries such as OpenSHMEM, OpenFAM can be used to develop directly applications that use FAM, or as a substrate for other middleware that exposes FAM to the programmer.

We are currently qualifying the reference implementation with the Slingshot provider, and are enabling access to OpenFAM APIs from Chapel and C programming languages. We are also exploring how OpenSHMEM programs can access FAM. In the next phase of the project, we plan to implement resiliency and fault tolerance in the reference implementation.

Acknowledgment

Many researchers and developers have contributed to the development of OpenFAM since its inception. The authors gratefully acknowledge contributions from Chinmay Ghosh, Cynara Justine, Kim Keeton, Vrashi Ponnappa, Sherin George, Dave Emberson, Darel Emmot, and V Bheemesh for their support and effort in making OpenFAM real.

References

[1] G. Ramirez-Gargallo, M. Garcia-Gasulla, and F. Mantovani, "TensorFlow on State-of-the-Art HPC Clusters: A Machine Learning use Case," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2019, pp. 526–533.

[2] J. D. McAlpin, "Memory Bandwidth and System Balance in HPC Systems Archives," Oct. 2016. [Online]. Available: http://sc16.supercomputing.org/tag/memory-bandwidth-and-system-balance-in-hpc-systems/

[3] M. Weiland, H. Brunst, T. Quintino, N. Johnson, O. Iffrig, S. Smart, C. Herold, A. Bonanni, A. Jackson, and M. Parsons, "An early evaluation of Intel's optane DC persistent memory module and its impact on high-performance scientific applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 1–19. [Online]. Available: https://doi.org/10.1145/3295500.3356159

[4] "DAOS and Intel® Optane™ Technology for High-Performance Storage." [Online]. Available: https://www.intel.com/content/www/us/en/high-performance-computing/daos-high-performance-storage-brief.html

[5] D. D. Sharma, "Compute Express Link 2.0 White Paper," Compute Express Link, Tech. Rep., Nov. 2020. [Online]. Available: https://www.computeexpresslink.org/_files/ugd/0c1418_14c5283e7f3e40f9b2955c7d0f60bebe.pdf

[6] D. De Sensi, S. Di Girolamo, K. H. McMahon, D. Roweth, and T. Hoefler, "An In-Depth Analysis of the Slingshot Interconnect," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2020, pp. 1–14.

[7] "InfiniBand Trade Association." [Online]. Available: https://www.infinibandta.org/

[8] K. Keeton, S. Singhal, H. Volos, Y. Zhang, R. C. Chaurasiya, C. R. Crasta, S. T. George, N. K. N, M. A. K, K. Natarajan, P. Shome, and S. Suresh, "MODC: Resilience for disaggregated memory architectures using task-based programming," Sep. 2021. [Online]. Available: https://arxiv.org/abs/2109.05329v1

[9] "daos-stack/daos," Aug. 2020, original-date: 2016-09-27T19:21:29Z. [Online]. Available: https://github.com/daos-stack/daos

[10] 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*, ser. Lecture Notes in Computer Science, S. Pophale, N. Imam, F. Aderholdt, and M. Gorentla Venkata, Eds. Cham: Springer International Publishing, 2019, pp. 70–89.

[11] "OpenSHMEM Specification | OpenSHMEM.org." [Online]. Available: http://openshmem.org/site/Specification

[12] "OpenFAM Development," Sep. 2021, original-date: 2019-10-14T18:53:54Z. [Online]. Available: https://github.com/OpenFAM/OpenFAM

[13] S. Singhal, C. R. Crasta, M. Abdulla, F. Barmawer, D. Emberson, R. Ahobala, G. Bhat, R. K. Rajak, and P. N. Soumya, "OpenFAM: A library for programming disaggregated memory," in *The OpenSHMEM and Related Technologies 2021, Workshop*, Sep. 2021, p. 18.

[14] "OpenFAM: A library for programming Fabric-Attached Memory." [Online]. Available: https://openfam.github.io/index.html

[15] "OpenFAM Development," Sep. 2021, original-date: 2019-10-14T18:53:54Z. [Online]. Available: https://github.com/OpenFAM/OpenFAM

[16] "OpenFAM Example Applications," Jun. 2021, original-date: 2019-10-14T18:53:54Z. [Online]. Available: https://github.com/OpenFAM/OpenFAM/tree/master/test/apps

[17] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple Linux Utility for Resource Management," in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Berlin, Heidelberg: Springer, 2003, pp. 44–60.

[18] "Libfabric." [Online]. Available: https://ofiwg.github.io/libfabric/

[19] "PMI v2 API - Mpich." [Online]. Available: https://wiki.mpich.org/mpich/index.php/PMI_v2_API

[20] "gRPC: A high performance, open source universal RPC framework." [Online]. Available: https://grpc.io/

[21] "HewlettPackard/gull," Jun. 2021, original-date: 2016-12-08T15:20:05Z. [Online]. Available: https://github.com/HewlettPackard/gull

[22] "HewlettPackard/meadowlark," Mar. 2021, original-date: 2016-12-08T15:21:01Z. [Online]. Available: https://github.com/HewlettPackard/meadowlark

[23] "ibv_rc_pingpong(1) - Linux manual page." [Online]. Available: https://man7.org/linux/man-pages/man1/ibv_rc_pingpong.1.html

[24] J. Dinan, D. Goodell, W. Gropp, R. Thakur, and P. Balaji, "Efficient Multithreaded Context ID Allocation in MPI," in *Recent Advances in the Message Passing Interface*, J. L. Träff, S. Benkner, and J. J. Dongarra, Eds. Berlin, Heidelberg: Springer, 2012, pp. 57–66.

[25] A. Bouteiller, S. Pophale, S. Boehm, M. B. Baker, and M. G. Venkata, "Evaluating Contexts in OpenSHMEM-X Reference Implementation," in *OpenSHMEM and Related Technologies. Big Compute and Big Data Convergence*, M. Gorentla Venkata, N. Imam, and S. Pophale, Eds. Cham: Springer International Publishing, 2018, pp. 50–62.

[26] A. Bienz, W. D. Gropp, and L. N. Olson, "Node Aware Sparse Matrix-Vector Multiplication," *arXiv:1612.08060 [cs]*, Nov. 2017, arXiv: 1612.08060. [Online]. Available: http://arxiv.org/abs/1612.08060

[27] F. M. Maley and J. G. DeVinney, "Conveyors for Streaming Many-To-Many Communication," in *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, Nov. 2019, pp. 1–8.

[28] T. Ma, M. Zhang, K. Chen, Z. Song, Y. Wu, and X. Qian, "AsymNVM: An Efficient Framework for Implementing Persistent Data Structures on Asymmetric NVM Architecture," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, Mar. 2020, pp. 757–773. [Online]. Available: https://doi.org/10.1145/3373376.3378511

[29] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay, "AIFM: High-Performance, Application-Integrated Far Memory," 2020, pp. 315–332. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/ruan

[30] S. M. Perumal and P. S. Kritzinger, "A Tutorial on RAID Storage Systems," 2004.

[31] "US Patent for Method and system for splicing remote direct memory access (RDMA) transactions in an RDMA-aware system Patent (Patent # 8,090,790 issued January 3, 2012) - Justia Patents Search." [Online]. Available: https://patents.justia.com/patent/8090790

[32] B. Tierney, E. Kissel, M. Swany, and E. Pouyoul, "Efficient data transfer protocols for big data," in *2012 IEEE 8th International Conference on E-Science*, Oct. 2012, pp. 1–9.

[33] S. Ghose, A. Boroumand, J. S. Kim, J. Gómez-Luna, and O. Mutlu, "Processing-in-memory: A workload-driven perspective," *IBM Journal of Research and Development*, vol. 63, no. 6, pp. 3:1–3:19, Nov. 2019, conference Name: IBM Journal of Research and Development.

[34] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama, "Impulse: building a smarter memory controller," in *Proceedings Fifth International Symposium on High-Performance Computer Architecture*, Jan. 1999, pp. 70–79.

[35] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca, "The architecture of the DIVA processing-in-memory chip," in *Proceedings of the 16th international conference on Supercomputing*, ser. ICS '02. New York, NY, USA: Association for Computing Machinery, Jun. 2002, pp. 14–25. [Online]. Available: https://doi.org/10.1145/514191.514197

[36] S. Ghose, K. Hsieh, A. Boroumand, R. Ausavarungnirun, and O. Mutlu, "Enabling the Adoption of Processing-in-Memory: Challenges, Mechanisms, Future Research Directions," *arXiv:1802.00320 [cs]*, Feb. 2018, arXiv: 1802.00320. [Online]. Available: http://arxiv.org/abs/1802.00320

[37] Q. Zhu, T. Graf, H. E. Sumbul, L. Pileggi, and F. Franchetti, "Accelerating Sparse Matrix-Matrix Multiplication with 3D-Stacked Logic-in-Memory Hardware," *2015 IEEE International Parallel and Distributed Processing Symposium Workshops /15 $31.00 © 2015 IEEE DOI 10.1109/IPDPSW.2015.20 1125 2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, p. 6.

[38] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "NDC: Analyzing the impact of 3D-stacked memory+logic devices on MapReduce workloads," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2014, pp. 190–200.

[39] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: Association for Computing Machinery, Jun. 2015, pp. 105–117. [Online]. Available: https://doi.org/10.1145/2749469.2750386