Disaggregated Memory in OpenSHMEM Applications – Approach and Benefits

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

Syed Ismail Faizan Barmawer
Hewlett Packard Labs
Hewlett Packard Enterprise
Bengaluru, India
sfaizan@hpe.com

Ramesh Chandra Chaurasiya
Hewlett Packard Labs
Hewlett Packard Enterprise
Bengaluru, India
rameshc@hpe.com

David Emberson
HPC Business Group
Hewlett Packard Enterprise
California, USA
emberson@hpe.com

Sajeesh K V
Hewlett Packard Labs
Hewlett Packard Enterprise
Bengaluru, India
sajeesh.kum.k-v@hpe.com

Harumi Kuno
Hewlett Packard Labs
Hewlett Packard Enterprise
Milpitas, USA
harumi.kuno@hpe.com

Ryan Menhusen
HPC Business Group
Hewlett Packard Enterprise
Colorado, USA
ryan.menhusen@hpe.com

John Byrne
Hewlett Packard Labs
Hewlett Packard Enterprise
California, USA
john.l.byrne@hpe.com

***Abstract*—HPC architectures frequently handle High Performance Data Analytics (HPDA) and Exploratory Data Analytics (EDA) workloads where the working data set cannot easily fit into node memory, causing applications to become bottlenecked by I/O performance. This poses challenges for programming models such as OpenSHMEM or MPI where all data in the working set is assumed to fit in the memory of the participating compute nodes. Often, data and results require additional I/O to be saved for analysis by other applications, or for subsequent invocations of the same application. Emerging disaggregated architectures, such as CXL GFAM, enable data to be held in external memory accessible to all compute nodes, thus providing a new approach to handling large data sets in HPC applications. In this paper, we present an approach to using disaggregated memory in OpenSHMEM applications and demonstrate the benefits of using disaggregated memory for HPC workloads.**

***Index Terms*—Disaggregated Memory, HPC programming models, OpenSHMEM, MPI**

I. Introduction

The existing memory model of OpenSHMEM [1] supports a remotely accessible memory segment called the Symmetric Heap as shown in Figure 1. Each Processing Entity (PE) contributes some memory to the symmetric heap, and all memory in the symmetric heap can be accessed remotely by all PEs using OpenSHMEM APIs. However, both OpenSHMEM and MPI [2] require the application to perform I/O to storage if the working set becomes larger than the aggregate memory capacity available in the compute nodes. Furthermore, the current memory model in OpenSHMEM is constrained to homogeneous configurations, making it hard to support emerging architectures which include GPUs or other accelerators. The OpenSHMEM community is addressing some of these limitations to provide support for distributed irregular data structures, heterogeneous configurations using CPUs, GPUs, and accelerators as well as to support dynamic memory allocations [10], [11], [14].

Accepted for publication at CUG, Perth, Western Australia, May 5-9, 2024

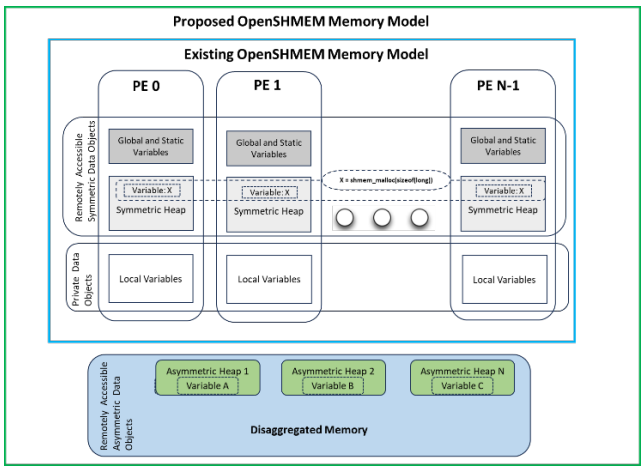

Fig. 1. Existing and Proposed OpenSHMEM Memory Model

In this paper, we describe experiments that integrate access to disaggregated memory from OpenSHMEM applications, and present evaluation results with two applications to demonstrate the benefits of using disaggregated memory.

We start in Section II with details of our proposal, which uses OpenSHMEM APIs to access the symmetric heap and OpenFAM APIs [6] to access disaggregated memory. In Section III, we start with an evaluation of the performance of OpenFAM, which we have used to access disaggregated memory. The evaluation is followed by descriptions of two applications, Sparse Matrix Vector Multiplication (SpMV) and LSD Radix Sort. In both cases, we outline a hybrid mode that uses both the symmetric heap and disaggregated memory, followed by a comparison of the hybrid mode version versus the SHMEM-only version of each application. Section III concludes with a a multi-pass FAM-based sort application to highlight how disaggregated memory can be used effectively by a sort algorithm for datasets that do not fit in compute node memory. Section IV concludes the paper.

II. Proposal

Emerging disaggregated architectures [3] provide a new approach to handling large data sets in HPC applications by supporting external memory accessible to all compute nodes over a high-speed, low-latency fabric. In the context of this paper, Disaggregated Memory (DM) [4] is a memory pool which is accessible to all compute nodes over a high-speed fabric. We also use the terms asymmetric memory or Fabric Attached Memory (FAM) interchangeably with DM in this paper. For the applications described, there is no requirement that the disaggregated memory be persistent or be connected via a dedicated memory interconnect [15], and it can be provided by any of the compute nodes or using dedicated memory servers in current HPC clusters. As shown in Figure 1, DM augments the OpenSHMEM symmetric heap by providing a pool of memory that is hosted independently of any PE, but is globally accessible from all PEs in the application. Because OpenSHMEM APIs do not provide access to disaggregated memory, we use OpenFAM [4], [5] to manage and access DM in the applications described in the paper. Given that the size of DM is independent of the number of PEs used by the application, and data within DM is outside the process address space used by the PEs, DM adds the following capabilities to OpenSHMEM applications:

- Independent scaling of memory and compute for applications, thus enabling compute resources required by the application to scale independently of the memory necessary for the application.
- Data availability across applications in workflows and across application runs (e.g., for checkpoints) without requiring data to be saved back to storage.
- The ability to allocate memory and provision data as part of application deployment, thus reducing time needed during application initialization.

In the future, disaggregated memory APIs can be wrapped under existing OpenSHMEM APIs, so SHMEM applications can continue to use a single API to access both the symmetric and asymmetric heaps.

III. Evaluation

We evaluate the benefits of disaggregated memory for SHMEM applications using two applications - SpMV [13] and LSD Radix Sort [12]. We compare SHMEM-only versions of the applications, which only use the symmetric heap, to hybrid versions that use both the symmetric heap and disaggregated memory. In each case, we describe the design for the hybrid application followed by a performance comparison of the hybrid mode version versus the SHMEM-only version. In all cases, OpenSHMEM APIs are used to access symmetric heap, while OpenFAM APIs are used to access disaggregated memory within the same application. Finally, we also present a multi-pass FAM-based sort application that uses only disaggregated memory to highlight the effective use of disaggregated memory for datasets that do not fit in the symmetric heap.

All performance results presented in this section were obtained on a cluster that consists of compute nodes and memory nodes connected over HPE SlingShot interconnect with 25 GB/s link bandwidth. Both the compute and memory nodes have 2 sockets, each with 64 AMD EPYC 7763 cores. The compute nodes have 1 TB memory and the FAM nodes have 4 TB memory each. The nodes are configured with SLES 15 SP4 and OpenFAM 3.1. Cray OpenSHMEMx 11.56 is used in the applications.

A. OpenFAM API performance numbers

As we are using OpenFAM to access and use disaggregated memory in OpenSHMEM applications, we first present the current throughput and latency numbers seen with OpenFAM for simple get and put operations. For these measurements, a single PE is configured on a compute node and a single memory server is configured on the memory node. Tests are conducted for both blocking and non-blocking get and put operations for different data sizes. The time taken per call is averaged over 10,000 iterations and used to calculate throughput. Figure 2 and Figure 3 show the throughput obtained for blocking get and put calls respectively as a function of message size for different numbers of threads.

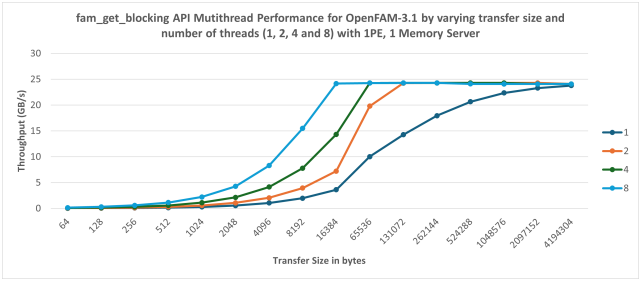

Fig. 2. fam get blocking

We observe that for blocking calls, the implementation can reach close to link bandwidth at 2 MiB message size with one thread, and at 16 KiB message size with 8 threads. Figure 4 and Figure 5 show the throughput obtained with non-blocking get and put calls for different message sizes

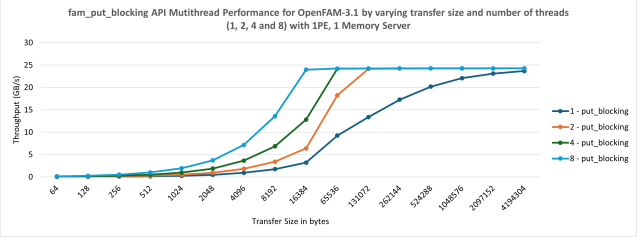

Fig. 3. fam put blocking

as the number of threads are varied between 1 to 8. As expected, non-blocking performance is better than blocking performance. The implementation can saturate the network at 16 KiB message size with a single thread, and can achieve close to link bandwidth with 4 KiB messages with 8 threads.

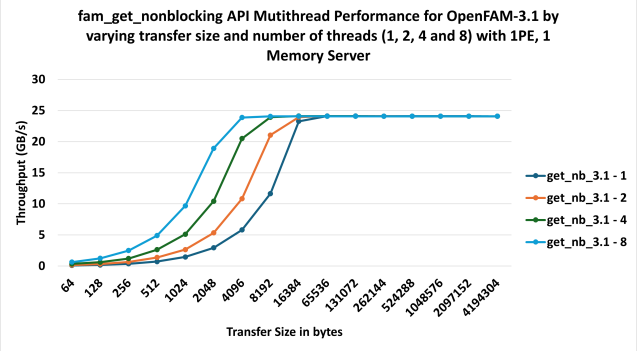

Fig. 4. fam get non blocking

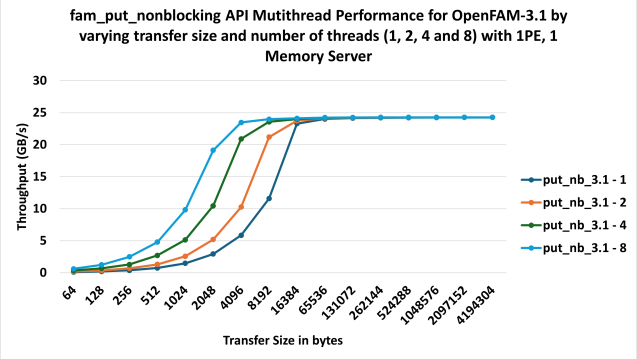

Fig. 5. fam put non blocking

Finally, Figure 6 shows the round-trip latency for OpenFAM blocking get and put calls for short messages (8 bytes - 256 bytes) as the number memory servers is varied. In all cases, the latency is less than 5 microseconds, and is constant as the number of memory servers is varied.

B. SPMV

Sparse matrix-vector multiplication (SpMV) [13] is a popular computational kernel used in a diverse set of application areas including scientific computing, economic modeling, and

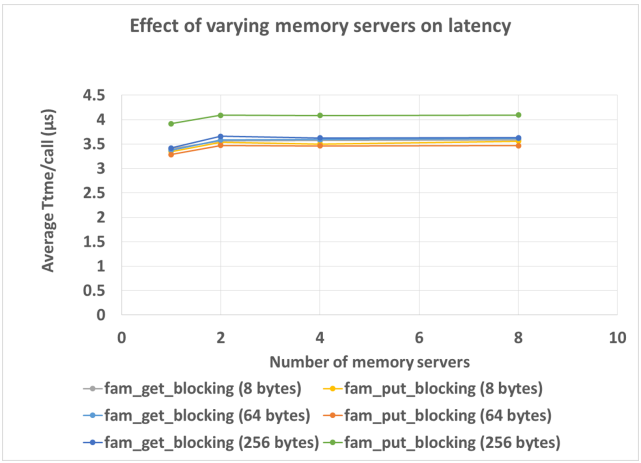

Fig. 6. fam get put latency

information retrieval. The basic problem formulation is the operation $y = Ax$, where A is a sparse matrix and x and y are dense vectors.

1) Existing design and data layout: Our reference SHMEM version of SpMV design assumes a one-dimensional row partition scheme of the input sparse matrix. Each Processing Entity(PE) owns a set of rows and the non-zeros within those rows. The implementation evenly divides the rows and the x-vector between PEs. Figure 7 provides an overview of the SpMV algorithm executed by each PE. Each PE maintains a subset of the input vector into its local memory, and reads the remaining X vector into local memory from across the PEs. It then iteratively reads a subset of A rows into local node memory, performs the multiplication computation, and writes the accumulated results back to the y-vector at its specified offset in PE 1.

2) Design approach for data layout and communication in hybrid mode: The hybrid mode also assumes a one-dimensional row partition scheme. Figure 8 provides an overview of the SpMV algorithm executed by each PE. As in the SHMEM version, each PE owns a set of rows and the non-zero values within those rows. The implementation evenly divides the rows between PEs. The distinction is that the input x-vector and the result y-vector are now placed in FAM instead of the symmetric heap. We have made the following changes to the algorithm and the data layout to the existing OpenSHMEM implementation of SpMV.

- Each PE reads the x-vector into local memory from FAM, performs the multiplication computation for its own subset of A rows, and writes back the accumulated results into FAM array. In the earlier design, each PE read the x-vector values from across the PEs; with the change in the data layout, each PE reads the x-vector from FAM.
- In the earlier design, all the PEs write the result vector to PE 1. In the new design, each PE writes the result vector in FAM, making it available to subsequent applications or workflows avoiding data movement between different memory hierarchies.

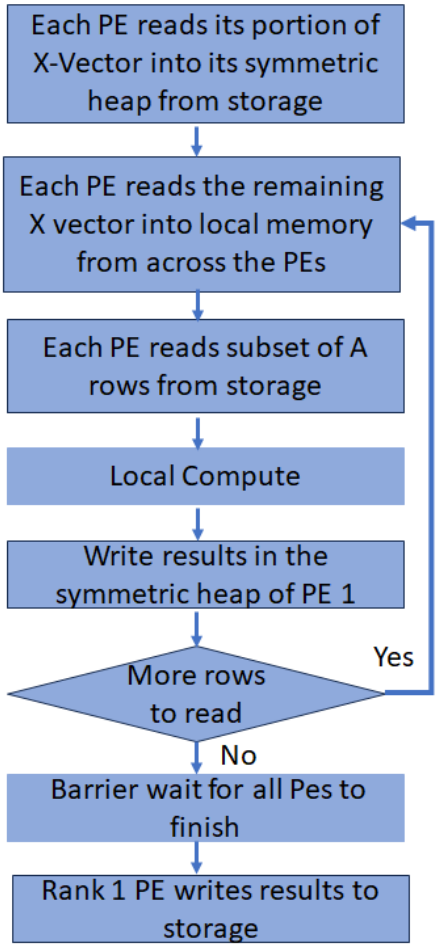

Fig. 7. SpMV SHMEM Version

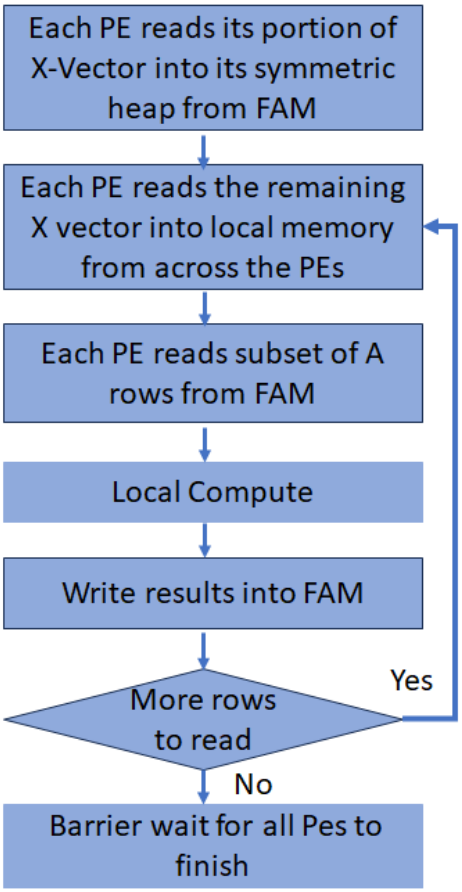

Fig. 8. SpMV Hybrid Version

3) Performance Testing: The SHMEM-only SpMV version uses Lustre as secondary storage to host input matrix and vector data and writes back the result into Lustre file system. Each PE reads only a subset of the input matrix and a subset of the input vector into its local memory. During computation, each PE fetches the rest of the input vector from other PEs in batches with a batch size of 1,024 elements. In this method, each PE's portion of the symmetric heap is used to hold a portion of input matrix and a portion of input vector. The hybrid mode SpMV version uses FAM instead of secondary storage to host input matrix and vector data and writes back the result into FAM. The rest of the logic for the hybrid mode SpMV version is similar to the SHMEM-only version described above.

The SHMEM-only SpMV and Hybrid Mode SpMV performance tests were run on eight compute nodes with 2 PEs/node; a total of 16 PEs. Each PE is bound to a different socket to maintain cache separation between the PEs. The hybrid mode used 4 OpenFAM memory servers. Figure 9 shows the comparison of the performance of SpMV hybrid mode application for matrix scale factor varying from 27 to 30. We see that as the problem sizes increase, the hybrid mode starts performing better than the SHMEM-only version, with a significant difference visible by scale factor 30. The trend suggests that the difference in performance between the two versions increases as problem size increases. Since the compute kernels are the same in both modes, the difference in performance is attributable to the difference in I/O performance.

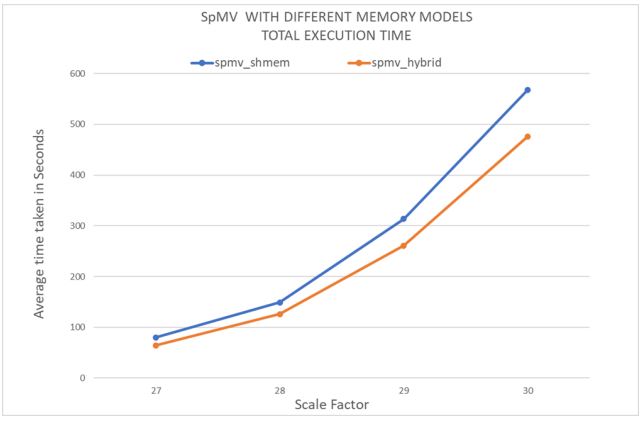

Fig. 9. Performance Comparison of SpMV Algorithms

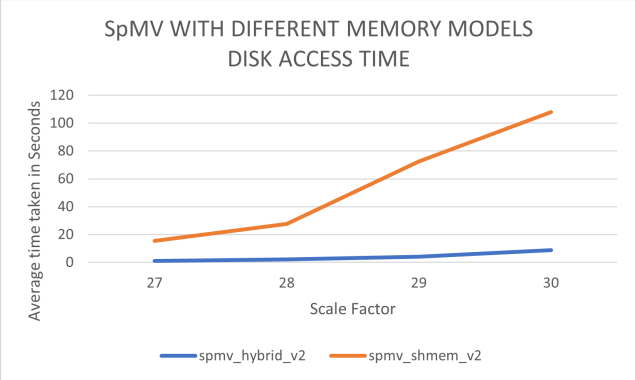

Fig. 10. Performance comparison of SpMV I/O time

Figure 10 shows the difference between the I/O times used within the applications. The figure clearly shows the benefit of using the faster FAM tier for storing both the x- and y-vectors in FAM over the case where the vectors are stored in Lustre.

C. LSD Radix Sort

LSD Radix Sort is an algorithm based on non-comparative sorting. The sort is performed in multiple iterations, with a number of digits sorted in each iteration. The number of digits selected is called the radix size, which is determined based on the processor's cache size. Every iteration of LSD Radix Sort algorithm begins with selecting the radix size number of least significant digits and then repeating the iteration for the next set of digits from right (least significant) progressing incrementally to the left (most significant). Within each iteration bins of a histogram are created for the selected radix size and are used to represent the frequency of occurrence of every digit. Once the histogram is generated, the ranks of the keys, which indicate the position of the key in the sorted order of keys, are determined. For parallel LSD Radix Sort, as shown in Figure 11, every PE works in its input data set and determines the histogram and the ranks local to the PE for its portion of input data set. Before repeating the iteration for the next set of digits towards the right, all participating PEs exchange histograms and ranks to arrive at the global histogram and global ranks for the entire input dataset handled by all the PEs. The PEs then exchange keys based on global ranks and arrive at sorted order for the selected radix size digits. The process is repeated for next set of digits in the key.

We next cover the existing design of the SHMEM version of Parallel LSD Radix Sort, the changes for hybrid mode, data-layout, and communication methods along with implementation details to understand and compare the communication patterns in the two versions. We address the various phases of Parallel LSD Radix Sort algorithm and outline the SHMEM-only and hybrid mode implementations of each phase. The SHMEM-only version of the LSD Radix Sort uses OpenSHMEM APIs for communication among PEs. All the data items used by the algorithm are allocated on the symmetric heap by all the PEs. For hybrid mode, we take the SHMEM-only

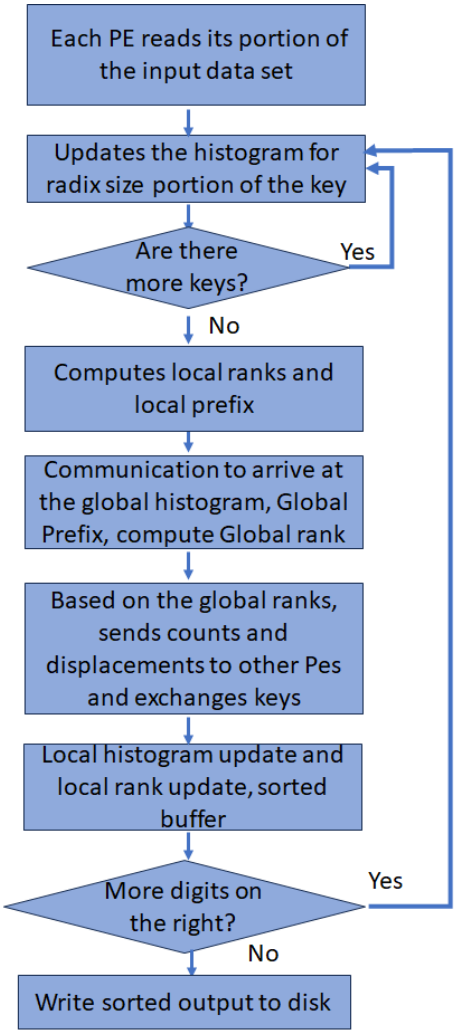

Fig. 11. Parallel LSD Radix Sort Algorithm

version of the LSD Radix Sort application as the base and modify it to use FAM wherever required across the different phases as summarized in Figure 12.

1) Phase 0: Partitioning of dataset: This is a preliminary phase before the beginning of the LSD Radix Sort algorithm. In the SHMEM-only version, each PE reads its portion of the input into its node local memory from the disk using file system I/O calls. In the hybrid mode implementation, each PE reads its portion into the input buffer from disaggregated memory using `fam_get_blocking()` routine.

2) Phase 1: Accessing keys and computing the local histogram: In this phase, each PE reads its portion of the input data set. The PEs then select the least significant radix size of every key and populate the local histogram for every digit in the portion of the key. The PEs also maintain local prefix data. The local prefix helps in maintaining the input order when there are keys of same value in the input data set. This is required for maintaining sort stability. There are no changes to the hybrid mode version as every PE works on its own input buffer and computes its own PE-specific local histogram.

Phase	Description	Data object used	Proposed change (from MPI/SHMEM)
0	Partitioning of dataset	Input buffer	Resides in Disaggregated Memory(DM)
1	Accessing keys and local histogram	Keys, Local histogram	Local computation - No change
2	Local rank computation and send buffer update	Send buffer	Local computation - No change
3	Global histogram update	Global histogram	Resides in DM
4	Global prefix	Prefix histogram	No Change
5	Global rank computation	Global rank	No Change
6	Send/Receive counts	send_counts[npes], recv_counts[npes]	PE writes and reads from DM
7	Send/Receive displacements	send_displacements[npes], recv_displacements[npes]	No Change
8	Exchange of keys	Receive buffer	No change
9	Local histogram update	Local histogram	Local computation - No change
10	Local rank update	Local rank	Local computation - No change

Fig. 12. Phases of LSD Radix Sort with proposed changes for using Disaggregated Memory

3) Phase 2: Local rank computation and Send buffer update: In this phase,

- Each PE computes the local rank of keys.
- If the same key occurs multiple times, then each occurrence has one higher rank than the previous occurrence of the same key in the order the keys are found in input data.
- Each PE arranges its input keys in a send buffer in the order of local rank of radix digits.

Like Phase 1, no change is needed for hybrid mode for this phase as every PE computes its own local rank and then works on updating a local copy of its send buffer.

4) Phase 3: Global histogram: In this phase, the algorithm computes a global histogram of all keys across all PEs using all-reduce communication. Every PE will have the same view of the global histogram. In the SHMEM-only version, every PE receives a local histogram of bucket size 2^{radix} bits from every other PE using GET calls. So, if N PEs are participating in the application run, the total number of communications calls, in worst case, is $N(N-1)$. For the hybrid mode, every PE writes its local histogram at its corresponding offset in the FAM buffer and then waits on the barrier for every other PE to complete. Once all the PEs meet at the barrier, they all read the combined local histogram from FAM with just one communication call. In this case, if N PEs are participating, the total number of communication calls, even in worst case, would be a maximum of $2N$ (with one-way big payload due to combined local histogram).

5) Phase 4: Global prefix (or prefix histogram): This phase consists of a communication call to get the prefix histogram of all keys across all PEs. There is no change from SHMEM-only version to hybrid mode for this phase.

6) Phase 5: Global rank computation: In this phase, each PE computes the global rank of its own keys using the global histogram of all keys and the prefix histogram. There is no change from SHMEM-only version to hybrid mode for this phase.

7) Phase 6: Send counts: Send counts which indicate the number of keys that need to be transferred to a destination PE are computed by dividing the global rank of keys within the PE by the number of keys per PE. Each PE sends the corresponding count value to the destination PEs. The data item used for send counts is `send_counts[nworkers]` As send counts are computed and communicated from each PE to every other PE, in the SHMEM-only version, OpenSHMEM put calls are used followed by barrier. For the hybrid mode, each PE writes send counts to FAM and waits on the barrier for this operation to complete across all the PEs. These counters are then read from FAM by PEs using single FAM get, as shown in the code snippet below.

```
for (i=0 ; i<nworkers ; i++){
    myFam->fam_put_nonblocking(
        &send_counts[i],
        rs_fdp_recv_counts,
        ((nworkers*i)+worker)*sizeof(long),
        sizeof(long));
}
myFam->fam_quiet();
// Let all the PEs update their send_counts
to other PEs recv_counts
myFam->fam_barrier_all();

// Get our recv_counts from FAM
myFam->fam_get_blocking(
    recv_counts,
    rs_fdp_recv_counts,
    nworkers*sizeof(long)*worker,
    nworkers*sizeof(long));
```

8) Phase 7: Send displacements: The displacements represent the starting index of keys in the send buffer for every PE. The send (and receive) displacements are computed by all the PEs using send (and receive) counts in this phase. The data item used for displacements is `send_displacement[nworkers]` which captures where the element from send buffer will be placed in target buffer.

9) Phase 8: Exchange of keys: In this phase, the PEs use all-to-all communication to exchange keys (elements in problem buffer). In the SHMEM-only implementation, each PE uses the 128-bit version of OpenSHMEM put to send "send count" keys to all other PEs. This phase ensures that the keys are sorted across PEs (based on the current radix digit) but does not guarantee sorted order within the PE. There were no changes made to this phase in the hybrid mode because moving elements from send to receive buffers is most likely to be local access depending on the nature of the input.

10) Phase 9: Local histogram update: This phase updates the local histograms based on the keys received in Phase 8. This phase is mostly compute-bound, so there is no change from SHMEM-only version to hybrid mode.

11) Phase 10: local rank update: In this phase, each PE computes then updates the local rank of keys. Similar to Phase

2, this ensures the sorted order of keys within the PE. This sorted receive buffer serves as input for the next radix loop. There are no changes to the hybrid mode version for this phase.

Phases 2 to 10 are repeated for the next significant radix digit through to the most significant radix digit. Eventually the sorted output is put to FAM at respective offsets as opposed to the disk storage used for the SHMEM-only version. The SHMEM-only version of LSD Radix Sort has been modified to work in hybrid mode using the approach (OpenFAM APIs and OpenSHMEM APIs working side by side) mentioned in Section II to access FAM.

D. The LSD radix sort SHMEM-only version and Hybrid mode performance comparison

In this subsection, we compare the performance of LSD Radix Sort with and without disaggregated memory. The results are obtained from running these applications with problem sizes varying from 64 million to 512 million elements per PE. Thirty-two PEs across 8 compute nodes (4 PEs/node) and 16 OMP threads per PE were used for running the tests. From initial experiments, we see that with the hybrid mode, when the application uses disaggregated memory and when the problem sizes fit in DRAM the end-to-end application time is ~45 to ~55 percent better than the SHMEM-only version as shown in Figure 13. The trend suggests that the difference in performance between the two versions increases as problem size increases. Figure 14 shows the disk access time vs FAM access time for the SHMEM and hybrid modes of the application. As with SpMV, this indicates that the faster storage offered by FAM may contribute significantly to the performance difference observed between the two versions.

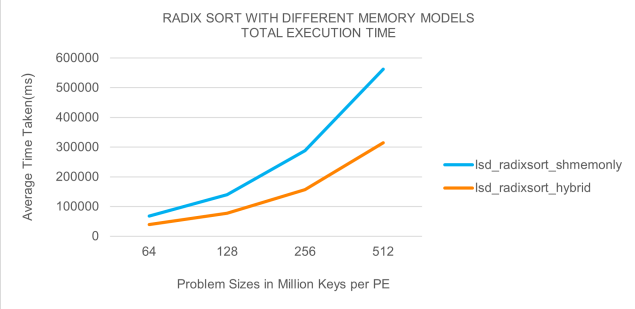

Fig. 13. Performance Comparison of Sort Algorithms

E. Multi-pass sort using FAM

Distributed sort algorithms process data in parallel using multiple nodes and require data exchange to move data between nodes at various phases. In addition, due to the limited amount of local memory that is available on individual nodes, the data on a given node might need to be sorted in multiple passes. Therefore, the performance of distributed sort algorithms depends on the network bandwidth available to load data in multiple passes and to exchange intermediate data among the nodes. The efficiency of a sort algorithm

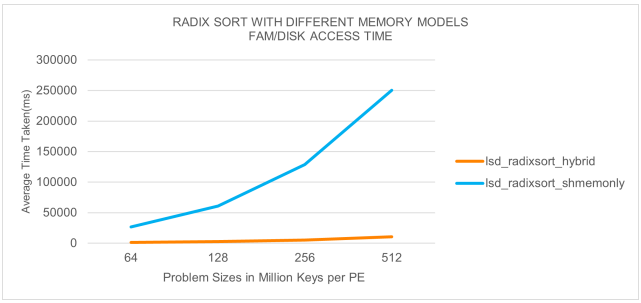

Fig. 14. Performance comparison of compute time

is measured based on the amount of data the algorithm can process per unit of time.

Our experiments show that coordination, barriers, and communication between PEs running on different nodes, as well as disk access in multiple passes when node-local memory is not sufficient to hold the entire data range, are the largest contributors to the sort time. For large (petabyte scale) problems, it is unrealistic to assume that the entire sort data would fit in the DRAM or symmetric heap of all the participating nodes. Traditional distributed sort algorithms including SHMEM-based algorithms do not work very well with multi-pass sort logic and the coordination, barriers, and communication requirements in these algorithms prove to be expensive for the sort functionality.

The graph in Figure 14 shows the time taken for disk access in the LSD radix sort single pass algorithm when data is accessed from disk in the SHMEM-only version vs FAM in the hybrid version. There is an exponential increase in disk access time with the SHMEM-only version. In the above experiments, for both the SHMEM-only version and the hybrid versions, the entire problem size was loaded into the DRAM of the nodes. For problem sizes that do not fit in the DRAM of the nodes, we need a multi-pass sorting algorithm. From our analysis of the results of LSD Radix Sort we see that the disk access for input read, and output write is the major contributor to the overall sort time and multipass sorting algorithms will have multiple iterations of disk access leading to very large sort times. The difference only widens for a multi-pass algorithm as with multi-pass algorithms there is disk access even within the algorithm logic. Thus we next describe an efficient FAM-based approach which improves sort performance by overlapping sort logic with communication and using fewer barriers, and less coordination and communication between the nodes. FAMSort is an algorithm with BurstSort logic efficiently leveraging FAM. FAMSort is designed to work with data sets larger than the available DRAM across nodes, to overlap sort logic with communication, and to have limited barriers, coordination, and communication between the PEs. The logic takes advantage of FAM by using it for incremental construction of large, shared data structures and by enabling dynamic worker coordination through work queues in FAM. We provide an overview of FAMSort in the remainder of this section. Existing BurstSort algorithms build a trie (prefix

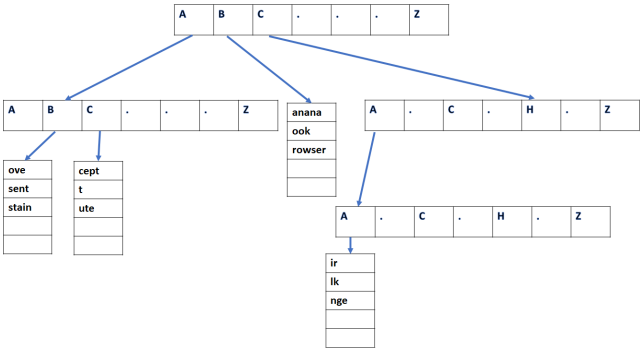

Fig. 15. BurstSort

tree), placing suffixes into limited sized buckets as shown in Figure 15. Full buckets burst into new tries. The algorithm is similar to MSD Radix Sort but faster for large data sets as it is more cache-efficient due to related radixes being placed closer together. But BurstSort loses these benefits when scaled beyond a node for large HPC data sets. Just using FAM to store the input data for the distributed BurstSort algorithm will not solve the communication and co-ordination overhead that is involved when the input data set is distributed between the nodes/PEs. We explain how we use the BurstSort logic with FAM efficiently through the enhanced FAMSort algorithm.

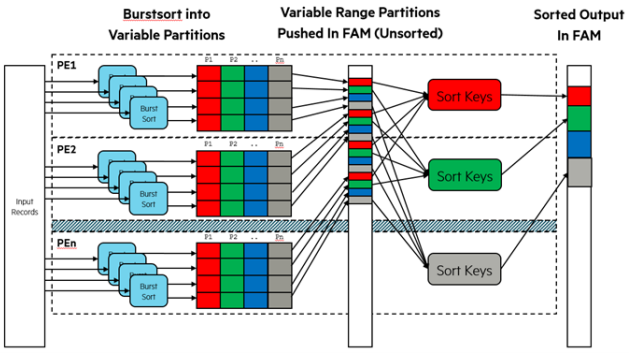

Fig. 16. FAMSort

The entire FAMSort process is divided into two phases as shown in Figure 16. The first phase assumes that the input data is available in FAM and accessible to all PEs. The input data consists of FAM data items with complete records to be sorted and data items that hold only the keys extracted from the records. We next highlight the steps in each of the phases followed by a detailed description of the two phases including the logic and data structures involved in the algorithm.

1) FAM Sort Process Phase 1 – Store keys and indexes into Burst tree:

- Gather keys from the input data items that contains the keys and an index that points to the corresponding complete record in the FAM.
- Insert the key + index into burst tree maintained in DRAM.
- Periodically traverse depth-first moving key partitions to FAM.

2) FAM Sort Process Phase 2 - Aggregate Partitions and Sort:

- Generate work queue tasks of partitions. A task is created for each of the prefixes in the range of prefixes.
- Compute nodes/PEs then acquire each of the tasks.
- If the number of keys in a task exceeds a given threshold, the bucket is pushed to the next level.
- PEs then Sort combined partitions with similar ranges.
- Sorted indices are finally written back to FAM.

The final data in FAM consists of the top-level prefix index table, sorted indices, and original data records as shown in Figure 17. The top-level prefix index table defines ordered groups of sorted indices and their location in FAM. Sorted Indices are pointers to non-contiguous locations on FAM where original input data is stored. Sorted indices help in re-structuring original data for incremental updates. The original data records remain in place saving data movement performance, and we propose that the compute nodes leverage sorted indices for `fam_gather( )` operations.

There is no range partitioning of the data in the first phase, avoiding the communication overhead involved in the range partitioning algorithms. Keys are added to a trie structure as they are processed, and once the buckets are full, buckets burst into new tries. These structures are maintained in DRAM and pushed to FAM when a certain limit is reached. The bucket contents are written depth first in the tree maintaining the sorted order. This allows for PEs to process portions of the input data depending on the amount of DRAM available. Use of FAM for creating work queues helps in optimal coordination between the PEs. It is implemented to work with limited barriers and coordination between PEs. The algorithm allows for keys to be partitioned dynamically, and partitions are based on the quantity of data and not value limits which increases PE utilization by avoiding empty buckets. The distribution of key values determines the ranges applied to each bucket. The data driven algorithm helps ensure uniform performance. Buckets with deterministic sizes help load balance.

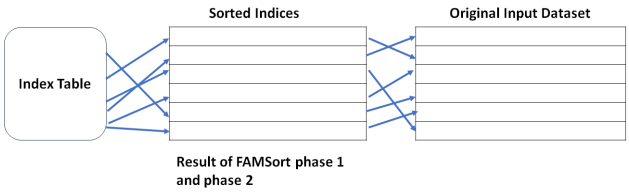

Fig. 17. FAMSort

3) FAMSort Performance Evaluation: We currently have a single-threaded implementation of FamSort. Thirty-two PEs were distributed across 8 compute nodes (4 PEs/node) for this evaluation. The graph in Figure 18 shows the comparison of the performance of FAMSort, LSD RadixSort hybrid mode, and SHMEM-only version. We leverage the results for LSD RadixSort from the experiments described in the previous

section. FAMSort was run with problem sizes varying from 128 million to 1024 million keys. At smaller problem sizes, FAMSort takes more time to complete when compared to the two versions of LSD RadixSort, but as the trend suggests, the FAMSort performs better as the problem size increases while the time taken for LSD RadixSort hybrid and SHMEM-only versions increases almost exponentially. The results show that though there is FAM access within the algorithm for input read and intermediate data structure writes, FAM Sort performs well. This provides an effective multi-pass sort algorithm for problem sizes that do not fit in DRAM.

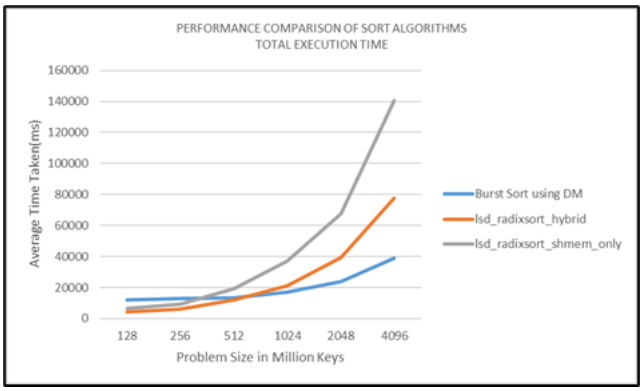

Fig. 18. FAMSort Results

IV. Summary

Competing approaches to enhance OpenSHMEM to enable large working sets use file-system and/or object-store abstractions [14] which do not include generic disaggregated memory in the base memory model, and also require IO extensions to support external memory/storage. We have not seen other benchmarks and results which quantify the performance impact to OpenSHMEM applications by effective utilization of both symmetric heap and disaggregated memory within the same OpenSHMEM application. Our experiments use multiple memory models in the same application. The approach described in the paper can be extended to other programming models such as MPI that allow co-existence of multiple frameworks. As future work, we can further enable the use of disaggregated memory by integrating the capability to address disaggregated memory directly into OpenSHMEM for effective resource utilization and ease of programming. We believe the approach described in this paper can be used to support CXL GFAM [18] access in OpenSHMEM applications, where CXL GFAM is configured as DM. We plan to validate this when we have access to CXL GFAM.

Acknowledgment

We thank Mark Pagel, Naveen Ravi, Danielle Sikich of the Cray OpenSHMEMx Team for their valuable support. We thank HPE's OpenFAM team, Application and Benchmarking Team for their contribution to OpenFAM and Radix Sort experiments. We thank the OpenSHMEM community for their valuable support. We thank Ryan Menhusen and Darel Emmot for their contributions to the FAMSort algorithm.

References

[1] "OpenSHMEM Specification 1.5." http://openshmem.org/site/Specification (accessed Sep. 05, 2020).
[2] "Message Passing Interface." https://www.mcs.anl.gov/research/projects/mpi/ (accessed Oct. 07, 2022).
[3] 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.
[4] 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.
[5] "OpenFAM: A library for programming Fabric-Attached Memory." https://openfam.github.io/index.html (accessed Aug. 29, 2021).
[6] "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).
[7] 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.
[8] "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).
[9] daos-stack/daos. DAOS Storage Stack, 2020. Accessed: Aug. 27, 2020. [Online]. Available: https://github.com/daos-stack/daos
[10] Extending the OpenSHMEM Memory Model to Support User-Defined Spaces : Aaron Welch, Swaroop Pophale et al.
[11] OpenSHMEM Memory Spaces. GitHub: OpenSHMEM, 2021. Accessed: Jun. 21, 2021. [Online]. Available: https://github.com/openshmem-org/specification/wiki/Memory-Spaces
[12] https://www.growingwiththeweb.com/sorting/radix-sort-lsd/
[13] http://www.cslab.ece.ntua.gr/cgi-bin/twiki/view/CSLab/SPMV
[14] https://github.com/Sandia-OpenSHMEM/SOS [2]M. Grodowitz, P. Shamis, and S. Poole, "OpenSHMEM I/O Extensions for Fine-Grained Access to Persistent Memory Storage," in Driving Scientific and Engineering Discoveries Through the Convergence of HPC, Big Data and AI, vol. 1315, J. Nichols, B. Verastegui, A. 'Barney' Maccabe, O. Hernandez, S. Parete-Koon, and T. Ahearn, Eds. Cham: Springer International Publishing, 2020, pp. 318–333. doi: 10.1007/978-3-030-63393-6_21.
[15] Y. Sun et al., "Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices." arXiv, Mar. 27, 2023. doi: 10.48550/arXiv.2303.15375.
[16] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, Nov. 2007, pp. 1–12. doi: 10.1145/1362622.1362674.
[17] Extreme Sort, https://www.eolymp.com/en/problems/7712
[18] D. D. Sharma, R. Blankenship, and D. S. Berger, "An Introduction to the Compute Express Link (CXL) Interconnect", arXiv, Mar. 22, 2024. https://arxiv.org/abs/2306.11227v2.