On the Memory Underutilization: Exploring Disaggregated Memory on HPC Systems

Ivy Peng*, Roger Pearce*, Maya Gokhale*
* *Lawrence Livermore National Laboratory*, Livermore, USA
*{peng8,pearce7,gokhale2}@llnl.gov

***Abstract*—Large-scale high-performance computing (HPC) systems consist of massive compute and memory resources tightly coupled in nodes. We perform a large-scale study of memory utilization on four production HPC clusters. Our results show that more than 90% of jobs utilize less than 15% of the node memory capacity, and for 90% of the time, memory utilization is less than 35%. Recently, disaggregated architecture is gaining traction because it can selectively scale up a resource and improve resource utilization. Based on these observations, we explore using disaggregated memory to support memory-intensive applications, while most jobs remain intact on HPC systems with reduced node memory. We designed and developed a user-space remote-memory paging library to enable applications exploring disaggregated memory on existing HPC clusters. We quantified the impact of access patterns and network connectivity in benchmarks. Our case studies of graph-processing and Monte-Carlo applications evaluated the impact of application characteristics and local memory capacity and highlighted the potential of throughput scaling on disaggregated memory.**

***Index Terms*—Disaggregated Memory, Memory Utilization, Remote Paging, Remote Memory**

I. Introduction

Compute nodes are the basic unit of today's HPC systems. Compute and memory resources are tightly coupled in each node, and users request resources in the unit of a node. However, in the past two decades, processor and memory technologies are advancing at a diverging rate. The transistor speed doubled every 18 months before the Dennard scaling ended, and the number of transistors continues increasing. In contrast, memory latency nearly stagnates, and DRAM technology has dominated for decades. Nevertheless, on tightly coupled architecture, these two resources need to be upgraded together. Recently, disaggregate architecture starts shifting the paradigm of resource deployment and allocation in data centers and clouds [1]–[4]. Different resources can be upgraded independently on a disaggregated system, and users can request resources at a fine granularity.

Disaggregated memory systems, such as network-attached memory, are one popular disaggregated architecture [3], [5]–[7]. Memory resources are shared among jobs at the rack- or system-level. Therefore, overall memory utilization on the system improves while individual jobs' performance may degrade with increased memory accesses over the network. Previous studies mostly focused on data center infrastructure and workloads. In this study, we consider HPC environments and aim to answer the following questions: Can disaggregate memory benefit HPC systems? How to support existing applications to leverage disaggregate memory? What application characteristics are critical for performance?

We performed a large-scale study to understand how exiting workloads on production HPC systems utilize the memory resources. For this, we analyzed more than two million jobs on four HPC clusters. Our results show that most jobs only utilize a small fraction of memory resources, and for more than 90% time, a node utilizes less than 35% memory capacity. One possible implementation of disaggregated memory systems is as a part of a cluster, where nodes have slim memory to satisfy most jobs while memory-intensive jobs can still have sufficient memory resources from memory attached to the network.

HPC applications and environments impose constraints different from data centers, e.g., node sharing among jobs and kernel modifications in jobs are rare. In this work, we provide a user-space library to enable existing applications to explore disaggregated memory resources on HPC systems. The users have the flexibility of selecting data objects that may be partially placed on the remote memory to study the impact on performance. Transparently, the runtime manages data migration to fetch and evict memory pages from memory servers. We emulated a disaggregated memory system on an HPC cluster and studied performance-critical characteristics, including access patterns, local/remote memory ratios, and network connectivity.

We summarize our contributions as follows.

- We performed a large-scale survey on four production HPC clusters to quantify memory underutilization on existing HPC systems;
- We investigated constraints and potential deployment models of disaggregated memory on HPC systems
- We designed and developed a user-space solution for applications to explore disaggregated memory on existing HPC systems
- We emulated a disaggregated memory system and performed sensitivity studies on access patterns, migration granularity, and network connectivity
- We case studied two memory-intensive applications to evaluate the potential of throughput scaling on disaggregate memory

II. Memory Utilization on HPC Systems

We perform a large-scale study to understand the memory utilization on existing HPC systems. We select four production

TABLE I: Evaluated HPC Clusters

Name	Processor	CPU	GPU	Memory	Network	Nodes	OOM
Quartz	Intel Xeon E5-2695	36	N.A	128 GB	IB QDR	2604	5.4
Catalyst	Intel Xeon E5-2695 v2	24	N.A	128 GB	IB QDR	324	5.3
Pascal	Intel Xeon E5-2695 v4	36	2 Tesla P100	256 GB	IB EDR	163	14.8
Surface	Intel Xeon E5-2670	16	2 Tesla K40	512 GB	IB FDR	158	14.4

clusters at Lawrence Livermore National Laboratory and perform analysis on over two million job records. The selected clusters feature different memory systems – The number of nodes ranges from 163 to 2604; Two systems are equipped with GPU accelerators; Memory resources range from 128 GB to 512 GB per node. All the clusters use the slurm workload manager for job scheduling. Table I summarizes the main configurations of these clusters. We combine job-level information and system-wide counter samples to quantify the memory utilization on these clusters.

A. Workload Memory Utilization

The first characterization of memory utilization uses the job-level resource usage captured in the slurm database. We process three-month job records. Each record captures various resource usage, such as memory, I/O, and network, for the job. For this study, we mainly use six fields of the job information, i.e., the number of nodes, the maximum and average virtual memory, the maximum and average resident set size, and the number of tasks, for the quantitative study. Depending on the application executed in a job, we categorize jobs into three categories, i.e., scientific simulations (SIM), machine learning (ML), and data analytics and visualization (DA). Our analysis shows that the majority (99%) of high-memory-usage jobs (the top 1% memory usage) on these clusters are running scientific simulations. Therefore, these clusters are traditional HPC systems. Note that a machine-learning or data analytics dominated cluster is likely to exhibit different behavior in memory utilization.

We quantify the number of jobs that are limited by the node memory capacity on these HPC clusters to understand whether memory is a over-provisioned or scarce resource on exiting systems. We identify these jobs from the out-of-memory (OOM) exit code captured in the logs. For the comparison across clusters, we normalize the number of OOM jobs in every 10,000 jobs and report the OOM rate for each cluster in Table I. The results show that only a small fraction of jobs, i.e., 0.05% to 0.14%, cannot fit in the memory on these HPC clusters. Note that the low OOM rate could be a result of users adapting their jobs to the hardware configuration on each cluster. We also note that many jobs have a high aggregate memory footprint on all allocated nodes, but low memory consumption on a single node. Since the primary incentive for users is to shorten time to solution, jobs of good scalability are distributed over a large number of nodes to accelerate execution, resulting in low memory utilization per node.

We quantify memory utilization per node in jobs to understand the requirement of memory capacity from the jobs

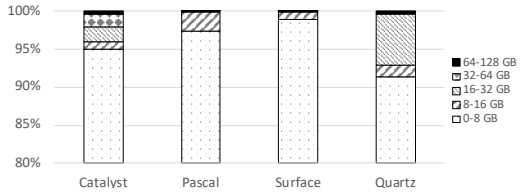


Fig. 1: The distribution of job memory utilization on four HPC clusters.

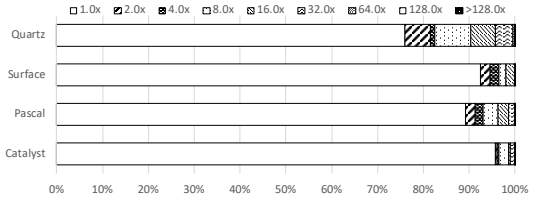


Fig. 2: The distribution of memory imbalance in jobs on four HPC clusters.

running on today's HPC clusters. Each job launches N_t parallel tasks on N_n compute nodes. We use the maximum resident set size in the N_t tasks to estimate the size mapped into the physical memory. Eq. 1 calculates the approximated memory utilization per node for a single job i.

$$\overline{Util_i^{mem}} = \frac{S_{rss} * N_t}{N_n * S_{mem}}, \tag{1}$$

where S_{mem} is the node memory capacity. Based on the memory utilization of all jobs, we analyze the distribution of $Util_i^{mem}$ among all jobs on four clusters, respectively. Figure 1 reports the histogram of job memory utilization. Three clusters, catalyst, pascal, and surface, have smaller memory utilization compared to the Quartz cluster. Despite surface and pascal have the largest memory capacity, their memory utilization is similar to other clusters. One reason for the low utilization on the two GPU-accelerated clusters could be that users partition input problems over multiple nodes to fit in the GPU memory per node. Quartz has a significantly larger fraction of jobs using more memory than the other clusters. Still, 92% jobs on Quartz uses less than 15% of the memory capacity.

We quantify the memory imbalance among nodes in a job to understand how much memory resource is overprovisioned due to the peak estimation. Eq. 2 evaluates the memory imbalance by normalizing the highest memory utilization with the average memory utilization for a job i.

$$Imb_i^{mem} = \frac{\max S_{rss}}{\overline{S_{rss}}} \tag{2}$$

We summarize the distribution of memory imbalance in all jobs in Figure 1. Note that $Imb_i^{mem} = 1$ means the memory utilization is well balanced among all nodes. We find that the Quartz cluster has more memory imbalanced jobs than the other clusters, i.e., 20% jobs on Quartz has 2-8 times memory imbalance. Note that on HPC systems, the memory

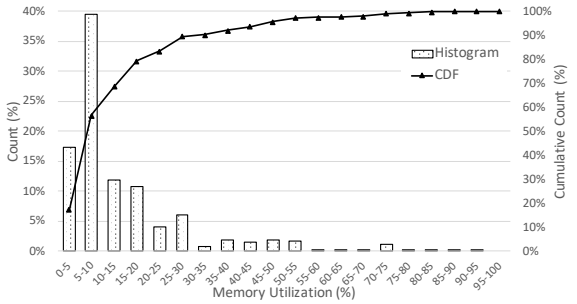


Fig. 3: System-wide memory utilization on the Quartz cluster for a six month period.

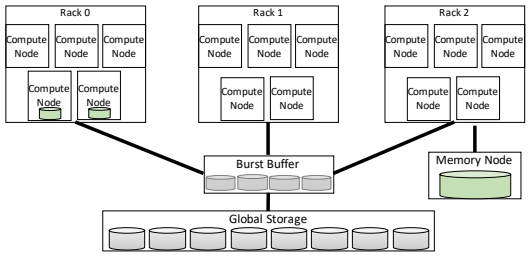


Fig. 4: Illustration of two potential implementations of disaggregated memory on HPC systems. Rack 0 and 2 represent a distributed architecture and a rack-level centralized architecture, respectively.

resources are requested in the unit of a node. A job with a high memory imbalance among nodes will have memory resources underutilized on most nodes.

B. System-wide Memory Utilization

Our second characterization focuses on node-level memory utilization. This analysis quantifies the system-wide memory utilization on each node to understand how compute nodes utilize memory resources over time across various jobs. We leverage the LDMS monitoring tool [8] that collects performance counters periodically on each node on the Quartz cluster and saves to a Cassandra database. We use the samples collected in one year span. From the *meminfo* and *memfree* counters, we quantify the total memory in use, including both user-space and OS memory usage on a node [9]. In T sampling periods, the average memory utilization is calculated by Eq. 3.

$$\overline{Imb_i^{mem}} = \frac{\sum_{i=1}^{i=T}(MemTotal - MemFree)}{\sum_{i=1}^{i=T} MemTotal} \quad (3)$$

Figure 3 reports the histogram and the cumulative distribution function (CDF) of the hourly memory utilization. Most of the time, i.e., 40% of the evaluated period, the hourly utilization is between 5-10$\%$ as indicated by the second bar in Figure 3. For 90% of the time, the memory utilization is below 35%, as indicated by the CDF line in Figure 3. The 90-percentile is consistent with a previous study on other HPC systems [9]. Memory underutilization is common on HPC systems because users are driven by the time to solution, and job allocation is based on the peak usage, i.e., the maximum memory usage per node. Consequently, only a few nodes in a job may utilize the allocated memory resources fully, which is even unlikely if jobs have severe memory imbalance. Fine-grained resource allocation is challenging, if not impossible, on the current architecture of tightly coupled resources on existing HPC systems. The long tail in the histogram (Figure 3) indicates that the chances for a compute node to exhaust its memory resource are low. For instance, memory utilization above 55% occurs in less than 2% of the time. Note that the unutilized memory increases procurement cost and operating cost because DRAM consumes static power even when idle, and the static power increases proportionally with the memory capacity.

III. Disaggregated Memory in HPC Systems

Generally speaking, a disaggregated memory system could have distributed or centralized architecture. In a centralized architecture (Rack 2 in Figure 4), abundant memory resources are available on memory servers/nodes. Each compute node has a small local memory. If a compute node needs more than its local memory, it requests from the memory servers. High-density and high-performance non-volatile memory technologies [10] are attractive for implementing memory servers. The distributed architecture (Rack 0 of Figure 4) has no dedicated memory nodes. Instead, each node may have a moderate memory capacity. If a compute node exhausts its local memory, it 'borrows' memory from some other nodes.

Various software approaches have been proposed for applications to utilize disaggregated memory systems. Operating system-based solutions [6], [7] typically require no application modifications and swap in pages from remote memory to local memory transparently. Hypervisor extensions [1] on cloud support virtual machines to leverage remote memory. However, HPC systems rarely co-locate jobs in one compute node, unlike data centers and cloud. Also, most HPC systems lack the support for changing the OS in each job allocation. Therefore, in this work, we explore a user-space solution rather than an OS-based solution for disaggregated memory systems on HPC systems.

We envision disaggregated memory as an in-transit service in the HPC environment, i.e., tasks launched in a job provides the service to other tasks in the same job. This deployment model works for both distributed and centralized implementation in Figure 4. In Rack 0, tasks on a node may borrow memory from other nodes. In Rack 2, tasks on compute nodes may request memory from tasks running on the memory blade attached to the rack. We consider disaggregated memory systems as a part of complex memory hierarchies, where a node has fast but small *local memory* and large but slow *remote memory*. The user would have the flexibility to select *remote data objects* that might be partially migrated to memory nodes.

IV. A User-space Remote Paging Library

In this work, we design a user-space remote paging library called *rMap* to allow applications exploring disaggregated

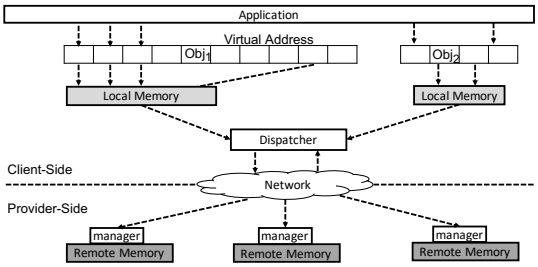


Fig. 5: An overview of *rMap* architecture.

memory. A compute node on a disaggregated memory system has only a small local memory. The objective is to enable memory-intensive workloads, whose memory footprint exceeds the local memory, to run on such a compute node as if on a huge-memory node. When an application exhausts the local memory on a node, *rMap* fetches memory pages on-demand from a memory node/server and transparently swaps into the local memory. Compared to global shared memory systems like PGAS, no changes in the programming model are required, ensuring the maximum reuse of existing efforts in most HPC applications.

Figure 5 illustrates the library architecture, including client-side and provider-side components. The client-side component resides in application processes while the provider-side component resides on memory nodes. An application may select multiple data objects to be remote, i.e., *remote data objects*. These data objects could be those that consume the most memory footprint. *rMap* manages each remote data object as a collection of equal-sized data chunks. When the application accesses a data chunk not in the local memory, the client-side dispatcher forwards requests to provider-side components, which then transfer the required data chunks to the client-side. If the local memory is exhausted, unused data chunks may be evicted.

A. Remote Region Management

An application can define multiple remote data objects. Each object, e.g., obj_1 and obj_2 in Figure 5, reserves a virtual memory address space that could be larger the local memory capacity. The virtual memory space assigned to each remote data object is called a *region*. Each region is divided into multiple *chunks* that is the basic unit for placement and migration. The size of a chunk is a configurable parameter incremental at the system page size. When a process accesses a chunk that is not in the local memory, the access is blocked, and *rMap* gets notified. Then, *rMap* looks up for the memory server owning the chunk and requests for it. Once the fetched chunk is placed into the local memory, the access is replayed. One optimization to accelerate write access is to postpone handshaking with the provider-side by directly allocating local memory to back updated data chunks.

rMap distributes the local memory for backing each remote data object based on demand. For instance, a smaller remote data object could have more chunks in local memory if they are accessed more frequently than a larger object. Local memory on disaggregated memory is a scarce resource for caching remote data objects. In *rMap*, the amount of local memory that can be used for remote data objects is also a configurable parameter. Different applications with different access characteristics may not benefit equally from increased local memory. Also, for multi-program workloads, each could be assured with a certain amount of local memory to avoid interference. When there is no local memory to accommodate newly accessed chunks, *rMap* evicts chunks by LRU approximation to the memory server.

B. Chunk Placement and Metadata

We design *rMap* for a multi-server multi-client deployment model. Remote regions in one application process may be backed by several memory nodes. One memory server could provide memory resources to multiple clients, i.e., application processes. This model naturally supports MPI applications – each rank is a client in *rMap*. It is portable on both centralized and distributed disaggregated memory, i.e., memory servers can be distributed on one or multiple memory nodes. Another benefit of the multi-server multi-client model is throughput, where concurrent accesses to chunks in one data object could have an aggregate bandwidth leveraging the high-performance network on HPC systems if they are distributed on multiple servers.

rMap maintains metadata for data chunks in the client-side component to track their ownership on the memory servers. As the data chunk is the basic unit for migration over the network, data in one chunk must be stored one the same memory node while different data chunks could be placed onto different memory nodes. The distribution of data chunks in one region can be either uniform or irregular over memory servers. If equal-sized sub-region are placed on servers, chunk lookup only requires simple offset calculation. However, a uniform distribution assumes that each memory node has the same amount of memory resources. In realistic scenarios, each memory node may have different memory resources, and those with more resources will hold more data chunks. Thus, for irregular distribution, *rMap* uses a hash table to bookkeep subregions, their start and end, and their owner servers.

C. Remote Request Pipeline

rMap separates communication with memory servers from region management. A client-side dispatcher collects requests for chunks from all regions into a central request pool. The dispatcher may explore opportunities to optimize these requests before distributing them to a group of workers. A group of lightweight workers shares the workload in the pool. These workers achieve high concurrency and load balancing through a dynamic scheduling scheme. Each worker handles a request by setting up the communication protocol and local memory for transferring data chunks. The coordination between the dispatcher and workers is through a FIFO pipe for minimum synchronization. Only the dispatcher update at the tail of the pipe while workers update the head.

We design the provider-side components to have the control of data transfer. Being near the data source, they can coordinate and manage data more effectively. In contrast, the client-side components only have information on their local requests, lacking a global view as the provider-side. Therefore, once a request is forwarded from the client-side, the worker is blocked, yielding resources to other threads ready to progress. The worker is wakened up when the requested data chunk arrives. It then continues to finish up data copy and release resources.

D. Implementation

We implement the proposed design in a C library[1] based on a user-space paging service [11]. *rMap* provides a minimal set of API for existing HPC applications to define selected remote data objects. Internally, *rMap* uses the userfaultfd [12] system call to register the regions of these remote data objects. When an accessed chunk is not in the local memory, the kernel sends notifications, in the form of page faults, to the user-space handler. The handler then determines the chunks that need to be fetched and their owner servers. Before dispatching the request, the handler may also evict unused chunks from the local memory to ensure space for the newly requested chunk. The workers are implemented using *pthread*. The library supports applications to specify the level concurrency of the workers.

The communication between client-side components and provider-side components uses a remote procedure call (RPC) library [13]. This communication layer supports multiple transport protocols commonly supported on HPC systems, such as RDMA, MPI, and TCP. When launching an application, the server-side could pre-process remote data objects before publishing their memory resources as a tuple of network address and memory object id. The client-side component then uses the published connection information to request for remote memory.

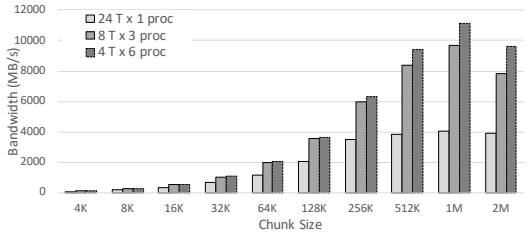
V. Evaluation

In this section, we emulate a disaggregated memory system and perform sensitivity tests and case studies.

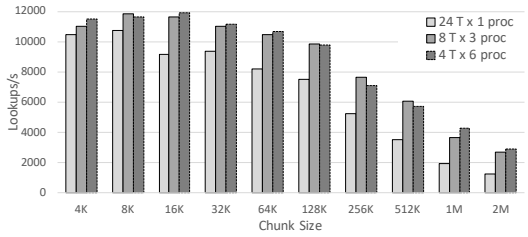
A. Emulation of a Disaggregated Memory System

We emulate a disaggregated memory system on an existing HPC cluster called Flash. Each node has 256 GB RAM, two Intel Xeon CPU E5-2670 processors on two sockets, and each with 24 cores (two hardware threads). The cluster uses Mellanox EDR InfiniBand interconnect. For emulation, each run limits the maximum local RAM that can be used by an application. If local RAM is exhausted, the library evicts local pages by LRU replacement policy and fetches remote pages from a 'memory node' mimicked by another node. The system runs the Red Hat Enterprise operating system with GNU/Linux 3.1.0. The cluster runs the slurm job scheduler. Each experimental run is deployed in a single job to mimic the deployment model on HPC systems.

[1]https://github.com/LLNL/umap/tree/remote_region



(a) Stream



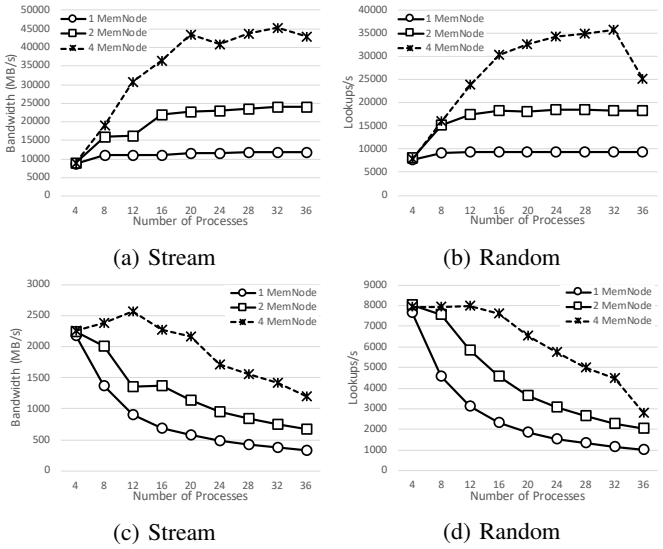
(b) Random

Fig. 6: The performance of Stream-Add and Random-Lookup benchmarks at increased chunk sizes at three configurations.

We use two synthetic benchmarks, i.e., stream and random, for sensitivity study. The first benchmark performs streaming access to each element in an array while the second one randomly looks up elements. Our first case study uses a graph-processing application derived from the Graph500 [14] that performs a breadth-first search (BFS) on an input graph. The second case study uses XSBench [15] from the ECP proxy app suite. The selection of remote objects currently requires programmers to change the allocation sites to the library APIs. Several patterns would make a data object more suitable for remote stores, e.g., large, long lifespan, read-intensive, and data reuse. Automatic identification of remote data objects is beyond the scope of this paper but discussed in [16]. The applications are compiled with GCC 8.3.1 and OpenMPI/4.0.0, and the geometric mean of the application-defined figure of metric (FoM) is reported.

B. Sensitivity Study

We evaluate the impact of chunk size and access patterns of remote data objects in two benchmarks. The first benchmark is ported from the STREAM benchmark. It defines two arrays as remote data objects and changes the chunk size from 4KB to 2MB. The experiments perform the stream kernels on one node and dedicate remote memory on another node. We report the measured performance in Figure 6a. In general, the streaming access pattern achieves higher performance at larger chunk sizes. Such remote data objects benefit little from data cached in local memory unless the objects can be fully cached in local memory. Instead, the most performance benefit comes from prefetching from remote memory to local memory, which is equivalent to increasing chunk sizes. Therefore, we observe higher throughput at larger chunk sizes. When the chunk size is too large to have enough computation to overlap the data transfer time, performance decreases, i.e., 2MB chunks



(a) Stream (b) Random

(c) Stream (d) Random

Fig. 7: The aggregate (top) and individual (bottom) performance at increased processes on 1, 2, 4 memory nodes.

achieved lower throughput than 1MB chunks in Figure 6a. The threading configurations start impacting performance at larger chunk sizes (64KB and above), and using fewer threads per process (4T in the test) improves performance significantly.

The random lookup benchmark in Figure 6b reports higher performance at smaller chunk sizes. This benchmark performs one million random lookups on a remote data object. Random access features low data reuse in the local memory as the data chunks fetched into the local memory may not be reused. Also, prefetching data into the local memory only detriment the performance because each lookup only accesses a small fraction of a data chunk. Therefore, a large chunk has more wasted data movement than a small chunk. We observe the peak performance at the smallest chunk size. Like the stream benchmark, the threading configurations start showing performance impact at larger chunk sizes (16KB and above). However, using 4T and 8T has less performance difference than that of the stream benchmark.

We evaluate the impact of network complexity by scaling up the number of processes and using different numbers of memory nodes. Figure 7a and 7b report the aggregate throughput and Figure 7c and 7d report the individual throughput. In general, the aggregate performance scales up as the number of processes increases while the number of memory nodes determines the peak throughput. For instance, the peak throughput at four memory nodes is about four times throughput at one memory node in both benchmarks. Also, random access is more sensitive to the increased network complexity, i.e., at four memory nodes, its performance drops considerably at the largest number of processes. This performance degradation is likely caused by network traffic contention.

Applications that exhibit random access may benefit from throughput scaling on disaggregated memory systems. Figure 7b and 7d show that the aggregate throughput continues scaling up to 32 processes even though the individual throughput starts decreasing from 16 processes. In contrast, the aggregate throughput of the stream benchmark (Figure 7a and 7c) stops scaling at 20 processes, while the individual throughput start decreasing at 16 processes.

C. Case Study I: Graph Processing

The first case study uses a data-analytics workload that performs breadth-first-search on a graph. Graph processing plays a vital role in social networks, bioinformatics, and data mining. The most memory-intensive data objects in graph applications are often related to the edges and vertices data structures. Other data objects often have relatively small sizes and short lifespans. We changed the allocation routine of the graph and edge structures to the *rMap* APIs. Four input graphs, i.e., scale 27, 28, 29, and 30, generated from a Kronecker generator as specified in Graph500 [14], are used for evaluation. These graphs have 4.3 to 34 billion edges and up to one billion vertices.

The disaggregated implementation enables a new processing paradigm that is different from the traditional distributed-memory implementation. Each process can perform an independent BFS search on the graph. In contrast, in the distributed implementation, all processes collectively work one search. For instance, the scale 30 graph cannot fit into a single node. Without the disaggregated implementation, a conventional distributed implementation would require at least two nodes for each search. One advantage of this execution model is to be able to initiate multiple different searches and early terminate if one search finishes fulfilling the requirement, e.g., model parallel.

The disaggregated implementation achieves throughput scaling as the number of processes increases. The experiment uses two memory nodes and increases the number of processes performing BFS from one to eight nodes. Figure 8 reports the aggregate throughput. The throughput scales up to six nodes, reaching different peak throughput on different inputs. Before the aggregate throughput stabilizes, the increased throughput indicates that the memory servers' peak capability has not been exhausted. Note that our emulation is based on the current hardware. Specialized hardware support for memory disaggregation will likely enable even higher aggergate performance.

One question in designing disaggregate memory is to determine cost-effective local memory capacity. We evaluate the impact of local memory by sweeping seven configurations of local memory capacity in Figure 10 that sets up 16 GB to 64 GB local memory. This experiment performs BFS search on four nodes and uses two memory nodes. We find that increasing the local memory capacity not always bring performance improvement. The workload is highly data-driven – some dense regions in the graph may have high access intensity while other sparse regions may have low access intensity. With offline profiling tools to identify the dense and sparse regions of a graph, our user-space solution could support data-aware optimization that explicitly pins dense regions in the local memory and reduce the capacity of local memory.

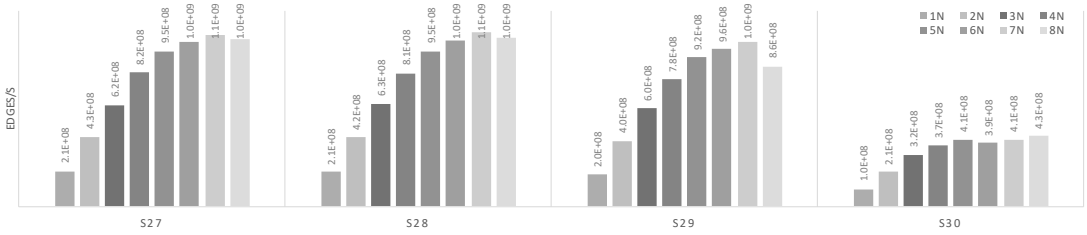


Fig. 8: The aggregate throughput of BFS on four input graphs (s27-s30) using increased processes (from one to eight nodes).

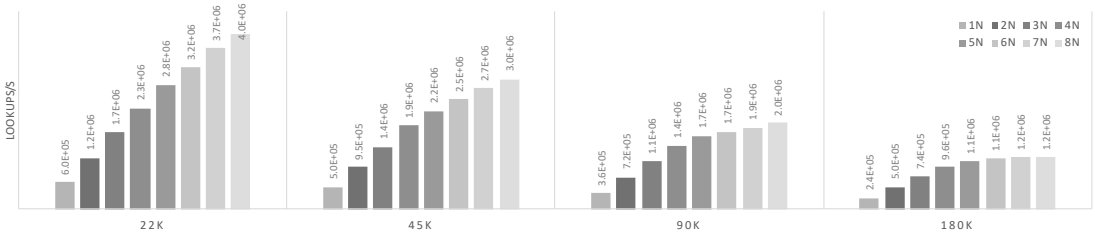


Fig. 9: The aggregate lookup rates in XSBench on four nuclide grids using increased processes (from one to eight nodes).

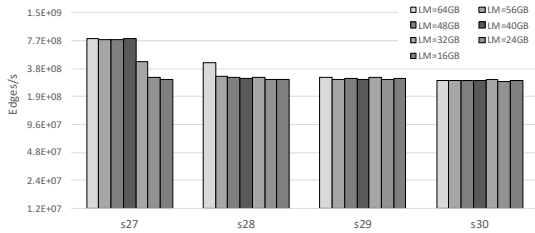


Fig. 10: The performance of BFS at different configurations of the local memory (LM) capacity.

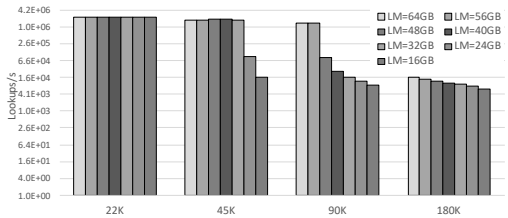


Fig. 11: The performance of XSBench at different configurations of the local memory capacity using four nuclide grids.

D. Case Study II: Monte Carlo Simulation

XSBench is a proxy app of the Monte Carlo neutronics application OpenMC. The main simulation phase performs lookups of the macroscopic neutron cross-sections. In this case study, we ported the MPI+OpenMP implementation to disaggregated memory systems. We select the nuclide grids and energy grids as the remote data objects by changing their allocation sites to *rMap* APIs. These two data objects consume more than 90% of the memory footprint. We evaluate 22K to 180K grid points per nuclide for 355 nuclides, and report the aggregate lookup rates in Figure 9.

The disaggregated performance of XSBench scales up almost linearly as the number of nodes increases (Figure 9). The processes that perform the cross-section calculation are mostly data-parallel without inter-process communication. Also, the remote data objects are read-intensive, imposing minimal overhead for consistency semantics. Therefore, using more processes scales up the aggregate performance until reaching the memory nodes' peak capacity.

We noticed that the aggregate throughput decreases across the input problems when the size of grid points increases from 22K to 180K in Figure 9. From the profiling results, we find that the nuclide grids have relatively small reuse distance. The energy grids have about 99% random accesses and only about 1% streaming-like accesses. Therefore, the data reuse in energy grids is low. Unlike BFS, which has data-dependent access to the remote data objects, random access to the energy grids has a uniform distribution. Therefore, the probability of an accessed chunk of the energy grids in the local memory is proportional to the ratio between the local memory size and the remote object size. In Figure 9, the size of local memory is constant, while the size of remote objects increases from 22K to 180K grid points. Therefore, we observe a decreasing aggregate throughput across the four input problems. For the same reason, when sweeping seven configurations of the local memory capacity (Figure 11), the performance always improves when the size of local memory increases. Unlike BFS, applications like XSbench generally benefit from increased local memory.

VI. Related Work

Various hardware designs [5], [17]–[21] for disaggregated systems have been proposed. The Machine from HPE [5] uses optical networking and fabric-attached memory to provide a globally accessible memory pool. Previous Mellanox nbdX [18] and currently NVMe over Fabrics [17] provides network-attached block device as disaggregated storage. Several data centers have employed disaggregated architecture at

production clusters, such as Intel's Rack-scale architecture [19] and Facebook's Disaggregated Rack [20]. Our study considers HPC environments and proposes a deployment model portable for potential architecture of disaggregate memory systems.

Software approaches for utilizing disaggregated memory systems include OS extension, hypervisor, and libraries. Lim et al. [1] identifies new replacement policies for disaggregated memory and explored the Memcached system. Gao et al. [2] evaluates the requirements on network bandwidth, latency, and protocols to support acceptable application performance. FaRM [22] leverages communication primitives in RDMA to reduce latency in accessing remote memory and propose a transactional interface in applications. Infiniswap [7] provides a kernel-based solution to enable remote paging transparently without application modifications. Hailstorm [4] developed a distributed LSM database deployed on a network-attached storage pool to mitigate load imbalance from tasks. Zhu et al. [23] explored storage disaggregation at the user-level by leverage NVMe over fabric for machine learning workloads. Allcock et al. [3] evaluates a Kove XPD memory appliance that is attached to the network and ported memory-intensive applications to their specific APIs. Many of these works target data centers and clouds environment and commercial workloads. Our work addresses the constraints in HPC environment, targets to minimize programming efforts, and supports user control and flexibility at user-space.

VII. Conclusions

Current HPC systems are composed of tightly coupled resources in nodes. Recently, disaggregated architecture starts emerging to enable fine-grained resource allocation. In this study, we performed a large-scale study to quantify the memory utilization on four production HPC clusters. We provide a user-space library for applications to explore disaggregated memory in HPC environments. Based on the library, we emulated a disaggregated memory system. Our experiments evaluated the performance impact of access patterns, migration granularity, network connectivity, and local memory size on application throughput and highlight design and optimization insights in two case studies.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract No. DE-AC52-07NA27344. This research was supported by the Exascale Computing Project (17-SC-20-SC). LLNL-CONF-809199.

References

[1] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch, "System-level implications of disaggregated memory," in *IEEE International Symposium on High-Performance Comp Architecture*. IEEE, 2012, pp. 1–12.

[2] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network requirements for resource disaggregation," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 249–264.

[3] W. Allcock, B. Bernardoni, C. Bertoni, N. Getty, J. Insley, M. E. Papka, S. Rizzi, and B. Toonen, "Ram as a network managed resource," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018, pp. 99–106.

[4] L. Bindschaedler, A. Goel, and W. Zwaenepoel, "Hailstorm: Disaggregated compute and storage for distributed lsm-based databases," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 301–316.

[5] HPE, "Hpe the machine," 2020. [Online]. Available: https://www.labs.hpe.com/memory-driven-computing

[6] S. Liang, R. Noronha, and D. K. Panda, "Swapping to remote memory over infiniband: An approach using a high performance network block device," in *2005 IEEE International Conference on Cluster Computing*. IEEE, 2005, pp. 1–10.

[7] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap," in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, 2017, pp. 649–667.

[8] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos *et al.*, "The lightweight distributed metric service: a scalable infrastructure for continuous monitoring of large scale computing systems and applications," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 154–165.

[9] G. Panwar, D. Zhang, Y. Pang, M. Dahshan, N. DeBardeleben, B. Ravindran, and X. Jian, "Quantifying memory underutilization in hpc systems and using it to improve performance via architecture support," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 821–835.

[10] I. B. Peng, M. B. Gokhale, and E. W. Green, "System evaluation of the Intel Optane byte-addressable nvm," in *Proceedings of the International Symposium on Memory Systems*, 2019, pp. 304–315.

[11] I. Peng, M. McFadden, E. Green, K. Iwabuchi, K. Wu, D. Li, R. Pearce, and M. Gokhale, "Umap: Enabling application-driven optimizations for page management," in *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. IEEE, 2019, pp. 71–78.

[12] L. kernel, "Userfaultfd," https://www.kernel.org/doc/Documentation/vm/userfaultfd.txt, 2020.

[13] J. Soumagne, D. Kimpe, J. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. Ross, "Mercury: Enabling remote procedure call for high-performance computing," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2013, pp. 1–8.

[14] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray Users Group (CUG)*, vol. 19, pp. 45–74, 2010.

[15] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "Xsbench-the development and verification of a performance abstraction for monte carlo reactor analysis," *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[16] I. B. Peng, R. Gioiosa, G. Kestor, P. Cicotti, E. Laure, and S. Markidis, "RTHMS: A tool for data placement on hybrid memory system," *ACM SIGPLAN Notices*, vol. 52, no. 9, pp. 82–91, 2017.

[17] D. Minturn, "Nvm express over fabrics," in *11th Annual OpenFabrics International OFS Developers' Workshop*, 2015.

[18] mellanox, "Mellanox: What is nbdx?" 2020. [Online]. Available: https://community.mellanox.com/s/article/what-is-nbdx-x

[19] Intel, "Intel rack scale design," 2020. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html

[20] J. Taylor, "Facebook's data center infrastructure: Open compute, disaggregated rack, and beyond," in *Optical Fiber Communication Conference*. Optical Society of America, 2015, pp. W1D–5.

[21] V. Shrivastav, A. Valadarsky, H. Ballani, P. Costa, K. S. Lee, H. Wang, R. Agarwal, and H. Weatherspoon, "Shoal: A network architecture for disaggregated racks," in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 2019, pp. 255–270.

[22] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "Farm: Fast remote memory," in *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, 2014, pp. 401–414.

[23] Y. Zhu, W. Yu, B. Jiao, K. Mohror, A. Moody, and F. Chowdhury, "Efficient user-level storage disaggregation for deep learning," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2019, pp. 1–12.