MegaMmap: Blurring the Boundary Between Memory and Storage for Data-Intensive Workloads

Luke Logan
Illinois Institute of Technology
Chicago, IL, USA
llogan@hawk.iit.edu

Anthony Kougkas
Illinois Institute of Technology
Chicago, IL, USA
akougkas@iit.edu

Xian-He Sun
Illinois Institute of Technology
Chicago, IL, USA
sun@iit.edu

***Abstract*— Large-scale data analytics, scientific simulation, and deep learning codes in HPC perform massive computations on data greatly exceeding the bounds of main memory. These out-of-core algorithms suffer from severe data movement penalties, programming complexity, and limited code reuse. To solve this, HPC sites have steadily increased DRAM capacity. However, this is not sustainable due to financial and environmental costs. A more elegant, low-cost, and portable solution is to expand memory to distributed multi-tiered storage. In this work, we propose MegaMmap: a software distributed shared memory (DSM) that enlarges effective memory capacity through intelligent tiered DRAM and storage management. MegaMmap provides workload-aware data organization, eviction, and prefetching policies to reduce DRAM consumption while ensuring speedy access to critical data. A variety of memory coherence optimizations are provided through an intuitive hinting system. Evaluations show that various workloads can be executed with a fraction of the DRAM while offering competitive performance.**

***Index Terms*—HPC, Systems Software, Memory Tiering, Storage Tiering**

I. Introduction

Traditionally, memory and I/O substrates have been considered separate entities due to their differences in terms of performance and persistence. However, modern data-intensive memory-centric workloads widespread in HPC and Cloud are challenging these distinctions. Data analytics [1], machine learning [2], and deep learning [3] codes perform large-scale computations on data which greatly exceed the bounds of memory, relying on explicit data movements to I/O systems to meet basic capacity requirements. This often leads to significantly increased development complexity [4] and suboptimal, one-off solutions where I/O and compute happen in distinct, synchronous phases [5], incurring the memory wall [6] problem in the compute phase and the notorious I/O bottleneck [7] during the I/O phase. Conversely, scientific simulation codes [8] are becoming increasingly memory-intensive and are developed assuming large memory capacities are provided to avoid out-of-core development complexity. To reduce code complexity and I/O costs, HPC and Cloud sites have been steadily increasing total DRAM capacity so that datasets can be staged primarily in main memory [9]. However, while many applications desire an effectively infinite memory to generate and analyze massive datasets, the ever-increasing size of data [10] and the extreme financial and energy costs of DRAM [11], [12] make scaling DRAM capacity unsustainable.

To provide applications the capacity and bandwidth they require, heterogeneous storage is being explored. Modern storage accelerators (e.g., NVMe, Phase Change Memory, and Compute Express Link) have made significant advances in terms of capacity, bandwidth and latency, offering performance within an order of magnitude of DRAM while providing high density [13].

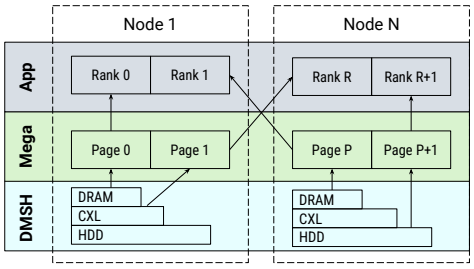

Fig. 1: Depiction of Coherent, Distributed, and Tiered MegaMmap Memory

These accelerators have already been adopted in several HPC sites. ORNL's Summit and Lawrence Livermore's El Capitan machines, for example, come equipped with high-bandwidth, low-latency, node-local NVMes, whereas Argonne's Aurora machine contains both NVMe and 3D XPoint modules. However, while modern storage provides great potential to improve performance, presenting this highly heterogeneous **Deep Memory and Storage Hierarchy (DMSH)** to applications remains an ongoing challenge.

In order to address the performance, programmability, and capacity limitations of distributed out-of-core algorithms, the semantic gap between memory and heterogeneous storage should be eliminated through a tiered, nonvolatile **Distributed Shared Memory (DSM)** abstraction. With this approach, datasets can be produced and analyzed in what appears as a single, coherent main memory, when in reality data is dispersed among both memory and tiered storage. This removes the burden of explicit I/O synchronization and memory management from developers, while simultaneously reducing data movement overheads by leveraging storage accelerators and allowing compute and I/O to execute concurrently. While several DSM solutions have been proposed and deployed in production Cloud centers [14], none currently provide transparent integration with tiered storage, persistence, and optimizations for HPC. In order to provide such an abstraction, several challenges must be overcome.

First, applications should be able to propagate their access pattern intention. Existing DSMs are unaware of the regions of shared memory that will be read or modified until it happens, which limits the effectiveness of prefetching and data placement algorithms [15] and causes extreme overheads to maintain memory coherence [16]. While this may appear as an imposition to users, many distributed and out-of-core algorithms are already structured to accomplish this, as they typically operate over large, well-defined subsets of out-of-core data between synchronization points, such as barriers and locks [17].

Second, memory coherence policies must be optimized to

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/$31.00 ©2024 IEEE

address the characteristics of HPC environments. DSMs are primarily designed to support chaotic, unpredictable multi-tenant Cloud workloads while remaining fault-tolerant, incurring prohibitive communication overheads to locate and invalidate replicas [17]. However, HPC applications are typically executed in isolation, do not require fault tolerance, and are oftentimes coordinated, where data structures are read and modified in well-defined phases [18] (e.g., producer-consumer workflows).

Third, new data placement and prefetching policies must be developed to support the DMSH and consider memory-centric application behavior. Existing DSMs operate only over a single tier and have no indication of access pattern. However, hardware in the DMSH is highly diverse, requiring considerations to device lifetime, access pattern, capacity, latency, and bandwidth. In addition, out-of-core workloads tend to operate over large datasets iteratively, where a large subset of the dataset is processed and then ignored [19]. This behavior offers opportunity to overlap data movement with the computation.

In this work, we present the design and implementation of MegaMmap: a new tiered, nonvolatile DSM for HPC that abstracts both remote memory and storage, effectively providing infinite memory capacity without sacrificing performance. MegaMmap provides a C++ library that leverages language features to provide a uniform, byte-addressable interface to present massive datasets as if they were in memory. A transactional memory API is provided, allowing applications to specify their intent to read or modify a region of data. Internally, MegaMmap manages the complexity of data placement among the distributed memory and storage hierarchy through the use of novel online prefetching and data organization algorithms. Factors such as randomness seeds and access intent are used to guide data organization decisions and ensure that global memory accesses do not frequently have to stall for remote memory or storage. MegaMmap provides a wide variety of intent-aware cache-coherency optimizations to minimize memory access latency for a variety of workloads. The contributions of this work are as follows:

1) A durable, persistent, and intuitive **Distributed Shared Memory (DSM) system**, which significantly reduces out-of-core development complexity by allowing applications to present massive datasets as memory objects.
2) A user-driven **transactional memory access API**, which leads to improved decision-making in cache coherence and data organization policies by propagating memory access intent.
3) A comprehensive set of **intent-aware memory coherence** optimizations, which improves the latency and bandwidth of memory accesses based on workload characteristics.
4) A wide variety of **tiered data organization** policies, which minimize I/O stall times by leveraging heterogeneous storage hardware and advance knowledge of access pattern intent.

II. Background and Related Work

Software DSM is being investigated as a method of reducing the programming burden of analyzing massive datasets in memory and sharing information between processes [20]. Much research has been conducted to discuss the latency considerations of cache coherency and the effectiveness of data placement and prefetching in shared memory systems.

Memory Mapped I/O (MMIO): MMIO [21] enables applications to share data across processes and present files as C-style arrays. In the persistent case, data from the file is automatically loaded from the storage system during a page fault. Modifications to the file are made in the OS page cache and then evicted either asynchronously or by an explicit synchronization call (e.g., msync or fadvise). MMIO simplifies programming out-of-core algorithms by removing the burden of deciding which parts of the file are buffered in main memory from the developer [21]. However, this approach traditionally suffers in performance due to frequent page faulting [22], little-to-no prefetching requiring complex mmap tracing [23], and significant I/O overhead caused by fixed (typically 4KB) page sizes [24]. The default page fault path in mmap can be customized to change prefetching and data placement policies by either editing the kernel directly or leveraging the userfaultfd [25] system call. Efforts are being made to improve the performance of MMIO for single-node and DSM systems, described below.

Single-Node Shared Memory: Several works have been proposed customizations to the traditional MMIO path to improve performance and resource utilization in single-node cases. uMMAP-IO [24], for example, utilizes this approach to extend virtual memory to SSDs and Hard Drives for HPC workloads, effectively increasing memory capacity. However, userfaultfd increases the cost of an individual page fault due to increased context switching between the kernel and userspace [26]. Many works aim to provide custom page fault handlers while avoiding userfaultfd [21], [27], [28], [29], [30]. Aquila [27] and FlatFlash [30], for example, make kernel modifications to reduce software overheads of the MMIO path. To avoid kernel modification, Tricache [29] uses LLVM to prepend a custom buffer cache handler before all CPU load/store operations, removing a physical page fault entirely. In their specific page handler code, NVMes are used as a temporary buffering layer via the Storage Performance Development Kit (SPDK). These works are focused on single-node, multi-threaded applications, not fully distributed, multi-process shared-memory applications.

Distributed Shared Memory (DSM): DSM allows applications to access remote memory as if it were its own memory. These works typically extend the virtual memory management subsystems of the OS to implement custom page fault handlers to provide the shared memory abstraction, although some use language features (e.g., operator overloading). A number of works have proposed designs and improvements to DSM systems. TreadMarks [16] uses virtual memory to provide a unified global address space, but suffers from extreme cache-coherence overheads. A1 [14] is a distributed in-memory graph database used by Microsoft to support the Bing search engine. A1 focuses on optimizing DSM latency for graph problems over commodity DRAM by using lightweight RDMA networks. Concordia [20] proposes a network protocol to reduce the overhead of DSM cache-coherency on fast networks by offloading the coherency protocols to programmable network switches. Leap [31] discusses a kernel-level algorithm to prefetch remote memory using RDMA into the Linux page cache. HotPot [17] explores a DSM that replaces DRAM with Intel Optane DC Persistent Memory (DCPMM), but does not investigate tiering algorithms. Alternatively, Grappa [32] provides specific optimizations of shared memory for graph-based and SQL-like workloads. Currently, DSMs focus on single-tier approaches for Cloud workloads, and have not yet explored the implications of expanding memory capacity to nodes with multiple tiers of storage.

Partitioned Global Address Space (PGAS) Languages: Alternative to DSMs, PGAS languages provide distributed

applications the illusion of a global unified memory address space through the use of language features [33]. The PGAS library manages the complexity of unifying the underlying disjoint memory systems and provides abstractions to allocate memory and inform data movement strategies [34]. To improve performance, PGAS libraries provide explicit APIs to fetch remote data as needed. A number of PGAS languages have been proposed, including OpenSHMEM [35], Charm++ [36], and UPC(++) [37]. UPC++ [37], for example, uses C++ to overload the address operator to abstract the complexity of mapping pointer offsets to physical memory locations. Synchronization primitives such as locks and futures are also provided. From these primitives, more complex objects such as unordered maps and queues can be implemented. Generally, these libraries have better performance than traditional DSMs, as they avoid page fault overheads and have finer control over message granularity and scheduling. However, developers are responsible for manually partitioning datasets into the PGAS library and tracking what data is local and remote, which can increase programming complexity and force strict requirements on application structure [32]. Lastly, PGAS, libraries currently only consider remote DRAM, and have yet to explore expansion to other storage tiers such as local and remote NVMe and HDD.

Non-DSM Tiered and PMEM Storage Management: Multi-tiering has existed for decades. Many prefetching and data placement methods that target caching between low-level CPU caches and main memory [38], [6] have been proposed. However, CPU caches have different hardware characteristics from storage such as Persistent Memory (PMEM) and NVMe [39], [40]. More relevantly, there has been some work on buffering between main memory and storage [7], [41], [42]. The Linux kernel provides the OS page cache, which enables I/O to be temporarily cached in main memory, while asynchronously shifting data to and from disk. However, the provided data placement and prefetching policies of the kernel only support two-tier caching (i.e., DRAM + HDD). Many PMEM filesystems [43], [44] have emerged that consider the interaction between DRAM and PMEM. While they abstract devices that are byte-addressible, the filesystems in general only support POSIX I/O and do not provide a DSM interface (nor MMIO) abstracting these devices. Other works have expanded buffering to N-tiers of storage. Hermes [7], for example, is an extensible distributed hierarchical buffering platform which transparently manages all local and remote storage tiers in order to provide increased bandwidth for HPC workloads. However, the existing data placement policies provided in Hermes focus on the I/O of high-bandwidth checkpoint-restart HPC workloads, not memory patterns in memory-centric workloads. In addition, these technologies are interfaced using I/O libraries such as POSIX and HDF5, and not through a global, byte-addressable shared memory abstraction.

Background Summary: Our extensive literature review indicates that no existing DSM system can effectively integrate memory and tiered storage to provide a unified, logical memory space for massive datasets. Existing DSM solutions focus only on local and remote memory, and do not consider intermediate storage. Solutions that provide byte-addressable abstractions over memory and storage simultaneously are only designed for single-node cases (e.g., MMIO) and cannot be directly applied to provide shared memory in distributed cases, as they do not consider the complexities of distributed memory coherence and metadata management.

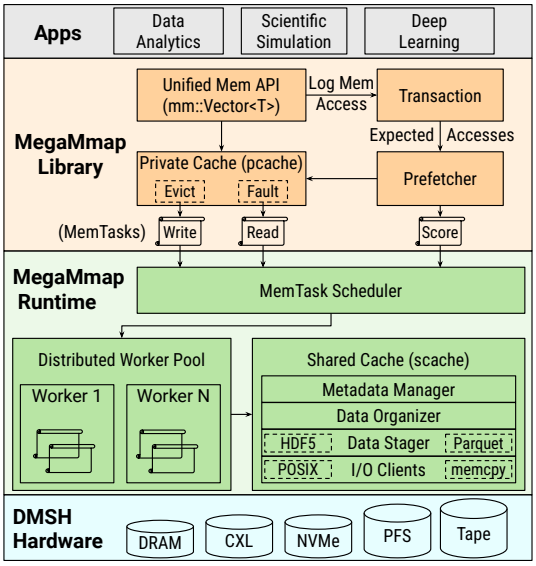

Fig. 2: MegaMmap Architecture

III. Design

MegaMmap is a fully distributed and tiered shared memory designed to provide modern scientific applications an efficient, intuitive, and portable way of sharing massive amounts of data between processes while hiding the complexity of data movement, cache coherence, and resource management. MegaMmap exposes byte-addressible APIs, similar to `mmap` and `std::vector`, allowing out-of-core datasets to be produced, presented, and analyzed in what appears as an infinite main memory. These APIs internally fragment and cache application data as pages of configurable size. Applications can specify their access intentions through a transactional memory API. Several cache coherence policies are provided through the use of a hinting system, reducing communication and data movement overheads for several common access patterns experienced in data-intensive workloads. To ensure DRAM is well-utilized and I/O stalls are minimized, pages are automatically evicted, prefetched, and organized in the DMSH through extensible prefetching and data organization engines. These engines can be tuned to the specific behavior of the application to offer maximum bandwidth and latency. We provide several such policies based on the memory patterns of several common HPC machine learning and simulation algorithms. The high-level architecture of MegaMmap is depicted in Figure 2. MegaMmap is designed with the following objectives in mind:

1) **Intuitive**: MegaMmap should hide the complexity of data movement for memory-centric, out-of-core workloads behind an easy-to-use memory API.
2) **Heterogeneous**: MegaMmap should provide algorithms that automatically manage the organization and coherency of data in a tiered environment.
3) **Semantic**: MegaMmap should provide developers a way to describe application intent to inform data organization policies.
4) **Portable**: MegaMmap should be compatible with a variety of application structures.
5) **Persistent**: MegaMmap should ensure that data is correct and persistent after a job ends.

```cpp
#include <mega_mmap/vector.h>

void KMeansInertia(std::vector<Point3D> &ks) {
  int rank = mpi::get_rank();
  int nprocs = mpi::get_comm_size();
  mm::Vector<Point3D> pts("/points.parquet");
  pts.BoundMemory(MEGABYTES(1));
  pts.Pgas(rank, nprocs);
  float distance = 0;
  tx = pts.SeqTxBegin(
    pts.local_off(), pts.local_size(),
    MM_READ_ONLY);
  for (Point3D p : tx) {
    distance += pow(NearestCentroid(p, ks), 2);
  }
  pts.TxEnd();
  return distance;
}
```

Listing 1: An example of MegaMmap for calculating inertia in KMeans. A nonvolatile, read-only shared vector `pts` maps an existing file named `pts.parquet`. The vector size is the dataset size (transparently queried using the parquet library) divided by the size of `Point3D`. The vector is limited to store 1MB of data in DRAM before evictions are triggered. The elements of the vector are logically partitioned evenly among processes (`Pgas`). A read transaction is initiated, where a for loop then iterates over the dataset and calculates the sum of squared distances (inertia).

A. An Intuitive and Portable Nonvolatile Shared Memory Abstraction

Programming distributed out-of-core algorithms requires significant developer effort to manage the capacity constraints of main memory [45], leading many data-intensive applications to divide into distinct, alternating, and synchronous phases of compute and I/O, where I/O is oftentimes the bottleneck [7], [46]. This conceptual separation between memory and storage leads to the ever-increasing DRAM capacity in modern computing machines to avoid such bottlenecks [47], causing severe environmental impacts due to high energy costs [48]. MegaMmap bridges this semantic gap between memory and storage by providing unified, coherent infinite shared memory abstractions. To avoid explicit I/O phases, MegaMmap provides a nonvolatile memory abstraction that transparently synchronizes data to arbitrary storage backends, such as a file on a PFS. Applications can inform MegaMmap of expected behavioral patterns through the use of a comprehensive transactional memory hinting API to optimize cache coherence and data movement policies. This section provides details on the specifics of the abstractions provided by MegaMmap.

An Infinite Shared Memory Abstraction: To give applications the illusion of a large memory, MegaMmap implements a shared memory vector API, providing implementations of several functions and operators including array index, memory copy, acquiring current size, appending, resizing, and destroying the data container. Processes connect to the shared vector using a semantic, user-defined key common to all processes. The vector API is highly versatile and allows more complex distributed data structures, such as matrices, logs, and multi-dimensional arrays, to be developed using simple offset calculations and appends. Through C++ templating, MegaMmap can theoretically store any type of data – including complex C++ classes, so long as a serialization method is provided. Unlike standard C++ vectors, MegaMmap shared vectors are not destroyed in the destructor – users must explicitly destroy them. This is to avoid the race condition where processes finish at separate times. Shared vectors can be either volatile or nonvolatile, where volatile vectors store temporary data that is difficult to fit in memory and nonvolatile vectors represent data that should eventually be persisted to storage for future use.

Presenting Persistent Datasets as Memory: Data-intensive applications either produce or analyze a persistent dataset that cannot fit trivially in memory. To reduce the burden of out-of-core programming for persistent data, MegaMmap allows nonvolatile shared vectors to be created with a backing persistent object (e.g., a file on parallel filesystem). In this case, the key of the vector is structured as a URL (i.e., "protocol://URI:params"), where all information needed to read and write the object, including factors such as identifiers and port number are provided. This format is versatile, allowing MegaMmap to represent a variety of data formats (e.g., HDF5) and repositories (e.g., Amazon S3) without requiring the application to commit to any particular library's semantics. For example, an HDF5 group can could be represented with the URL `hdf5:::///path/to/df.h5:mygroup`. Alternatively, multiple data objects, such as files produced in a file-per-process HPC simulation, can be mapped as a single uniform vector via a regex query such as `file:///path/to/dataset.parquet*`.

Enabling Resource Utilization Control: Balancing memory constraints typically requires programmers to be cognizant of the amount of data being allocated and the persistence requirements of the data, which is particularly challenging in out-of-core programs. MegaMmap provides explicit, fine-grained control over memory and storage capacity constraints. Applications can specify the maximum amount of DRAM and high-performance storage to use for caching using either the native C++ API or the MegaMmap configuration YAML file, which additionally contains settings regarding the nodes to deploy MegaMmap on, port numbers, etc. When the capacity constraints are reached, MegaMmap will automatically and transparently evict and reorganize pages in the DMSH.

Supporting Arbitrary Application Structures: There are many ways to structure distributed applications, such as parallel message passing and MapReduce. To remain compatible with any application structure, MegaMmap provides several synchronization options to ensure parallel application correctness. This includes distributed locks and barriers. These mechanisms can also be supplied by means of a higher-level library, such as MPI and UPC++.

Informing Policy with Transactional Memory: In order for memory tiering to perform well, applications must exhibit predictable behavior and convey that behavior to the underlying prefetching, eviction, and coherence algorithms. Transactions are used to inform MegaMmap of the access pattern a region of shared memory is about to incur and when it has completed within a particular process, accomplished through the transaction begin (`TxBegin`) and transaction end (`TxEnd`) methods. The `TxBegin` API includes hints on access method, such as read, write, and append. It also includes the indices that will be accessed. For sequential accesses, the offset and size of memory can be used. Modifications made between `TxBegin` and `TxEnd` are not required to be visible to other processes immediately. TxEnd commits all unflushed data that was not automatically flushed. Users can develop their own custom

```
struct PageRegion {
  size_t page_idx, off, size;
  bool modified_;
};
class Transaction {
  bitfield32_t flags;
  size_t head, tail;

  virtual std::vector<PageRegion>
  GetPages(size_t off, size_t count) = 0;

  auto GetTouchedPages() {
    return GetPages(head, tail - head);
  }
  auto GetFuturePages(size_t count) {
    return GetPages(tail, count);
  }
};
```

Listing 2: The transaction base class. `GetPages` predicts the regions of a page that will be accessed based on the number of memory accesses encountered (`tail`). `head` is the number of memory accesses acknowledged by the prefetching algorithm.

transactions by inheriting from the transaction class (shown in Listing 2) and using a templated `TxBegin` method. With this knowledge in advance, prefetching, eviction, reorganization, and coherence algorithms can be far more effective.

Use Case - KMeans: KMeans is a read-intensive clustering algorithm, where each process is given a non-overlapping subset of a dataset. In parallel, each training iteration locates the centroid nearest to a sample point in terms of a given distance metric (e.g., euclidian) and calculates inertia by summing the calculated distances. An example of how to apply MegaMmap to this workload is shown in Listing 1.

Use Case - Random Forest (RF): RF is a mixed workload, where each iteration builds a decision tree based on a random subset of features and samples. This practice is known as out-of-order bagging. Each process of RF randomly subsets a subsample of the whole dataset with replacement and decides the split point of the decision tree based on the entropy of each feature. The dataset is then divided into left and right samples based on the split point.

B. Actively Ensuring Data Persistence and Consistency

Storage accelerators in HPC sites are ephemeral, ending when the application's job ends. In order to ensure that data remains persistent, all cached data must be flushed out of these devices into a persistent backend, such as a parallel filesystem. In addition, data may be expected in a particular format, such as HDF5 or NetCDF. This requires data stored in cache to be converted into the application's desired format. To ensure data remains persistent after the job ends, MegaMmap actively flushes modified data to storage during periods of computation. This reduces the cost of explicit synchronization functions. MegaMmap also ensures that data is stored in its proper format by transparently and asynchronously staging data in and out of the backing storage. This section provides details on the overall lifecycle of memory pages in MegaMmap and the high-level architecture in Figure 2.

Distributed Heterogeneous Caching Structure: There are two page caches in MegaMmap: the `Private Cache (pcache)` and `Shared Cache (scache)`. The pcache is a DRAM-only cache of configurable maximum size that is stored per-process, while the scache is distributed, tiered, and coherent across multiple processes. Each application process is linked to the MegaMmap library, which internally stores the pcache and a queue for submitting `MemoryTasks` to the MegaMmap runtime, which is a process running separate from applications that manages the scache. The runtime can dedicate a configurable maximum number of CPU cores and dynamically adjusts the number of cores based on experienced load using an approach similar to LabStor [46]. When the application reads or writes to MegaMmap, it first goes through the pcache. If a memory page is not currently cached, a page fault will occur. If space is needed to service the page fault, an eviction will occur. Users can also explicitly flush the content of pages to ensure modifications are visible to other processes. During page fault, eviction, and flushing operations, the MegaMmap library constructs a MemoryTask that contains the subset of a page to read or update from the scache. The task will be placed in the queue and polled by the runtime, which will then be scheduled to a worker and executed.

Strong Consistency through MemoryTask Scheduling: MegaMmap allows distributed processes to interact with potentially the same data concurrently. It is possible that multiple processes intend to modify the same parts of data simultaneously. To provide **strong consistency** (i.e., read-after-write guarantees), the runtime's `Scheduler` ensures MemoryTasks for the same page are hashed to the same worker. To improve performance, MegaMmap allows reader tasks to proceed in parallel until a writer task is encountered. Writes can be executed concurrently if they are to non-overlapping subsets of the page. To further improve performance, the MegaMmap scheduler divides workers into low-latency and high-latency groups. MemoryTasks containing less than 16KB of data will be sent to low-latency workers, which are scheduled on different CPU cores from high-latency workers. This is to ensure that latency-sensitive requests are not stalled by large requests, which has been shown to improve quality-of-service in several applications [46].

Lifecycle of Modified Data: MegaMmap transactions mark a period of time when modifications made to memory do not need to be visible to other processes immediately. For this reason, MegaMmap uses copy-on-write semantics for handling memory modifications. Processes write to their local pcache first and have their own view of data. During the end of a transaction, an eviction, or an explicit user-driven call to flush, the process's local modifications are made visible to other processes. Since transactions store the exact memory accesses made, only the bits of the page that were modified during a transaction will be a part of the writer MemoryTask operation. This reduces I/O amplification and improves data correctness, since stale data will not be included in the MemoryTask operation updates. To further improve performance, writer MemoryTasks are constructed with a copy of the modifications and are then executed in a non-blocking, asynchronous fashion. During an eviction, the application will only experience the performance cost of a memory copy, instead of metadata updates, networking operations, and potentially an I/O operation. When the writer MemoryTask is scheduled on one of the Runtime's workers, the data organization engine will be invoked to decide the placement of data in the DMSH.

Read, Page Fault, and Prefetcher Path: When reading data from MegaMmap, either the data is present in the pcache or must be fetched from the scache. Read operations first check the pcache for existence. If a page is not present in the

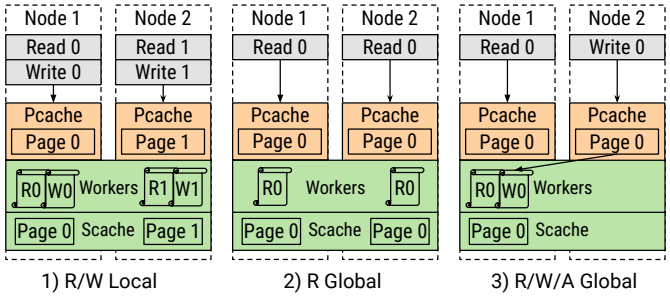

Fig. 3: Overview of coherence policies. Read (R) / write (W) to non-overlapping regions will first place data locally on the node that produced it without synchronization. Read-only will allow for data to be replicated in the shared cache to improve availability. Read, write, append global will hash page faults and evictions to the same worker.

pcache, a synchronous page fault occurs, which constructs a read MemoryTask and submits to the runtime. To reduce the frequency and cost of page faults, the MegaMmap library also provides a lightweight `prefetcher`, which constructs asynchronous score MemoryTasks based on the nature of the active memory transaction. The importance score is a number between 0 and 1 representing the priority of a memory page to a particular process. These scores are sent to the Data Organizer in the runtime, which determines the composition of data in the DMSH. Pages with higher scores will be elevated to high-performance tiers, while pages with lower scores will be evicted to lower tiers. The Data Organizer will take the maximum of scores if several processes score the same page within a configurable timeframe.

Persistently Integrating Memory with Storage: To improve code reuse and portability, MegaMmap allows arbitrary datasets to be presented as shared vectors, where the vector's name is the URL of the data to load in storage (e.g., a path on a POSIX filesystem). This requires data to be transparently integrated with a particular backend before a job ends. To accomplish this, the `Data Stager` is responsible for serializing, deserializing, and flushing content to the backend. The stager is an extensible component containing integrations with widely-used file formats (e.g., HDF5, Adios2, parquet) and storage services (e.g., PFS, Amazon S3). When an application initially constructs a persistent MegaMmap vector, the key of the vector will indicate the type of staging service to use through a URL format. For example, "hdf5:///path/to/df.h5:mygroup" will ensure that pages which are part of the df.h5 vector will be sent to the HDF5 stager. Periodically and during the termination of the runtime, the stager task will be scheduled to serialize pages in the scache and persist them. During a page fault, if a page is not present in the scache, the stager will be invoked to read and deserialize a subset of data from the persistent backend. These (de)serialization functions allow MegaMmap to transparently load content from storage in the format applications expect to operate on, without requiring specific linkage or semantics of the I/O library.

C. Efficient Memory Coherency

Cache coherence (i.e., two processes accessing the same memory should have equivalent data views) has been widely studied in the context of persistent distributed data structures. There are several workload patterns and use cases where optimizations can be made to reduce communication, synchronization, and data movement. Traditional DSMs focus on coherence to handle small random reads and writes to local and remote DRAM housing temporary content. These algorithms assume that applications accessing data are uncoordinated, in addition to requiring specific page sizes (4KB, 2MB, and 1GB) due to physical restrictions imposed by the CPU. This results in poor I/O access patterns, increased communications, and significant software overheads. MegaMmap supports reads and writes to both temporary and persistent data fragmented among multiple tiers of memory and storage. To offer high performance for a variety of HPC workloads, MegaMmap supports workload-specific page granularity configuration, partial paging, and configurable coherency optimizations depending on access pattern and coordination. This section discusses the coherence algorithms used for the pcache and scache.

Coherency Optimizations with Transactional Memory: Modern DSMs are built to be correct for chaotic access patterns and multi-tenancy, but offer limited optimization for applications with well-defined behavior. Shared data structures in parallel programs are oftentimes divided into distinct phases of read-only, write-only, or append-only, where each process owns a non-overlapping partition of the shared memory for its lifetime. This well-defined nature presents opportunity to eliminate communication overheads and reduce data movements. While coherence in MegaMmap is mainly the responsibility of the application programmer using synchronization points such as barriers and locks (similar to any MPI or PGAS program), transactions further allow programmers to capture various common access patterns between these points (shown in Figure 3) via transactions to optimize communication and data movement costs.

Read / Write Local: In this case, all processes interact with nonoverlapping regions of the DSM. By nature, the caches will be coherent across machines for both operations. During mutable array index operations, the transaction will track the set of pages that were modified. During evictions or flushing operations, the page modifications will be copied to the runtime and processed. Only the parts of the page that were modified will be transferred, ensuring that two processes modifying different parts of the same page do not conflict. An example of this access pattern is matrix transpose and multiplication. These operations are embarrassingly parallel, but are oftentimes intermediate steps in a more complex program that may require a different access pattern in the near future.

Read Only Global: This is common for analysis programs such as machine learning and deep learning. In this case, data is never modified, and no process connected to the shared memory will modify the data. In this case, MegaMmap allows the data to be replicated in both the pcache and scache. This improves data availability and reduces I/O and networking costs by allowing data to be placed near the process requiring the data.

Write and Append Only Global: This is common in many programs, including analysis, inference, and simulation. Gray-Scott [49], for example, is a simulation that models the reaction between chemicals that iteratively modifies concentration values in a 3-D grid. Alternatively, in DBSCAN [50], a k-d tree is created by appending samples to the left and right branches based on a split point. In these cases, MegaMmap leverages properties of the runtime's scheduling system to ensure consistency. During page eviction, MemoryTasks are constructed and scheduled in the runtime. The runtime internally ensures the tasks are ordered and that the data operations defined in the MemoryTask run to completion before the next begins.

Read, Write, and Append Global: This case is less common in HPC, but comes up frequently in Cloud workloads. A key-value store, for example, may incur writes and reads to the same region simultaneously. MegaMmap handles this case the same as the write/append case. When the MegaMmap transaction spans only a single page, strong consistency and atomicity are guaranteed across simultaneous reads and modifications since MemoryTasks will be hashed to the same worker and processed in sequence. Transactions spanning across several pages will have to use synchronization primitives, including locks and barriers, to guarantee correctness – although in many cases this explicit synchronization can be avoided by simply setting a large enough page size.

Collective: In cases where a memory region is going to be accessed by several processes, any of the above access patterns can be marked as collective to allow MegaMmap to improve scalability. A communicator can be passed indicating the set of processes accessing the region. Memory accesses will follow a tree-based pattern to avoid overloading a single node, similar to allgather operations in MPICH [51], [52].

Changing Phases: A MegaMmap vector's access pattern may change throughout the lifetime of a program. For example, in DBSCAN, the k-d tree construction algorithm first divides a root sample by a split point into left and right append-only samples. After all processes have finished splitting their root sample, the left and right samples are then only ever read from. MegaMmap supports phases through synchronization primitives such as barriers. If a region changes from read-only to write-only, all replicas produced during reads will be invalidated.

Reducing Data Movement Through Configurable and Partial Paging: Fixed pages sizes are restrictive, and can result in I/O amplification if the page size is too large or poor access patterns if the page size is too small. Users can choose a custom page size for a particular MegaMmap vector. Vectors do not need to have the same page sizes, but the page size for a particular vector will be equivalent across processes and immutable after the creation of the vector. In addition, since transactions indicate the exact regions of memory that will be accessed, it is possible to know in advance the regions of the page that will be accessed. This allows MegaMmap pages to contain only the fragments of data needed during a page fault.

D. Masking I/O Stalls with Informed Tiered Data Movement Policies

Modern memory-centric HPC and Cloud workloads, including simulation and machine learning, are bottlenecked by the cost of data movement speeds resulting from both the Memory Wall [6] and I/O Bottleneck [7]. Traditional DSMs do not consider the complexity of tiering and require a large, expensive data copy in order to persist data. While DMSH tiering policies have been provided in I/O buffering platforms, they are tailored to workloads where I/O and memory are treated as separate and are typically optimized for bursty producer-only [7], [53] and repetitive read workloads [42] by leveraging statistics and machine learning to guess I/O intention. Unlike existing tiering methodologies, MegaMmap incentivizes applications to specify their access intent through a transactional memory API, offering the opportunity to provide well-informed data placement decisions. While this may appear as an imposition to users, many distributed and out-of-core algorithms are already structured to accomplish this, as they typically operate over large, well-defined subsets of out-of-core data between synchronization points, such as barriers and locks [17].

Data Prefetching: With advance knowledge of the subsets of data being accessed on each iteration, significant performance and resource utilization benefits can be made. In Algorithm 1, MegaMmap's prefetcher is shown. The prefetcher takes as input the maximum amount of memory that the MegaMmap'd region can occupy ($Vec.Max$), the current amount of data in the region ($Vec.Cur$), the page size ($Vec.PageSize$), the minimum score that can be assigned ($MinScore$), and the active transaction object (Tx). The transaction object represents the expected access pattern of the memory region. Whenever a memory access to a MegaMmap vector is made, the transaction tail ($Tx.Tail$) is incremented. $Tx.Head$ represents the current number of memory accesses that have been acknowledged by the prefetcher.

When invoked, the prefetcher determines the set of pages that can be evicted. This is done by determining the set of pages that will not be accessed in the near future. First, all pages that have already been touched (pages between $Tx.Head$ and $Tx.Tail$) will be marked with a score of 0. All pages that are expected to be touched and could be stored in the vector if it were empty (pages between $Tx.Tail$ and $Tx.Tail + Vec.Max/Vec.PageSize$) will be marked with a score of 1. Any page with a score of zero between $Tx.Head$ and $Tx.Tail$ will be evicted. Note that certain transactions (e.g., random) may touch a page several times. The scores between $Tx.Head$ and $Tx.Tail$ may not be 0 if a page is expected to be retouched.

After this, the prefetcher will begin scoring future pages. Pages that can fit in the current amount of space available in the pcache will be marked with a score of 1 (pages between $Tx.Tail$ and $Tx.Tail + (Vec.Max - Vec.Cur)/Vec.PageSize$). All pages that will be accessed soon, but cannot fit in the pcache, will be marked with a score proportional to the minimum amount of time before a page fault could occur. This is calculated by summing the amount of time it theoretically would take to read a page from the scache considering the bandwidth of the tier it is currently located on. When setting the score of a page, the ID of the node setting the score is also propagated to improve the locality of data.

Data Organization: The Data Organizer is responsible for interpreting the scores supplied by the prefetcher. Score updates to the same page will all be hashed to the same worker. Periodically (configurable by the user) the Data Organizer interprets the scores and determines the node and tier where data should be placed. Each tier is assigned a score based on its performance characteristics, where tiers with a score closer to 1 have high I/O performance. The organizer will first attempt to place pages in the fastest tiers if there is available capacity. Pages with lower scores in a tier will be prioritized for eviction to make space for higher-scoring data. If a node sets a high score for a page, the organizer will store the page on that node.

E. Implementation Details

MegaMmap was implemented in 5K lines of C++ code. MegaMmap utilizes Hermes [7], which is a hierarchical buffering platform, to provide basic infrastructure for enacting data movement policies and provide metadata management to locate data in the DMSH. We extended Hermes to support low-latency I/O NVMe devices through the SPDK [54] and traditional libc `mmap` and `memcpy` for upcoming CXL devices. In addition, we augment Hermes to have an extensible data staging layer and support staging for several I/O libraries, including HDF5 1.14 [55] and parquet [56]. Hermes uses Mochi thallium [57] to support low-latency RDMA transfers.

Algorithm 1 Private Cache Prefetcher

```
function PREFETCHER(Vec, Tx, MinScore)
    Evict(Vec,Tx)
    Prefetch(Vec,Tx,MinScore)
    Tx.Head=Tx.Tail
end function
function EVICT(Vec, Tx)
    N=Vec.Max/Vec.PageSize
    for Page in Tx[Tx.Head,Tx.Tail) do
        Page.SetScore(0.0,Vec.NodeId)
    end for
    for Page in Tx[Tx.Tail,Tx.Tail+N) do
        Page.SetScore(1.0,Vec.NodeId)
    end for
    EvictIfZeroScore(Tx[Tx.Head,Tx.Tail))
end function
function PREFETCH(Vec, Tx, MinScore)
    BaseTime=0
    N=(Vec.Max−Vec.Cur)/Vec.PageSize
    for Page in Tx[Tx.Tail,Tx.Tail+N) do
        T=Page.GetTier()
        BaseTime+=Page.GetSize()/T.BW
    end for
    EstTime=BaseTime
    Score=1.0
    while Score>MinScore do
        Page=Tx[Tx.Tail+N]
        T=Page.GetTier()
        EstTime+=Page.GetSize()/T.BW
        Score=EstTime/BaseTime
        Page.SetScore(Score,Vec.NodeId)
        N=N+1
    end while
end function
```

Minimizing Indexing Overhead: To avoid hashtable lookups on every memory access, the page that was last accessed is checked first. This is because many algorithms iterate over a page in its entirety. On average, reading from MegaMmap vectors adds two integer operations and a conditional statement as overhead to a typical memory access (std::vector). We found that this overhead is minor ($\approx 5\%$) compared to a typical memory access in an iterative workload that multiplies a matrix by a scalar.

IV. Evaluations

A. Experimental Methodology

1) Hardware: All tests were conducted on a research cluster, designed to support a hierarchical storage architecture. The cluster consists of storage and a compute rack, each having 32 nodes. The two racks are interconnected by two isolated Ethernet networks (one of 40Gb/s and the other 10Gb/s), with RoCE enabled. Each compute node has a dual Intel(R) Xeon Scalable Silver 4114 with 24 cores and 48 threads, 48 GB RAM, 128GB NVMe PCIe x8 drive, 256GB SSD drive, and 1TB HDD.

2) Software: In several of our experiments, we use the MLlib library of Apache Spark 3.4.1 [19] as a comparison. Spark is largely designed for Cloud environments, where fault tolerance is a primary design goal. To make our comparisons fair, we disable all fault tolerance features to ensure no unnecessary data replication occurs. For MPI-based applications, we use mpich 3.4.3. We do not compare against published DSM solutions such as Grappa, HotPot, and Microsoft A1. This is because Grappa is now deprecated, A1 is not publicly available, and HotPot is designed assuming a particular hardware and software configuration incompatible with our available testbed, including a specific network fabric, drivers, and Linux 3.11.0. We run each experiment 3 times and report the average.

To demonstrate the performance and programmability of MegaMmap, we implemented and verified several real-world machine learning and simulation applications. Each algorithm was verified by comparing their outputs on several datasets to their published counterparts. We measure code volume in terms of LOC using cloc [58], which ignores visual spaces and comments. We also provide an overview of the behavior of these algorithms in terms of their data movement patterns. Each of these algorithms can be parallelized using an arbitrary number of processes p.

	KMeans	RF	DBSCAN	Gray-Scott
MegaMmap	589 C++	403 C++	748 C++	282 C++
Original	1009 Scala/Py	710 Scala/Py	1518 C++	407 C++

Fig. 4: MegaMmap code 45% - 2x smaller. In each case, all I/O partitioning, I/O compatability, and most messaging is removed.

KMeans: KMeans is a foundational and widely-used clustering algorithm that groups data into k spherical clusters. We implement a custom version of KMeans‖ [59], which is the same algorithm used in Apache Spark [60]. KMeans‖ initially divides the dataset evenly among each process and then performs several sequential, read-only iterations over the dataset to determine the initial and final centroids, resulting in approximately $2*log(n)$ iterations for the entire algorithm to converge. After this, data points are assigned to centroids. The assignments are persisted automatically using a file-backed MegaMmap.

Density-based Scanning (DBSCAN): DBSCAN is another important clustering algorithm that identifies arbitrarily-shaped clusters by separating points based on a given distance ϵ. We implement custom versions of μDBSCAN [50]. Initially, DBSCAN constructs a k-d tree, which is a decision tree that splits the dataset by axis. At the first iteration, the dataset is split evenly among the processes. The median and entropy is estimated per-axis using a small, random subsample. The axis with the largest entropy is chosen, and each process divides the dataset into two fractions: left and right of the median. Processes are then partitioned to handle the subsets. This recursion terminates when all points in the subsample have a distance within ϵ of the median or when the sample is smaller than min_pts. Now that each points belongs to a μcluster (set of points in a leaf), the μclusters can be merged in parallel to form the full clusters, which are then persisted.

Random Forest: Random Forests (RFs) are commonly used by data scientists for modeling complex, nonlinear associations in data and providing an indication of feature importance for dimensionality reduction techniques [61]. We implement a custom version of RF. Initially, each process performs out-of-order bagging (oob) on $N/(oob*p)$ randomly-selected samples, where N is the overall dataset size and oob is the number of bags to generate. Each oob iteration measures the

entropy (Gini impurity) of each feature in a chosen feature subset. The per-process oob results are then aggregated to find the feature maximizing entropy. A point is then randomly selected from the dataset and used as the split point. The dataset and processes are then divided into two partitions: left and right. The recursion continues until either the maximum depth (max_depth) of the tree is reached or the entropy difference is below a threshold. The algorithm repeats num_trees times.

Gray-Scott: The Gray-Scott reaction-diffusion model [8] is a system of partial differential equations that captures the dynamic interactions between two chemical species U and V, and holds a pivotal position in the realms of biology, chemistry, and physics. We implement a custom version of Gray-Scott [8] in 300 lines of code. Initially, a grid of volume L^3 is defined and evenly subdivided among each process. Each cell in the grid contains the concentrations of U and V at time step T. At each iteration, the concentrations are updated and exchanged between processes, transferring approximately $24*(L/p)^2$ bytes of data per process. After a certain number of iterations ($plotgap$), the grid of size $O(L^3)$ is checkpointed. The algorithm repeats $steps$ times.

3) Datasets: To evaluate the performance of KMeans and DBSCAN, we use datasets produced by Gadget 4 [62], which is a parallel cosmological N-body and SPH code that simulates cosmic structure formations and calculations relevant for galaxy evolution and galactic dynamics. Several works cluster and model the outputs produced by this simulation to locate halo formations [50], [63] and identify the importance of cosmological features. We include the exact parameters in the AD appendix. The final output data is stored in an HDF5 file containing 3D floating-point particle positions and velocities. To evaluate Random Forest, these values are taken as input and used to predict output clusters. 80% of the original dataset was used for training and 20% for testing using a stratified random sampling that takes 80% of each cluster.

4) Objectives: This evaluation plan aims to demonstrate :

1) Memory coherence of DSMs are not a scalability bottleneck compared to leading HPC+Cloud communication solutions, such as MPI and Spark.
2) Tiering memory can increase the resolution of scientific datasets by eliminating memory constraints, allowing more detail in the final simulation data.
3) Intelligently tiering memory can bring performance benefits to out-of-core algorithms.
4) DRAM consumption can be lowered by offloading memory to tiered storage.

B. Experimental Results

1) Scalability of DSMs for HPC: In this evaluation, we demonstrate the scalability of MegaMmap for in-memory workloads compared to alternative approaches. This exercises the basic communication, coherence, and latency overheads of MegaMmap. To do this, we perform a weak scaling study that compares MegaMmap-based algorithms to the algorithms in the original work. All tests use datasets that allow competing algorithms to maintain all data (including copies) entirely in DRAM. In these evaluations, MegaMmap is configured with no optimizations enabled and only uses memory. For KMeans, we use a dataset of size $2GB$ per node, $k=8$ with a maximum of 4 iterations. We use the same dataset for DBSCAN and use $\epsilon=8$ and $min\_pts=64$. For Random forest, we use a dataset of size $128M$ per node and produce 1 decision tree with $max\_depth=10$. Gray-Scott is configured to produce 16GB of data per node ($L=784$ for 1 node, $L=1920$ for 16 nodes) and does not perform checkpointing ($plotgap=0$). For each algorithm, we run 48 processes (or threads) per node, for a maximum of 768 processes.

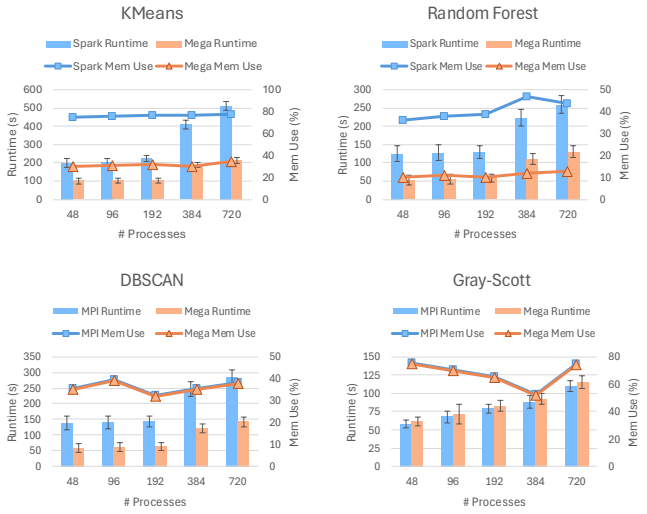

Fig. 5: Weak scaling study of MegaMmap vs alternative application designs, where datasets fit entirely in memory. MegaMmap performs competitively to MPI and as much as 2x faster than Spark.

Overall, from Figure 5, it can be observed that MegaMmap-based algorithms scale and perform competitively to state-of-practice parallel programming approaches. For Spark, performance was as much as 2x faster. This was partially due to implementation details, including its use of the slower TCP protocol and Java Runtime. However, more importantly, Spark creates several copies of the dataset when initially loading data from the backend and during the map/reduce phases. We also found that Spark used 3-4x the amount of DRAM than MegaMmap-based algorithms. For DBSCAN and Gray-Scott, MegaMmap performs similarly to the MPI-based implementation showcasing the scalability of MegaMmap's memory coherence algorithms. Both DBSCAN and MegaMmap stage the datasets entirely in DRAM at the beginning of the program and largely reuse the same portions of the dataset, so there is little benefit to prefetching. Gray-Scott performs synchronous memory-to-memory communication, which MPI excels at and MegaMmap matches. This is because the coherence algorithms in MegaMmap do not spawn flurries of latency-sensitive requests for replica invalidation like traditional DSMs [16]. While DSMs have been traditionally discarded for abstracting distributed memory in HPC due to coherence overheads, it can bee seen that MegaMmap's coherence policies offer competitive performance to modern HPC and Cloud approaches for programming distributed memory systems.

2) Increasing Dataset Resolution: In this evaluation, we demonstrate the impact of tiering memory and storage for increasing dataset resolution, measured by the total dataset size. To do this, we run Gray-Scott to produce grids of varying size. We vary the grid size between $L=2048$ and $L=3456$. We compare the MPI-based implementation for various I/O backends (OrangeFS [64], tiered filesystem Assise [43], and tiered I/O buffering system Hermes [7]) vs MegaMmap in terms of dataset

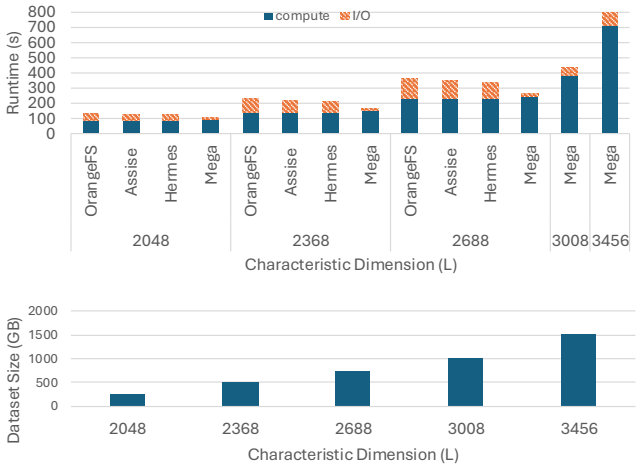

Fig. 6: Increasing the resolution of Gray-Scott through tiering. After $L=2688$, MPI-based Gray-Scott crashes due to memory overutilization. MegaMmap is unbounded, and can run simulations as large as $L=3456$, producing 2x the simulation data of $L=2688$. It's also at least 20% faster than other tiered I/O systems due to effective asynchronous data movement.

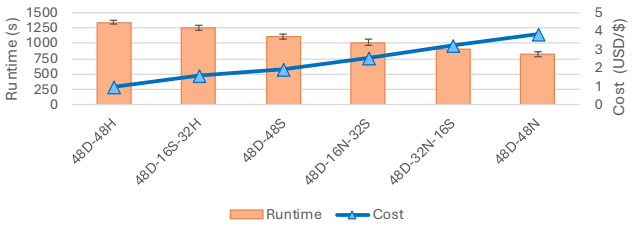

Fig. 7: Tiering study of MegaMmap for 768-process Gray-Scott. D=DRAM, H=HDD, S=SATA SSD, N=NVMe. The number indicates the amount of storage per-node given to the program (e.g., 48D means 48GB DRAM per node). MegaMmap improves performance as much as 1.8x by using NVMe. However, performance is related closely to cost.

size and memory utilization. $L=2048$ produces a 38GB dataset (2.5GB per node), while $L=3456$ produces a 1.5TB dataset (96GB per node). For MegaMmap, the tiers are configured such that there is 48GB of DRAM and 128GB of NVMe per node, which fits the entire dataset at a scale of $L=3456$.

Overall, from Figure 6, it can be seen that MegaMmap performs at least 20% faster than all versions of Gray Scott until $L=2688$. This is because MegaMmap places data during the first compute phase, while all others must wait for this phase to complete. Additionally, MegaMmap's data movement algorithms have advanced knowledge of memory access patterns, leading to improved placement decisions. After $L=2688$, memory is fully utilized. For the MPI-based version, the default behavior of Linux is to terminate programs overutilizing memory, so $L=2688$ is the point at which science can no longer progress. However, through MegaMmap, we are able to conduct experiments as high as $L=3456$, resulting in over 2x the information of $L=2688$ and double the capacity of main memory. This is because MegaMmap leverages the node-local NVMes to enlarge effective memory capacity. Currently, state-of-practice solutions limit science to the boundaries of main memory. MegaMmap eliminates this boundary by automatically tiering memory with storage, allowing memory-constrained scientific problems to become feasible.

3) Performance and Cost Benefits of Persistent Tiered Memory: In this evaluation, we demonstrate the performance impacts of expanding the memory hierarchy to storage for persistent datasets that do not fit trivially in DRAM. To do this, we run Gray-Scott to generate an out-of-core grid. We set L=3456, the maximum size used in the previous experiment, and run 768 processes (16 nodes). The grid size is a total of 1.5TB (96GB per node) and is flushed every step ($plotgap=1$). We run 5 steps, generating a total of 8TB of data (480GB per node). We compare various compositions of the DMSH for handling the placement of this dataset, spanning between NVMe, SSD, and HDDs. We also measure the financial cost of tiering strategies by multiplying utilized storage by $/GB, which we estimate from retailers such as Amazon. We found that HDDs are roughly .02 $/GB, SATA SSDs are .04$/GB, and NVMes are .08$/GB.

Overall, from Figure 7, it can be observed that deepening the memory hierarchy to include storage tiers can greatly improve overall application performance. In this case, Gray-Scott performs a write-intensive workload, where a large grid is modified and persisted on every step. MegaMmap overlaps computation with data movement leveraging the asynchronous behavior of the write/append only cache coherence policy when modifying the grid and the asynchronous data staging engine when persisting the data during checkpoints. Individual data movements are accelerated by allowing data to reside on faster storage tiers. This data is aggressively demoted to lower tiers by the data organizer to make room for future incoming data.

At the baseline, MegaMmap leverages slow, high-volume storage (HDDs) to handle memory overflow, incurring significant I/O penalties and experiencing the slowest overall performance. As we add high-performance storage, this trade-off changes. With 16GB of NVMe and 32GB of SSD per-node, performance improves 1.5x over the baseline as the dataset doesn't immediately touch the HDDs, which are 6-10x slower than the SSD and NVMe. When increasing the amount of NVMe to 32GB, performance improves an additional 20%. When using only DRAM and NVMe, performance is overall 1.8x faster than the baseline. Though, in terms of cost, 48D-48S has 1.5x improvement over 48D-48H at half the financial cost of 48D-48N. By expanding memory to high-performance storage, MegaMmap allows applications to transparently stage data in accelerated storage and avoid costly synchronous disk seeks, providing substantial performance benefits when data doesn't fit entirely in DRAM. Even cheaper hardware can result in significant performance benefits compared to HDD-only solutions.

4) Lowering DRAM Consumption: In this evaluation, we measure the performance impacts of reducing DRAM consumption using optimal configurations of MegaMmap. To do this, we generate datasets for each application and then vary the maximum DRAM capacity. The datasets are generated using Gadget and are 1TB in size (32GB per node). We run KMeans and DBSCAN to first analyze the datasets produced by Gadget. We use $k=8$ with a $max\_iter=4$ for KMeans. For DBSCAN, we use an $\epsilon=8$ and $min\_pts=64$. The cluster assignments are stored in a binary file. RF analyzes this data and produces 1 decision tree with a $max\_depth=10$. For Gray-Scott, we set $L=3128$, also producing 1TB. Each application is executed with 1536 total

Fig. 8: Memory scaling study of MegaMmap. Each program has 1536 processes over 32 nodes. Through intelligent tiering, DRAM can be lowered as much as 2.6x while maintaining competitive (within 10%) performance of full DRAM capacity.

processes (32 nodes). We vary the amount of DRAM being used for each program type. All overflowing data will fit in NVMe.

Overall, from Figure 8, it can be observed that each of the algorithms execute with nearly the same performance even with just half the main memory, demonstrating that MegaMmap can effectively overlap data movements with computation. KMeans can use $2.6\times$ less DRAM while DBSCAN and RF can use $2\times$ less DRAM with minimal performance overhead. This result is largely due to the effectiveness of asynchronous prefetching and eviction. KMeans, RF, and DBSCAN iterate over portions of the dataset highly predictably, where KMeans is mostly sequential while RF and DBSCAN are more pseudo-random. This allows MegaMmap to evict pages that have been touched during data-intensive computations and predict the pages that will be accessed next. In the case of Gray-Scott, which can also use $1.6\times$ less DRAM, this result is largely due to asynchronous writes and flushing. Data is initially placed in the fastest tier with available capacity, and then aggressively flushed down to lower tiers. After a certain point, each of the programs incur significant overheads due to frequent synchronous page faults and I/O stalls caused by frequent spills to NVMe, resulting in performance degradation of as much as $2.5\times$. Overall, this evaluation demonstrates that achieving high performance for memory-centric, data-intensive problems can be accomplished without increasing DRAM capacity through intelligent use of storage tiering and asynchronous behavior.

V. Discussion & Limitations

Node Failure: Currently, MegaMmap assumes that the nodes are reliable and that the application would fail anyway if a node were to go down. If a node were to fail, the entire DSM would be corrupted. However, the MegaMmap runtime could be extended to support reliabilty and fault tolerance by implementing replication [65].

Security: Utilizing persistent storage to store temporary information can result in security concerns. If the data being analyzed is classified, the DSM must buffer data with the same level of access as the original content. This protection can be accomplished by changing the access control permissions of data within the MegaMmap runtime. [66].

Memory Corruption: Applications may encounter a situation where hardware stores data incorrectly. Bit flips in DRAM are not uncommon [67], and there are algorithms such as error correcting codes [67] that MegaMmap could implement to ensure that data remains correct.

Accelerator APIs: As the variety of accelerators such as FPGAs and GPUs continue to rise, software support must be provided. We designed MegaMmap with such abstraction that it could be easily extended to offer coherent access to distributed memory and storage from within accelerators.

Multi-Tenancy: MegaMmap is designed to run within an HPC job – where a single phase of a workflow runs. For this reason, we do not explore the effects of multi-tenancy in this work, as only one application accesses the DSM at a time. In the future, we intend to expand the shared cache design to consider multi-tenant DSMs and contention mediation.

Legacy Applications and Language Support: MegaMmap is a library (similar to MPI and PGAS) designed to leverage language features, specifically operator overloading, to accomplish a friendly DSM interface. Any language supporting this (e.g., Python, Julia, C#, Rust) can implement a similar API. C programs can typically switch to a C++ compiler to use MegaMmap. To adapt an existing program to MegaMmap, only the data structures that are too large to fit in memory need to be converted. Accesses to these vectors would also have to be updated to use transactions. This can vary in time and complexity based on the volume and complexity of the original code. As future work, we intend to explore compiler extensions to allow MegaMmap to be embedded automatically into applications, which would better support legacy programs.

VI. Conclusion

In this work, we address the issues facing distributed, memory-centric out-of-core algorithms through the design and implementation of MegaMmap, a software DSM that enlarges effective memory capacity through intelligent tiered DRAM and storage management. We demonstrate that the complexity of developing out-of-core algorithms can be reduced by enabling massive persistent datasets to be presented as memory objects while transparently managing data placement and space utilization. We showcase how expanding memory tiering to high-performance storage can be used to both improve performance and dramatically reduce the need for DRAM in parallel workloads. We demonstrate how leveraging a transactional memory API can be used to reap substantial benefits in tiering performance. Evaluations showcase that developing algorithms with MegaMmap reduces peak memory utilization by as much as $2.6\times$ compared to leading solutions while accomplishing similar or better performance.

VII. Acknowledgement

The material presented is based upon work supported by the National Science Foundation (NSF), Office of Advanced Cyberinfrastructure, under Grants CSSI-2104013 and Core-2313154. Additionally, this work is partially supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research, under Contract and DE-SC0024593. We would like to thank the Chameleon testbed, supported by the NSF, for providing an environment for development and debugging.

REFERENCES

[1] M. M. A. Patwary, S. Byna, N. R. Satish, N. Sundaram, Z. Lukić, V. Roytershteyn, M. J. Anderson, Y. Yao, and P. Dubey, "Bd-cats: big data clustering at trillion particle scale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.

[2] M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, perspectives, and prospects," *Science*, vol. 349, no. 6245, pp. 255–260, 2015.

[3] C. Adams, "Neutrino and cosmic tagging with unet," 2015.

[4] W. F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck, A. Huebl, M. Kim, J. Kress, T. Kurc, Q. Liu, J. Logan, K. Mehta, G. Ostrouchov, M. Parashar, F. Poeschel, D. Pugmire, E. Suchyta, K. Takahashi, N. Thompson, S. Tsutsumi, L. Wan, M. Wolf, K. Wu, and S. Klasky, "Adios 2: The adaptable input output system. a framework for high-performance data management," *SoftwareX*, vol. 12, p. 100561, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711019302560

[5] J. Lüttgau, S. Snyder, P. Carns, J. M. Wozniak, J. Kunkel, and T. Ludwig, "Toward understanding i/o behavior in hpc workflows," in *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)*, 2018, pp. 64–75.

[6] Y.-H. Liu and X.-H. Sun, "Lpm: concurrency-driven layered performance matching," in *2015 44th International Conference on Parallel Processing*. IEEE, 2015, pp. 879–888.

[7] A. Kougkas, H. Devarajan, and X.-H. Sun, "Hermes: a heterogeneous-aware multi-tiered distributed i/o buffering system," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, 2018, pp. 219–230.

[8] J. E. Pearson, "Complex patterns in a simple system," *Science*, vol. 261, no. 5118, pp. 189–192, 1993. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.261.5118.189

[9] A. Pupykina and G. Agosta, "Survey of memory management techniques for hpc and cloud computing," *IEEE Access*, vol. 7, pp. 167 351–167 373, 2019.

[10] Q. He, S. Zhou, B. Kobler, D. Duffy, and T. McGlynn, "Case study for running hpc applications in public clouds," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010, pp. 395–401.

[11] K. H. Lee *et al.*, "A strategic analysis of the dram industry after the year 2000," Ph.D. dissertation, Massachusetts Institute of Technology, 2013.

[12] S. Mittal, "A survey of architectural techniques for dram power management," *International Journal of High Performance Systems Architecture*, vol. 4, no. 2, pp. 110–119, 2012.

[13] A. Shanbhag, N. Tatbul, D. Cohen, and S. Madden, "Large-scale in-memory analytics on intel® optane™ dc persistent memory," in *Proceedings of the 16th International Workshop on Data Management on New Hardware*, ser. DaMoN '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3399666.3399933

[14] C. Buragohain, K. M. Risvik, P. Brett, M. Castro, W. Cho, J. Cowhig, N. Gloy, K. Kalyanaraman, R. Khanna, J. Pao *et al.*, "A1: A distributed in-memory graph database," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 329–344.

[15] J. Lee, H. Kim, and R. Vuduc, "When prefetching works, when it doesn't, and why," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 1, mar 2012. [Online]. Available: https://doi.org/10.1145/2133382.2133384

[16] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "Treadmarks: Shared memory computing on networks of workstations," *Computer*, vol. 29, no. 2, pp. 18–28, 1996.

[17] Y. Shan, S.-Y. Tsai, and Y. Zhang, "Distributed shared persistent memory," in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017, pp. 323–337.

[18] B. Behzad, S. Byna, Prabhat, and M. Snir, "Optimizing i/o performance of hpc applications with autotuning," *ACM Trans. Parallel Comput.*, vol. 5, no. 4, mar 2019. [Online]. Available: https://doi.org/10.1145/3309205

[19] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. USA: USENIX Association, 2010, p. 10.

[20] Q. Wang, Y. Lu, E. Xu, J. Li, Y. Chen, and J. Shu, "Concordia: Distributed shared memory with in-network cache coherence." in *FAST*, 2021, pp. 277–292.

[21] N. Y. Song, Y. Son, H. Han, and H. Y. Yeom, "Efficient memory-mapped i/o on fast storage device," *ACM Transactions on Storage (TOS)*, vol. 12, no. 4, pp. 1–27, 2016.

[22] I. Malliotakis, A. Papagiannis, M. Marazakis, and A. Bilas, "Hugemap: Optimizing memory-mapped i/o with huge pages for fast storage," in *Euro-Par 2020: Parallel Processing Workshops: Euro-Par 2020 International Workshops, Warsaw, Poland, August 24–25, 2020, Revised Selected Papers 26*. Springer, 2021, pp. 344–355.

[23] J. Won, S. Yun, A. Jemin, J.-C. Kim, and K. Kang, "Spidermine: Low overhead user-level prefetching," in *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, ser. SAC '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1332–1341. [Online]. Available: https://doi.org/10.1145/3555776.3577754

[24] S. Rivas-Gomez, A. Fanfarillo, S. Valat, C. Laferriere, P. Couvee, S. Narasimhamurthy, and S. Markidis, "ummap-io: User-level memory-mapped i/o for hpc," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2019, pp. 363–372.

[25] W. Liu and A.-I. A. Wang, "Lfuzz: Exploiting locality-enabled techniques for file-system fuzzing," in *European Symposium on Research in Computer Security*. Springer, 2023, pp. 507–525.

[26] S. Bitchebe and A. Tchana, "Out of hypervisor (ooh): When nested virtualization becomes practical," 2022.

[27] A. Papagiannis, M. Marazakis, and A. Bilas, "Memory-mapped i/o on steroids," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 277–293.

[28] A. Papagiannis, G. Xanthakis, G. Saloustros, M. Marazakis, and A. Bilas, "Optimizing memory-mapped i/o for fast storage devices," in *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, 2020, pp. 813–827.

[29] G. Feng, H. Cao, X. Zhu, B. Yu, Y. Wang, Z. Ma, S. Chen, and W. Chen, "Tricache: A user-transparent block cache enabling high-performance out-of-core processing with in-memory programs," *ACM Transactions on Storage*, vol. 19, no. 2, pp. 1–30, 2023.

[30] A. Abulila, V. S. Mailthody, Z. Qureshi, J. Huang, N. S. Kim, J. Xiong, and W.-m. Hwu, "Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 971–985.

[31] H. Al Maruf and M. Chowdhury, "Effectively prefetching remote memory with leap," in *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, 2020, pp. 843–857.

[32] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, "Latency-tolerant software distributed shared memory," in *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, 2015, pp. 291–305.

[33] G. Almasi, "Pgas (partitioned global address space) languages." 2011.

[34] A. M. Aji, L. S. Panwar, F. Ji, K. Murthy, M. Chabbi, P. Balaji, K. R. Bisset, J. Dinan, W.-c. Feng, J. Mellor-Crummey *et al.*, "Mpi-acc: accelerator-aware mpi for scientific applications," *IEEE transactions on parallel and distributed systems*, vol. 27, no. 5, pp. 1401–1414, 2015.

[35] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing openshmem: Shmem for the pgas community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, 2010, pp. 1–3.

[36] L. V. Kale and S. Krishnan, "Charm++ a portable concurrent object oriented system based on c++," in *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, 1993, pp. 91–108.

[37] J. Bachan, S. B. Baden, S. Hofmeyr, M. Jacquelin, A. Kamil, D. Bonachea, P. H. Hargrove, and H. Ahmed, "Upc++: A high-performance communication framework for asynchronous computation," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 963–973.

[38] X. Lu, R. Wang, and X.-H. Sun, "Care: A concurrency-aware enhanced lightweight cache management framework," in *The 29th IEEE International Symposium on High-Performance Computer Architecture (HPCA-29)*. IEEE Computer Society, 2023.

[39] M. Hoseinzadeh, "A survey on tiering and caching in high-performance storage systems," *arXiv preprint arXiv:1904.11560*, 2019.

[40] Y. T. Jin, S. Ahn, and S. Lee, "Performance analysis of nvme ssd-based all-flash array systems," in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2018, pp. 12–21.

[41] H. Devarajan, A. Kougkas, L. Logan, and X.-H. Sun, "Hcompress: Hierarchical data compression for multi-tiered storage environments," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 557–566.

[42] H. Devarajan, A. Kougkas, and X.-H. Sun, "Hfetch: Hierarchical data prefetching for scientific workflows in multi-tiered storage environments," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 62–72.

[43] T. E. Anderson, M. Canini, J. Kim, D. Kostić, Y. Kwon, S. Peter, W. Reda, H. N. Schuh, and E. Witchel, "Assise: Performance and availability via client-local {NVM} in a distributed file system," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 1011–1027.

[44] J. Kim, Y. J. Soh, J. Izraelevitz, J. Zhao, and S. Swanson, "Subzero: Zero-copy io for persistent main memory file systems," in *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2020, pp. 1–8.

[45] G. Quintana-Ortí, F. D. Igual, M. Marqués, E. S. Quintana-Ortí, and R. A. Van de Geijn, "A runtime system for programming out-of-core matrix algorithms-by-tiles on multithreaded architectures," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 4, pp. 1–25, 2012.

[46] L. Logan, J. C. Garcia, J. Lofstead, X.-H. Sun, and A. Kougkas, "Labstor: A modular and extensible platform for developing high-performance, customized i/o stacks in userspace," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2022, pp. 1–15.

[47] J. Lüttgau, M. Kuhn, K. Duwe, Y. Alforov, E. Betke, J. Kunkel, and T. Ludwig, "Survey of storage systems for high-performance computing," *Supercomputing Frontiers and Innovations*, vol. 5, no. 1, 2018.

[48] T. Vogelsang, "Understanding the energy consumption of dynamic random access memories," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010, pp. 363–374.

[49] Gray-scott simulation code. [Online]. Available: https://github.com/pnorbert/adiosvm/tree/master/Tutorial/gray-scott

[50] A. Sarma, P. Goyal, S. Kumari, A. Wani, J. S. Challa, S. Islam, and N. Goyal, "μdbscan: an exact scalable dbscan algorithm for big data exploiting spatial locality," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2019, pp. 1–11.

[51] R. Thakur and W. D. Gropp, "Improving the performance of collective operations in mpich," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2003, pp. 257–267.

[52] Z. Cai, Z. Liu, S. Maleki, M. Musuvathi, T. Mytkowicz, J. Nelson, and O. Saarikivi, "Synthesizing optimal collective algorithms," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 62–75.

[53] N. Rajesh, H. Devarajan, J. C. Garcia, K. Bateman, L. Logan, J. Ye, A. Kougkas, and X.-H. Sun, "Apollo: An ml-assisted real-time storage resource observer," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '21. New York, NY, USA: Association for Computing Machinery, 2020, p. 147–159. [Online]. Available: https://doi.org/10.1145/3431379.3460640

[54] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul, "Spdk: A development kit to build high performance storage applications," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2017, pp. 154–161.

[55] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An overview of the hdf5 technology suite and its applications," in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, ser. AD '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 36–47. [Online]. Available: https://doi.org/10.1145/1966895.1966900

[56] D. Vohra, *Apache Parquet*. Berkeley, CA: Apress, 2016, pp. 325–335. [Online]. Available: https://doi.org/10.1007/978-1-4842-2199-0_8

[57] R. B. Ross, G. Amvrosiadis, P. Carns, C. D. Cranor, M. Dorier, K. Harms, G. Ganger, G. Gibson, S. K. Gutierrez, R. Latham, B. Robey, D. Robinson, B. Settlemyer, G. Shipman, S. Snyder, J. Soumagne, and Q. Zheng, "Mochi: Composing data services for high-performance computing environments," *Journal of Computer Science and Technology*, vol. 35, no. 1, pp. 121–144, 2020. [Online]. Available: https://doi.org/10.1007/s11390-020-9802-0

[58] A. Danial, "cloc: v1.92," Dec. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5760077

[59] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii, "Scalable k-means++," *arXiv preprint arXiv:1203.6402*, 2012.

[60] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning in apache spark," *The journal of machine learning research*, vol. 17, no. 1, pp. 1235–1241, 2016.

[61] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001. [Online]. Available: https://doi.org/10.1023/A:1010933404324

[62] V. Springel, R. Pakmor, O. Zier, and M. Reinecke, "Simulating cosmic structure formation with the gadget-4 code," *Monthly Notices of the Royal Astronomical Society*, vol. 506, no. 2, pp. 2871–2949, 2021.

[63] L. Lucie-Smith, H. V. Peiris, A. Pontzen, and M. Lochner, "Machine learning cosmological structure formation," *Monthly Notices of the Royal Astronomical Society*, vol. 479, no. 3, pp. 3405–3414, 06 2018. [Online]. Available: https://doi.org/10.1093/mnras/sty1719

[64] M. M. D. Bonnie, B. Ligon, M. Marshall, W. Ligon, N. Mills, E. Q. S. Sampson, S. Yang, and B. Wilson, "Orangefs: advancing pvfs," in *USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[65] H. Devarajan, A. Kougkas, and X.-H. Sun, "Hreplica: a dynamic data replication engine with adaptive compression for multi-tiered storage," in *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 2020, pp. 256–265.

[66] R. Rosen, "Namespaces and cgroups, the basis of linux containers," *Seville, Spain, Feb*, 2016.

[67] D. Fiala, F. Mueller, and K. B. Ferreira, "Flipsphere: A software-based dram error detection and correction library for hpc," in *2016 IEEE/ACM 20th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. IEEE, 2016, pp. 19–28.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

The contributions of this work are as follows:

- C_1 A durable, persistent, and intuitive **Distributed Shared Memory (DSM) system**, which significantly reduces out-of-core development complexity by allowing applications to present massive datasets as memory objects.
- C_2 A user-driven **transactional memory access API**, which leads to improved decision-making in cache coherence and data organization policies by propagating memory access intent.
- C_3 A comprehensive set of **intent-aware memory coherence** policies, which improves the latency and bandwidth of memory accesses based on workload characteristics.
- C_4 A wide variety of **tiered data organization** policies, which minimize I/O stall times by leveraging heterogeneous storage hardware and advance knowledge of access pattern intent.

B. Computational Artifacts

There is only one artifact. All evaluation figures were produced (and can be reproduced) from this artifact.

- A_1 https://github.com/grc-iit/mega_mmap

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1-C_4	Figures 4 - 7

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

This artifact contains the MegaMmap runtime, library, and developed applications. LOC measurements from this repo's benchmark directory were used for proving C_1. The applications in the benchmark directory are used to evaluate components of MegaMmap in challenges C_1 - C_4. There are 4 applications: KMeans, DBSCAN, Gray-Scott, and Random Forest.

Expected Results

- C_1 Each application in the benchmark directory should be between 300 - 600 LOC, demonstrating intuitiveness. There are 4 applications: KMeans, DBSCAN, Gray-Scott, and Random Forest.
- $C_{2,3}$ When running the applications on workloads that fit entirely in main memory, the only overhead in MegaMmap is caused by cache coherence informed by the transaction API. It should be seen that MegaMmap-based algorithms perform at least as well as MPI-based versions and better than Spark, demonstrating that transactions can be used to make cache coherence overheads minimal compared to state-of-practice approaches.
- $C_{2,4}$ When running the applications on workloads where the dataset does not fit entirely in memory, MegaMmap's transaction-informed tiering policies are showcased. It should be seen that the applications do not require full memory utilization to accomplish peak performance due to intelligent data movement. It should also be seen that performance improves when data can fit entirely in high-performance storage and never spills to disk. This shows that transactions can be used to effectively overlap I/O stall times with computation, minimizing data movement overheads.

Expected Reproduction Time (in Minutes)

Artifact Setup: The artifact should compile and install within 30 minutes – it has several dependencies. We provide scripts in the form of YAML files that define the exact parameters of experiments. These are included under the test/iter-pipelines directory.

Artifact Evaluation:

1) MegaMmap-based KMeans with a dataset size of 4GB per node takes roughly 100 seconds. For 32GB it took roughly 8 minutes.
2) MegaMmap-based DBSCAN with a dataset size of 4GB took roughly 130 seconds. For 32GB it took roughly 10 minutes.
3) MegaMmap-based Random Forest with a dataset size of 4GB per-node takes roughly 150 seconds. For 32GB it took roughly 13 minutes.
4) MegaMmap-based Gray-Scott that generated 16GB per node took roughly 2.5 minutes. For 32GB it took roughly 6 minutes.

The overall execution of our evaluation was roughly 16 hours, since applications were executed numerous times and with many different configurations.

Artifact Analysis: The analysis is a matter of seconds, requiring only the summarization of small CSV files containing a few hundred entries of runtime, CPU utilization, and memory utilization statistics.

Artifact Setup (incl. Inputs)

Hardware: All tests were conducted on a research cluster, designed to support a hierarchical storage architecture. The cluster consists of storage and a compute rack, each having 32 nodes. The two racks are interconnected by two isolated Ethernet networks (one of 40Gb/s and the other 10Gb/s), with RoCE enabled. Each compute node has a dual Intel(R) Xeon Scalable Silver 4114 with 24 cores and 48 threads, 48 GB

RAM, 128GB NVMe PCIe x8 drive, 256GB SSD drive, and 1TB HDD.

Software:

1) **MegaMmap**: Currently the master branch. Url: https://github.com/grc-iit/mega_mmap DOI: https://doi.org/10.5281/zenodo.13329688
2) **Hermes**: We use Hermes 1.2.1. Url: https://github.com/HDFGroup/hermes Tag: https://github.com/HDFGroup/hermes/releases/tag/v1.2.1
3) **Jarvis-CD**: Currently the master branch. Url: https://github.com/grc-iit/jarvis-cd Tag: https://github.com/grc-iit/jarvis-cd/releases/tag/v1.0.0
4) **Apache Spark**: We use version 3.4.1. Url: https://archive.apache.org/dist/spark/spark-3.4.1/spark-3.4.1-bin-without-hadoop.tgz
5) **MPICH**: We use version 3.4.3. Url: https://www.mpich.org/static/downloads/3.4.3/mpich-3.4.3.tar.gz
6) **Gadget4**: We use commit a3270b2ce91067af4045ac8a68398a12ede70b85. Url: https://gitlab.mpcdf.mpg.de/vrs/gadget4

Datasets / Inputs: Datasets can be generated using the Gadget4 simulator or our internal kmeans dataset generator (included in github), which outputs data in a similar format to Gadget and can be used to accelerate reproducability. While Gadget4 was used for the datasets, its setup is far more complex. The use of Gadget4 was not for performance reasons, but to demonstrate that MegaMmap can be used in realistic workflows and datasets.

We used two main datasets during our evaluation:

1) 64GB Gadget4 simulation (for Evaluation 1)
2) 1TB Gadget4 simulation (for Evaluation 4)

Installation and Deployment: MegaMmap is implemented in C++ and requires a C++17-compliant compiler. The minimum version tested is GCC 9.4.1. We use spack to automate the installation of MegaMmap and its dependencies, including MPICH.

Artifact Execution

We conducted 4 different evaluations of MegaMmap. To help automate and reproduce experiments, we use a deployment tool named Jarvis-CD. Through Jarvis, we provide deployment scripts for Spark, MegaMmap, Hermes, and the Jarvis resource monitoring tool pymonitor.

Experiments are conducted similarly to one another. The experiments are represented as Jarvis Grid Search YAML files (e.g., https://github.com/grc-iit/mega_mmap/blob/master/test/unit/iter-pipelines/mm_kmeans_spark.yaml). It defines the order to deploy applications and the variables to tune. Each experiment has the following general workflow:

1) The MegaMmap runtime is deployed on each node an application is expected to run on.
2) The pymonitor tool is then deployed on each node. Pymonitor produces a time series CSV file containing CPU, network, and storage utilization statistics in the background.
3) The application (KMeans, DBSCAN, Random Forest, or Gray-Scott) is launched in parallel and executed to completion. The runtime of the application is stored internally in Jarvis.
4) Steps 1 - 4 repeat until all possible configurations defined in the grid search have been tested.
5) Jarvis produces a single CSV file that, for each tested configuration, contains the aggregated resource utilization statistics and application runtime.

We describe the evaluations below. Each evaluation was repeated 3 times.

Evaluation 1: Scalability of DSMs for HPC: In this evaluation, we demonstrate the scalability of MegaMmap for in-memory workloads compared to alternative approaches. This exercises the basic communication, coherence, and latency overheads of MegaMmap. To do this, we perform a weak scaling study that compares MegaMmap-based algorithms to the algorithms in the original work. All tests use datasets that allow competing algorithms to maintain all data (including copies) entirely in DRAM. In these evaluations, MegaMmap is configured with no optimizations enabled and only uses memory. For KMeans, we use a dataset of size $2GB$ per node, $k = 8$ with a maximum of 4 iterations. We use the same dataset for DBSCAN and use $\epsilon = 8$ and $min\_pts = 64$. For Random forest, we use a dataset of size $128M$ per node and produce 1 decision tree with $max\_depth = 10$. Gray-Scott is configured to produce 16GB of data per node ($L = 784$ for 1 node, $L = 1920$ for 16 nodes) and does not perform checkpointing ($plotgap = 0$). For each algorithm, we run 48 processes (or threads) per node, for a maximum of 768 processes.

Evaluation 2: Increasing Dataset Resolution: In this evaluation, we demonstrate the impact of tiering memory and storage for increasing dataset resolution, measured by the total dataset size. To do this, we run Gray-Scott to produce grids of varying size. We vary the grid size between $L = 2048$ and $L = 3456$. We compare the MPI-based implementation vs the MegaMmap implementation in terms of dataset size and memory utilization. $L = 2048$ produces a 38GB dataset (2.5GB per node), while $L = 3456$ produces a 1.5TB dataset (96GB per node). For MegaMmap, the tiers are configured such that there is 48GB of DRAM and 128GB of NVMe per node, which fits the entire dataset at a scale of $L = 3456$.

Evaluation 3: Performance Benefits of Persistent Tiered Memory: In this evaluation, we demonstrate the performance impacts of expanding the memory hierarchy to storage for persistent datasets that do not fit trivially in DRAM. To do this, we run Gray-Scott to generate an out-of-core grid. We set L=3456, the maximum size used in the previous experiment, and run 768 processes (16 nodes). The grid size is a total of 1.5TB (96GB per node) and is flushed every step ($plotgap = 1$). We run 5 steps, generating a total of 8TB of data (480GB per node). We compare various compositions of the Deep Memory and Storage Hierarchy (DMSH) for handling the placement of this dataset, spanning between NVMe, SSD, and HDDs. These compositions include:

1) 48GB DRAM - 48GBHDD

2) 48GB-DRAM / 16GB NVMe / 32GB SATA SSD
3) 48GB-DRAM / 32GB NVMe / 16GB SATA SSD
4) 48GB-DRAM / 48GB NVMe

Evaluation 4: Lowering DRAM Consumption: In this evaluation, we measure the performance impacts of reducing DRAM consumption using optimal configurations of MegaMmap. To do this, we generate datasets for each application and then vary the maximum DRAM capacity. The datasets are generated using Gadget and are 1TB in size (32GB per node). We run KMeans and DBSCAN to first analyze the datasets produced by Gadget. We use $k = 8$ with a $max\_iter = 4$ for KMeans. For DBSCAN, we use an $\epsilon = 8$ and $min\_pts = 64$. The cluster assignments are stored in a binary file. RF analyzes this data and produces 1 decision tree with a $max\_depth = 10$. For Gray-Scott, we set $L = 3128$, also producing 1TB. Each application is executed with 1536 total processes (32 nodes). We vary the amount of DRAM being used for each program type between 4GB and 32GB. All overflowing data will fit in NVMe.

Artifact Analysis (incl. Outputs)

All outputs used in experiments are stored as CSVs containing the configuration of the MegaMmap system (e.g., storage capacity limits, number of processes, number of nodes, etc.), the runtime of the program, average/peak memory/CPU utilization, and standard deviations of runtime and resource utilization. Without any additional processing, these figures can be used to produce the figures in the evaluation.

Artifact Evaluation (AE)

A. Computational Artifact A_1

Artifact Setup (incl. Inputs)

1. Download and install Jarvis-CD, Spack, Hermes, and Spark using deps.sh. This script assumes that you have python$\geq$ 3.6 and environment modules (e.g., module load):

```
bash deps.sh
```

Note that this will install Hermes with the default parameters, which only works with the slower network providers (e.g., TCP/sockets). It is difficult to automate the detection of network drivers for RDMA. In deps.sh, you will likely need to change the `spack install hermes` command to include more specific configurations of libfabric for your machine. For example, `spack install hermes^libfabric+verbs+mlx` will install hermes with RDMA for mellanox interconnects.

2. Define where Jarvis stores produced configuration files. There are three paths that need to be defined:

- CONFIG_DIR: A directory where jarvis metadata for pkgs and pipelines are stored. This directory can be anywhere that the current user can access. E.g., ${HOME}/jarvis-conf.
- PRIVATE_DIR: A directory which is common across all machines, but stores data locally to the machine. Some jarvis pkgs require certain data to be stored per-machine. OrangeFS is an example. E.g., /tmp/jarvis-priv
- SHARED_DIR: A directory which is common across all machines, where each machine has the same view of data in the directory. Most jarvis pkgs require this, but on machines without a global filesystem (e.g., Chameleon Cloud), this parameter can be set later. In a typical HPC site, this would be somewhere in your home directory. E.g., ${HOME}/jarvis-shared

To define the paths, run the following:

```
jarvis init \
[CONFIG_DIR] \
[PRIVATE_DIR] \
[SHARED_DIR (optional)]
```

3. Define a resource graph for your machine. The resource graph defines the network and storage configuration, including paths and such. It is a YAML file with the following format:

```
fs:
- avail: 500g
  dev_type: ssd
  device: null
  fs_type: null
  host: null
  mount: ${HOME}/jarvis_data
  rota: null
  shared: false
  tran: ''
```

```
net:
- domain: lo
  fabric: 127.0.0.1/32
  host: null
  provider: sockets
  shared: false
  speed: 1G
```

There should be an entry for each storage device and network you plan to use. You can discover networks using the fi_info tool provided by libfabric, which is a dependency of megammap.

4. Register MegaMmap in Jarvis

```
cd /path/to/mega_mmap
jarvis repo add test/jarvis_mega_mmap
```

5. Load and cache the megammap environment in Jarvis. The following commands will store a snapshot of all relevant environment variables needed to execute workflow stages.

```
module load spark
module load arrow
spack load mega_mmap
jarvis env build mega_mmap
```

Artifact Execution

The experiments are defined using Jarvis workflow YAML files (shown in test/unit/iter-pipelines). These YAML files define the execution steps and environment setup needed to run the experiments. To run any particular workflow file, you can do:

```
jarvis ppl run yaml /path/to/workflow.yaml
```

Evaluation 1: Scalability of DSMs for HPC: This evaluation spanned 8 applications. 2x KMeans, 2x DBSCAN, 2x Random Forest, and 2x Gray Scott. These files are located under the path: test/unit/iter-pipelines/weak_scaling. These evaluations perform a weak scaling study, where the size of a dataset and the number of processes are varied simultaneously. To execute the evaluations, the Jarvis-CD YAML files. The evaluator may need to tweak these YAML files to specific machine characteristics – particularly memory capacity and scale. They are organized as follows:

1) mm_kmeans_mega.yaml: This will produce a synthetic dataset that fits in memory, launch the Hermes runtime, a custom monitoring program for tracking memory utilization named **pymonitor**, and then launch MegaMmap's KMeans implementation. The number of processes and dataset size are varied automatically as configured in this file.
2) mm_kmeans_spark.yaml: This will produce a synthetic dataset that fits in memory, launch Spark cluster, launch pymonitor, and then launch Spark's KMeans implementation. The number of processes and dataset size are varied automatically as configured in this file.
3) mm_rf_mega.yaml: This will produce a synthetic dataset that fits in memory, launch the Hermes runtime, launch pymonitor, and then launch MegaMmap's Random Forest implementation. The number of processes and dataset size are varied automatically as configured in this file.
4) mm_rf_spark.yaml: This will produce a synthetic dataset that fits in memory, launch Spark cluster, launch pymonitor, and then launch Spark's Random Forest implementation. The number of processes and dataset size are varied automatically as configured in this file.
5) mm_dbscan_mega.yaml: This will produce a synthetic dataset that fits in memory, launch the Hermes runtime, launch pymonitor, and then launch MegaMmap's DBSCAN implementation. The number of processes and dataset size are varied automatically as configured in this file.
6) mm_dbscan_mpi.yaml: This will produce a synthetic dataset that fits in memory, launch pymonitor, and then launch the MPI-based DBSCAN implementation. The number of processes and dataset size are varied automatically as configured in this file.
7) mm_gray_scott_mega.yaml: This will launch the Hermes runtime, launch pymonitor, and then launch MegaMmap's Gray Scott implementation. The number of processes and dataset size are varied automatically as configured in this file.
8) mm_gray_scott_mpi.yaml: This will launch pymonitor and then launch the MPI-based Gray Scott implementation. The number of processes and dataset size are varied automatically as configured in this file.

Evaluation 2: Increasing Dataset Resolution: This evaluation focuses on Gray Scott for different dataset resolutions. These experiment is defined by the files in: test/unit/iter-pipelines/df_resolution. There are two:

1) mm_gray_scott_mega.yaml: This will launch the Hermes runtime, launch pymonitor, and then launch MegaMmap's Gray Scott implementation. The resolution of the dataset L is increased for a fixed number of processes and nodes.
2) mm_gray_scott_mpi.yaml: This will launch pymonitor and then launch the MPI-based Gray Scott implementation. The resolution of the dataset L is increased for a fixed number of processes and nodes.

Evaluation 3: Performance Benefits of Persistent Tiered Memory: This evaluation focuses on Gray Scott for different storage hardware compositions. This experiment is defined by the file in: test/unit/iter-pipelines/tiering. There is only one file:

1) mm_gray_scott_mega.yaml: This will launch the Hermes runtime, launch pymonitor, and then launch MegaMmap's Gray Scott implementation. The resolution of the dataset L is fixed to produce 1.5TB of data. The tiering strategy is varied between various percentages of RAM, NVMe, SSD, and HDD.

Evaluation 4: Lowering DRAM Consumption: This evaluation, we run each of the MegaMmap-based applications for different DRAM capacities and offload pages to the nearest tier of storage – in our case NVMe. These experiment is defined

by the files in: test/unit/iter-pipelines/mem_scaling. There are four:

1) mm_kmeans_mega.yaml: This will produce a synthetic dataset that fits in memory, launch the Hermes runtime, launch pymonitor, and then launch a memory-constrained MegaMmap KMeans implementation. The amount of memory dedicated to the algorithm is varied. The number of processes and scale remain constant.
2) mm_rf_mega.yaml: This will produce a synthetic dataset that fits in memory, launch the Hermes runtime, launch pymonitor, and then launch a memory-constrained MegaMmap KMeans implementation. The amount of memory dedicated to the algorithm is varied. The number of processes and scale remain constant.
3) mm_dbscan_mega.yaml: This will produce a synthetic dataset that fits in memory, launch the Hermes runtime, launch pymonitor, and then launch a memory-constrained MegaMmap DBSCAN implementation. The amount of memory dedicated to the algorithm is varied. The number of processes and scale remain constant.
4) mm_gray_scott_mega.yaml: This will launch the Hermes runtime, launch pymonitor, and then launch a memory-constrained MegaMmap Gray Scott implementation. The amount of memory dedicated to the algorithm is varied. The number of processes and scale remain constant.

Artifact Analysis (incl. Outputs)

We do not currently automate figure generation; however the text files produced are in CSV format that can be used to build the figures. By default, in each of the workflow YAML files, the output data is stored in the location $(jarvispath+ shared)/output/stats_dict.csv. This can be tweaked by changing the "output" key at the bottom of each YAML. The detailed description of the outputs are as follows:

Evaluation 1: Scalability of DSMs for HPC: For each of the 8 Jarvis YAMLs, a stats_dict.csv will be created containing the application name (e.g., KMeans-Mega), the number of processes, the dataset size, memory utilization, and the overall runtime of the program. It should be found that the Spark-based algorithms take significantly longer (in our case 2x) longer to run than the MegaMmap-based algorithms. In addition, they use significantly more memory (up to 4x) than the MegaMMap implementation. For the MPI-based algorithms, MegaMmap should perform similarly in terms of both memory and runtime – demonstrating a DSM abstraction can perform competitively to traditional HPC designs.

Evaluation 2: Increasing Dataset Resolution: For each of the two Jarvis YAMLs defined in this evaluation, a stats_ dict.csv will be created containing the application name (e.g., GrayScott-Mega), the number of processes (constant), the resolution of the dataset L, the overall runtime of the program, and the memory utilization. It should be found that the MPI-based implementation of Gray Scott will fail after the cluster memory is expended. MegaMmap should continue running even after the memory is fully utilized due to intelligent page eviction. In our case, the memory of our system was 48GB – so after GS produced 48GB of data, only MegaMmap would function.

Evaluation 3: Performance Benefits of Persistent Tiered Memory: For the Jarvis YAML defined in this evaluation, a stats_dict.csv will be created containing the application name (e.g., GrayScott-Mega), the number of processes (constant), the tiering composition (e.g., 48G-DRAM-16G-NVMe), the overall runtime of the program, and the memory utilization. It should be found that SSD-based tiering is faster than HDD (approximately 30%) and NVMe-based is faster than SSD (approximately 15%).

Evaluation 4: Lowering DRAM Consumption: For each of the Jarvis YAMLs defined in this evaluation, a stats_dict.csv will be created containing the application name (e.g., Kmeans-Mega), the number of processes (constant), the overall runtime of the program, and the memory utilization. It should be found that decreasing memory capacity of the application effects performance minimally. Memory should be able to be reduced by at least 50% while incurring minimal performance loss. Eventually, after memory is reduced too much (i.e., cut in half or third), applications become noticeably slower.