DI-MMAP – A Scalable Memory-Map Runtime for Out-of-Core Data-Intensive Applications

Brian Van Essen† Henry Hsieh†‡ Sasha Ames† Roger Pearce†∗ Maya Gokhale†
†*Center for Applied Scientific Computing*
Lawrence Livermore National Laboratory, Livermore, CA 94550
{*vanessen1, hsieh7, ames4, pearce7, gokhale2*}*@llnl.gov*

‡*Department of Computer Science, University of California, Los Angeles*
∗*Department of Computer Science and Engineering, Texas A&M University*

***Abstract*—We present DI-MMAP, a high-performance runtime that memory-maps large external data sets into an application's address space and shows significantly better performance than the Linux `mmap` system call. Our implementation is particularly effective when used with high performance locally attached Flash arrays on highly concurrent, latency-tolerant data-intensive HPC applications. We describe the kernel module and show performance results on a benchmark test suite, a new bioinformatics metagenomic classification application, and on a level-asynchronous Breadth-First Search (BFS) graph traversal algorithm. Using DI-MMAP, the metagenomics classification application performs up to $4\times$ better than standard Linux `mmap`. A fully external memory configuration of BFS executes up to $7.44\times$ faster than traditional `mmap`. Finally, we demonstrate that DI-MMAP shows scalable out-of-core performance for BFS traversal in main memory constrained scenarios. Such scalable memory constrained performance would allow a system with a fixed amount of memory to solve a larger problem as well as provide memory QoS guarantees for systems running multiple data-intensive applications.**

***Keywords*-data-intensive; memory-map runtime; memory architecture; NVRAM;**

I. Introduction

Data-intensive applications form an increasingly important segment of high performance computing workloads. These applications process large external data sets and often require very large working sets that exceed main memory capacity, presenting new challenges for operating systems and runtimes. In this work, we target a data-intensive node architecture with direct I/O-bus-attached Non-Volatile RAM, such as Flash arrays today, and STT-RAM, PCM, or memristor in the future. These persistent memory technologies provide new opportunities for extending the memory hierarchy by supporting highly concurrent read and write operations that can be exploited by throughput driven (latency tolerant) algorithms such as parallel graph traversal [1].

In this work, we advocate a memory-mapping approach that maps low latency, random access storage into an application's address space, allowing the application to be oblivious to transitions from dynamic to persistent memory when accessing out-of-core data. However, we, along with many others, have observed that the memory-map runtime in Linux is not suited for memory-mapped out-of-core applications [2] and cannot efficiently support this model. In Linux, even with highly optimized massively concurrent algorithms and high bandwidth low latency storage, applications designed to interact with very large working sets in main memory incur significant performance loss if they read and write data structures that are memory-mapped from external storage.

For this reason, most out-of-core algorithms use explicit I/O to load and store data between external store and application-managed data buffers. Optimizing an application for out-of-core execution is an exercise in carefully choreographing data movement, requiring explicit data requests through direct I/O and manual buffering [3].

The idea of memory-mapping data from storage into main memory is appealing for its simplicity. Additionally, it paves a path for scalable out-of-core computation because buffering and data movement are implicitly handled by the operating system's runtime rather than the application.

In prior work [2], we demonstrated that the standard memory-map runtime in Linux will rapidly lose performance as concurrency increases and as memory within the system becomes constrained. Therefore, we have developed a new high-performance runtime that can seamlessly integrate NVRAM into the memory hierarchy using the memory-map abstraction. Our new module, a data-intensive memory-map runtime (DI-MMAP) addresses the performance gap in the standard Linux memory-map implementation.

This paper demonstrates the effectiveness of DI-MMAP for data-intensive applications. We demonstrate that DI-MMAP can consistently achieve significant performance improvement over standard Linux `mmap` on our test suite, including an unstructured read/write access pattern, micro-benchmarks that demonstrate searching several types of data structure, a bioinformatics application that searches a large (hundreds of GB) "in-memory" metagenomics database, and a level-asynchronous BFS algorithm. Our memory-map runtime delivers up to $4\times$ the performance of standard Linux `mmap` on the bioinformatics application and approaches the

peak performance of raw, direct I/O on a random I/O benchmark. Furthermore, executing a fully external memory BFS algorithm on DI-MMAP is 7.44× faster than with Linux `mmap`. Finally, DI-MMAP shows scalable out-of-core performance for BFS traversal in main memory constrained scenarios (*e.g.* 50% less memory with only a 23% slowdown), allowing a system with a fixed amount of memory to solve a larger problem. Alternatevely, it would provide memory QoS guarantees for systems running multiple data-intensive applications.

II. The DI-MMAP runtime

The data-intensive memory-map runtime (DI-MMAP) is a high performance runtime that provides custom memory-map fault handling and page buffering. It is a loadable Linux character device driver and it works outside of the standard Linux page caching system. It was first introduced in [4] and is derived from the PerMA simulator outlined in [2], sharing a common core codebase. Source code is available at [5]. It has been developed and tested for the 2.6.32 kernels in RHEL6.

The key features of the runtime are:

- a fixed size page buffer organized into multiple page management queues
- minimal dynamic memory allocation
- a simple FIFO buffer replacement policy
- tracking and sampling of page faults
- preferential caching for frequently accessed pages
- bulk TLB eviction

The combination of these features allows DI-MMAP to provide exceptional performance at high levels of concurrency compared to standard `mmap`, as shown in Section VI. The DI-MMAP device driver is loaded into a running Linux kernel. As it is loaded, the device driver allocates a fixed amount of main memory for page buffering. Using static page allocation versus dynamic page allocation improves performance by approximately 4%, see Section VI-B. Once the device driver is active, it provides two mechanisms for interaction, a direct mapped method for block devices and a DI-MMAP file system. For the direct mapped interface, it creates a control interface file in the `/dev` filesystem. The control file is then used to create additional pseudo-files in the `/dev` filesystem that link (*i.e.* redirect) to block devices in the system. When a pseudo-file is accessed all requests are redirected to the linked block device.

The DI-MMAP file system provides a file system overlay for an existing directory in the standard Linux file system. When the `di-mmap-fs` is loaded it is supplied with an existing (backing) directory within the Linux file hierarchy and it will create a virtual file at the new mount point for every file in the backing directory. Similarly to the direct mapped interface, when a file within a `di-mmap-fs` mount point is accessed, all requests are redirected to the underlying file in the backing directory.

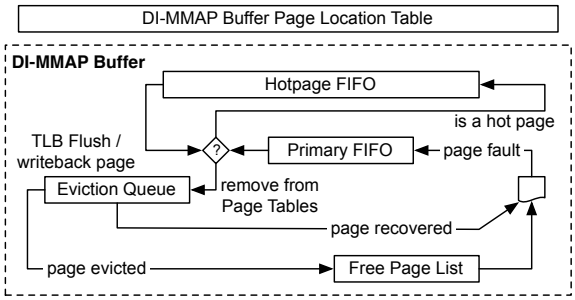


Figure 1. DI-MMAP page buffer

DI-MMAP uses a simple FIFO buffering system with preferential storage of frequently accessed pages (*i.e.* a hot page FIFO). Figure 1 shows a logical diagram of the DI-MMAP buffer and its page management queues. The buffer contains enough pages to fill all of the queues plus one spare page. When a page fault occurs, the page location table is checked to see if another process (or thread) has already faulted the page into the buffer. If the page is in the buffer, the page is added to the page table of the faulting process and the fault is completed. Otherwise, a free page is allocated from a pool of empty pages. Data is then read from the block device into the fresh page, and the page is queued into the series of FIFOs.

In the steady state, a page fault will displace the oldest page in the primary FIFO. If the displaced page has been faulted more frequently than the buffer's average it will be placed into the hot page FIFO, otherwise it will be placed into the eviction queue. When a newly displaced page is inserted into the hot page FIFO, it will displace an older hot page, which is then placed in the eviction queue. Once a page is in the eviction queue, it will eventually be flushed to storage if dirty, cleaned and returned to the free page list. The buffer page location table is implemented as a hash table with chaining. To maintain a long term fault history, the buffer page location table maintains a fault count per hash bucket, which is used as the starting fault count for each new page that maps to that bucket. Similarly to a bloom filter or branch history table, this provides an approximate history that is based on the ratio of number of unique pages to number of buckets.

The value of tracking page faults and using a hot page FIFO to store active pages is quantified in Figure 13, Section VI-E2, for a BFS graph traversal. These results highlight the overhead and inefficiency for many data-intensive applications of the Linux `mmap` policy of evicting the Least Recently Faulted (LRF) page. In particular, while a Least Recently Used (LRU) policy is generally good for data with temporal locality, the Linux page cache lacks a mechanism to track page accesses or the history of page faults and fault frequency. Therefore the eviction algorithm is limited to evicting the least recently faulted page, regardless of its actual use, and is unable to provide more comprehensive policies for frequently accessed pages.

Another important aspect to maintaining performance is to properly manage TLB occupancy and eviction. Examples of the performance loss that can occur due to excessive TLB thrash have been noted by other research projects, such as Wu et al.'s [6] work on storage class memory. To address these problems, DI-MMAP removes pages from the page table of every process (it was mapped in to) as they are scheduled for eviction, but the translation look-aside buffers (TLBs) are flushed in bulk (only when the eviction queue is full). Figure 5 in Section VI-B illustrates the value of a bulk flush versus individual TLB page invalidation. Another optimization is page recovery, which is based on the well known technique of using victim buffers with caches [7]. The eviction queue provides the functionality of a victim buffer as well as a sampling window for tracking page fault activity to identify hot pages. When a page fault occurs for a page that is in the eviction queue, it is not flushed out. Instead, it is put into the primary FIFO, and the page's fault counter is incremented to indicate that it has some temporal locality (thus it might be a hot page). When a page is finally removed from the buffer and placed in the free page list, the page's fault counter is used to update the buffer page location table bucket's fault counter. If the page's fault counter is higher than the bucket's fault counter then the bucket's fault counter is set to equal the page's. If the page's and bucket's fault counter are equal, the bucket's counter is decremented (min. of one), *i.e.* decaying the fault counter.

Another feature of the DI-MMAP kernel module is that it can be loaded multiple times. This allows for multiple instances of the runtime, each of which provides an independent buffer. The impact of multiple buffers on applications is the subject of future research.

III. Related Work

Providing more control, and application specific-control, over memory page management is not a new idea. Previously, there were several research efforts focused on the virtual memory management system in the Mach 3.0 micro-kernel that have yet to be revisited for modern HPC operating systems. They studied the effects of different page eviction policies, application-specific pools of pages, and even application defined replacement policies.

The HiPEC project by Lee et al. [8] developed a small programming language that was used to create application-specific replacement policies for the virtual memory runtime. They show that for a particular phase of an application with nested loops, switching from a traditional least-recently-used (LRU) page replacement policy to a most-recently-used (MRU) policy can speedup the loop execution by $\sim 2\times$ by reducing page faults.

The work of Park et al. [9] used the flexibility of the Mach micro-kernel to directly develop multiple memory management policies for memory-mapped data. Park demonstrated again that a performance gain of $\sim 2\times$ was achievable with a replacement policy that matched an application's I/O patterns. Furthermore, they also provided an example of how to customize a replacement policy for the access patterns of a specific data structure. Finally, they demonstrated that with the right memory management policy, it is possible to get scalable out-of-core execution.

Qureshi et al. [10] studied the impact of alternate insertion policies for a CPU's cache. They found that several alternate line insertion policies provided excellent performance for cyclic access patterns. Specifically, by marking new cache lines as least recently used (LRU) rather than most recently used (MRU), large cyclic access patterns would not trash the cache's entire contents and would actually achieve modest cache reuse.

All of these previous research projects have demonstrated that customized memory management and paging policies can dramatically improve a system's performance. They demonstrated that scalable performance is possible as applications shift from in-memory to out-of-core computations. The proliferation of data-intensive applications and high performance NVRAM storage provides compeling motivation to revist these ideas in modern HPC operating systems.

IV. Data-Intensive Computing Applications

We focus on high performance computing data-intensive applications that

- analyze hundreds of GiB to TiB size data sets
- have algorithmic data structures whose sizes often don't fit in main memory
- may display irregular random memory access behavior
- can exploit massive thread level concurrency

Our goal is to enable parallel algorithms tuned for memory locality to interact with large data sets as if in memory by mapping data structures to files stored in locally attached enterprise grade Flash arrays. To better assess the performance of DI-MMAP on scientific and data analysis problems, we study two realistic data-intensive applications: metagenomics classification for pathogen detection and breath-first search (BFS) graph traversal. Each application is described in greater detail below.

A. Livermore Metagenomics Analysis Toolkit (LMAT)

Metagenomics involves the sequencing of heterogenous genetic fragments taken from the environment, in which the fragments (also called "reads") may be derived from many organisms. This area is extremely beneficial for numerous applications in bioinformatics, e.g. to discover toxic organisms in a biological sample. Sequencing technologies are increasing their rate of output; thus, there is a pressing need for accelerating sequence classification algorithms to keep pace with the sequencer improvement rate. The Livermore Metagenomics Analysis Toolkit (LMAT) is a new bioinformatics application developed at LLNL to identify pathogens in samples containing an unknown variety of

biological material. The LMAT classification application queries a database of genetic markers called k-mers, which are length k contiguous sequences of DNA bases that appear in a genome. LMAT is highly data-intensive: the input consists of millions of reads of length 50-100 bases, and the constituent k-mers of those reads must be searched in the k-mer database. The reads are independent, and thus can be analyzed in parallel without requiring synchronization. The k-mer database is stored as a single large file in order to best classify the k-mers in a read according to their position in the taxonomy of all known genomes.

We place the large (hundreds of GiB) k-mer database in Flash storage and memory-map the database file to access the k-mers and associated taxonomy classification information. Memory-mapping the database file eliminates the need to explicitly load it into main memory, a drawback of previous approaches. Additionally, Flash storage gives a lower-cost alternative to large-memory machines, but the challenge is to use caching techniques that reduce the performance penalty incurred by using Flash instead of DRAM. The access patterns to the datasets are extremely random. Thus, performance optimizations for rotating media — with sequential access preferred — do not apply well to this workload.

The metagenomic database contains k-mer markers referring to genomes from within a reference database (set of collected genomes) along with additional data associating the k-mer with a genome and the genome's position in the taxonomy tree of organisms. To facilitate indexing, k-mers are encoded as 64-bit integers [11]. Each integer serves as the key to the k-mer index. The integer keys map to pointers, which in turn refer to the associated values: binary data storing lists of 32-bit taxonomy identifiers and 16-bit count fields (for use within classification) for the organisms containing the k-mer marker within the reference database. The length of these lists spans from a few to (rarely) thousands of taxonomy identifiers, where each taxonomy identifier indicates a parent of the k-mer in the taxonomy tree. The k-mer lookups are performed concurrently using OpenMP threading.

We use two forms of index data structures for mapping k-mers to their constituent organisms. The first uses a *gnu* hash map with the k-mer as key, and pointers to the associated genomes and taxonomy information as value. A lookup retrieves the associated data (taxonomic information), which ranges from hundreds to thousands of bytes.

The second type of index is a "two-level" index. The first level maps the high-order bits of an integer-encoded k-mer to a pointer into a second level of lists, each of which is sorted. The second level uses the remaining low-order bits to find a pointer to the taxonomy information. These second-level lists fit within a single page of memory and are quickly binary searched during a k-mer lookup operation. The value storage is the same as used with gnu hash. Figure 2 illustrates

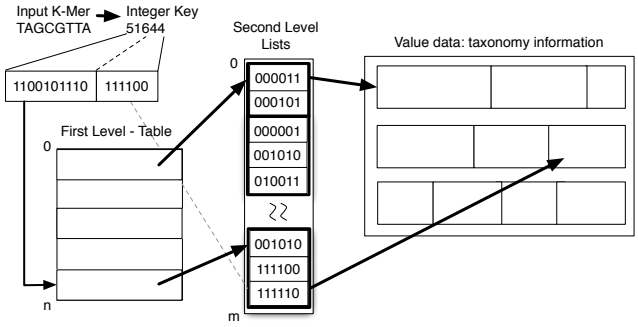


Figure 2. Structure of the two-level index. A k-mer shown in ascii form is encoded as an integer. Its binary representation is split to show use within the levels of the index.

the structure of the index, where n is the length of the first-level array (corresponding to the number of high-order bits selected), and m is the length of the second-level array of lists. Note that m is equal to the total number of k-mers in the database. This data structure is specific for integer keys and is tuned for k-mer data. It is not suitable for general purpose key-value storage that typically performs a hash-function calculation on variable-length strings of ascii characters.

The structure of both the gnu hash and two-level index generally cause the overall memory access patterns within the data structure to be randomly distributed. However, the layout of the two-level approach introduces some spatial locality between independent lookups in two ways. First, the upper level of the two-level index is a smaller structure than the gnu hash table. By virtue of a smaller size, we expect fewer capacity misses and greater reuse of cache pages than the gnu hash table. Second, the gnu hash uses linked-list chaining to resolve hash collisions. These chains span multiple pages of memory. In contrast, the second level lists usually fit on a single page of memory (in exceptional cases, some lists span the boundaries of multiple consecutive pages), thus enabling highly localized access.

LMAT uses k-mer lookup as a frequent kernel operation. The application processes input files containing reads from the metagenomic samples, where each sample is a list of reads of 50 to 100 characters each. Once the application has queried the index using the extracted k-mers, it uses those results — the presence of particular taxonomic identifiers — to assign a label identifying an organism or higher taxonomic entity to each read. Input data is trivial to partition for processing in parallel; thus, many classification procedures are run concurrently using OpenMP threads.

B. HavoqGT: BFS Graph Analysis

Large graph analysis is one of the driving examples of data-intensive problems. Traversing the graph typically produces an unstructured sequence of memory references that have very little temporal or spatial locatily. Additionally, graph traversal has a low computation to communication ratio, as the bulk of the work is to access vertexes and edges. The Highly Asynchronous VisitOr Queue Graph Toolkit

(HavoqGT) is being developed at LLNL, and implements a parallel, level-asynchronous, Breadth-First Search traversal well suited to large (e.g. 2^{31} nodes) scale-free graphs. Our algorithm uses the vistor abstraction in which a small function is applied to each graph vertex. The function reads and writes priority queues associated with the graph vertexes and edges. The algorithm is asynchronous: each visitor function is applied independently, and it is not necessary to synchronize at the end of each level of the graph [1].

Our goal in evaluating the BFS algorithm was to measure performance under several memory partitioning scenarios. The algorithm has two classes of data structures: algorithmic data structures used during search, such as the visitor queues, and the graph itself. To enable us to experiment with various partitions of in-memory vs. out-of-core alloctions using the same BFS executable, we manually partitioned the Breadth-First Search algorithmic data structures and the graph data structures into independent memory regions. This partitioning enabled the data structures to be backed by independent memory-mapped files, and allowed us to run the algorithm fully in-memory, semi-externally (partially in-memory, meaning that the algorithmic data structures are allocated in memory and the graph data is allocated to storage), and fully externally, all using exactly the same binary. The partitioning alternatives were exercised by placing the files in a combination of `tmpfs` and NVRAM Flash storage.

In the semi-external configuration the entire graph is mapped to Flash and is read-only. The access pattern to the graph is unstructured. In a fully external configuration the algorithmic data structures, which are read/write data structures, are also on Flash. The algorithmic data represents the minority of the total data, but the majority of the total memory accesses. Additionally, roughly one third of the algorithmic data exhibits temporal locality, while the remaining two thirds have access patterns that are mostly unstructured. These varied usage patterns allowed us to evaluate the utility of the buffer management algorithms, especially the interaction between hot page FIFO and eviction queue.

This method of partitioning the BFS data structures also made it possible to evaluate performance of DI-MMAP in memory constrained environments by offloading what would traditionally be heap allocated data structures to persistent storage.

V. Experimental Methodology

The DI-MMAP runtime is designed to provide high performance on highly-concurrent, data-intensive workloads. To test DI-MMAP we use four types of benchmarks: a synthetic random I/O workload, a set of three microbenchmarks, a metagenomics classification application, and a level-asynchronous, breadth-first search graph traversal. The synthetic random I/O workload was chosen because it is a good approximation for the unstructured access patterns found in many data-intensive applications. The micro-benchmarks are three commonly used data traversal and search algorithms. Finally, both the LMAT classifier and HavoqGT BFS traversal are highly data-intensive applications.

There are two common approaches to testing DI-MMAP. In both approaches data was loaded onto one or more PCIe-attached Flash storage card(s). In the first approach, the DI-MMAP runtime creates pseudo-devices that linked to the raw Flash cards. Each benchmark then memory-maps the DI-MMAP pseudo-device(s), enabling all page faults for the mapped address range to be serviced and buffered by the DI-MMAP runtime. The second approach mounts the Flash cards in the Linux file system with DI-MMAP creating a second mount point that is backed by the Flash device's mount point. In the DI-MMAP mount point a file for each file in the backing store is created, which will redirect accesses to the underlying file on the backing device. Both approaches provide similar levels performance, with each providing a unique method for accessing Flash storage: either a bag of bits, or a traditional file system. These results are then compared to the existing Linux memory-map runtime and to direct (unbuffered) I/O as appropriate.

A. LRIOT

The Livermore Random I/O Testbench (LRIOT) is a synthetic benchmark that is designed to test I/O to high-performance storage devices. We have developed LRIOT to augment the industry standard FIO benchmark for testing high data rate memory-mapped I/O with different process/thread combinations. LRIOT can generate tests that combine multiple processes and multiple threads per process to simulate the highly concurrent access patterns of latency tolerant data-intensive applications. Furthermore LRIOT can generate uniform random I/O patterns that mimic the unstructured access patterns of algorithms such as breadth-first search graph analysis [2]. LRIOT can also do standard and direct I/O in addition to memory-mapped I/O, and thus provides a common testing framework. Finally, the LRIOT benchmark has been validated against the FIO benchmark and provides comparable results for direct I/O.

B. Micro-benchmarks

To complement the LRIOT experiments, we tested three micro-benchmarks that reproduce memory access patterns common to data-intensive applications. The micro-benchmarks are: binary search on a sorted vector, lookup on a ordered map structure that is implemented as a red-black tree, and lookup on an unordered map structure implemented as a hash map. The micro-benchmarks use the C++ STL and Boost library implementations of these algorithms.

C. LMAT

We perform two types of experiments to evaluate DI-MMAP using the metagenomic database. First, we report the performance of a raw k-mer lookup benchmark. Second,

we report the performance of the LMAT application. The k-mer lookup test application reports timings of many single lookups, while the LMAT application times input processing, k-mer lookups, sample classification and output. In these scenarios, we compare the performance when using standard Linux `mmap` to map a file and DI-MMAP. We configure DI-MMAP to use 16 GiB of main memory for its page buffer.

Our database of reference genomes contains information from roughly 26,000 organisms from five categories of microorganisms: viruses, bacteria, archaea, fungi and protists. The database was indexed with a k-mer length of 18 for these experiments, which results in an index set with approximately 7 billion k-mers. We present experimental results that include performance of both the gnu hash index and the two-level indexing scheme. The gnu hash database uses roughly 635 GiB of Flash storage and the the two-level index database uses 293 GiB. Specifically for the two-level approach, and a k-mer length of 18, the first level index uses 1 GiB and the second level uses 56 GiB for the 7 billion k-mers. The remaining storage is used for the taxonomic informational data associated with each k-mer. We present results from these two different indexing schemes in separate figures as the focus of these experiments is to compare the performance of DI-MMAP with Linux `mmap`.

For the raw k-mer lookup benchmark experiment, we use the following input sets: first a synthetic metagenome derived from a human gut sample (HC1) and second, three real-world collections of metagenomic samples labeled SRX, DRR, ERR. Using the HC1 input set, we consider a selection of increasing thread counts. Our results from this input set only uses the raw k-mer lookup benchmark, but for both types of indexing.

Using the three real-world metagenomic samples, we include results for both the gnu hash indexing and two-level indexing with both the k-mer lookup benchmark and LMAT application experiments, four experiments in total. In contrast to the sweep of thread counts used with HC1 experiments, we tested the real metagenomic sample input sets using only two thread counts: 16 and 160. These are selected as they are the approximate peak values for Linux `mmap` and DI-MMAP respectively.

D. HavoqGT: BFS Graph Analysis

As noted in Section IV-B the implementation of HavoqGT BFS used in these experiments partitioned all of the data structures that would normally be heap allocated into individual memory regions. This allowed each of the individually memory-mapped files to be stored in memory via `tmpfs` or in Flash storage. The partitioning created five files for the following data structures: a priority queue, BFS progress data, manual cache of vertices, vertex data, and edge data. The data set that was used for these experiments was generated by the R-MAT [12] graph generator, which produces realistic and challenging experiments and is used by the Graph500 [13] benchmark. We used the generator to create a scale-free graph with 2^{31} vertexes, with an average out-degree of 16. The graph instance is labeled RMAT 31 in subsequent figures. The data size for this graph is 146 GiB of vertex and edge data, and requires 24 GiB of BFS algorithmic data that is split evenly among the three algorithmic data structures. When executing the application in a semi-external configuration the 24 GiB of BFS algorithmic data is in memory (using `tmpfs`), 16 GiB is allocated for DI-MMAP's page cache, and only 16 GiB remains free for Linux `mmap`'s page cache. For the fully-external configuration a total of 40 GiB is allocated to DI-MMAP's page cache, and only 40 GiB remains free for Linux's page cache, for the combination of algorithmic and graph data structures.

VI. Results

The following experiments are designed to compare the performance of DI-MMAP relative to the existing implementation of `mmap` in Linux for data-intensive workloads: highly concurrent and when there is insufficient main memory to hold the entire data set. In addition to showing the performance using DI-MMAP, these experiments demonstrate that the efficiency of DI-MMAP can enable an application to execute with less page cache than standard `mmap` without significant loss in performance. This scalability in performance sets the stage for allowing an application to shift part of its algorithmic data out of main memory, thus allowing data-intensive applications to tackle even larger problems. These tests are conducted on a variety of synthetic benchmarks, microbenchmarks, and two data-intensive applications to illustrate the flexibility of the DI-MMAP runtime.

One of the optimizations that we have previously experimented with for Linux `mmap` is to use the `MADV_DONTNEED` flag for the `madvise` system call to help alleviate memory pressure. We have demonstrated the effectiveness of the `MADV_DONTNEED` flag before in both BFS [1] and random I/O workloads [2]. To achieve the maximum benefit from the `madvise` system call the data access pattern has to be unstructured, read-only, and the application needs an additional thread of control that can periodically issue the system call. It should also be noted that the `MADV_DONTNEED` flag cannot be safely used for memory-mapped writable data structures, as the system is not required to write out dirty pages of data. For these experiments we have made comparisons of the efficacy of `madvise` in the context of the HavoqGT BFS traversal which was architected to meet these requirements.

A. LRIOT: Uniform random I/O distribution

The first experiment compares the performance of DI-MMAP, standard `mmap`, and direct I/O. LRIOT generated a random sequence of 6.4 million read operations to a raw 128GiB region that was striped across three 80 GiB SLC

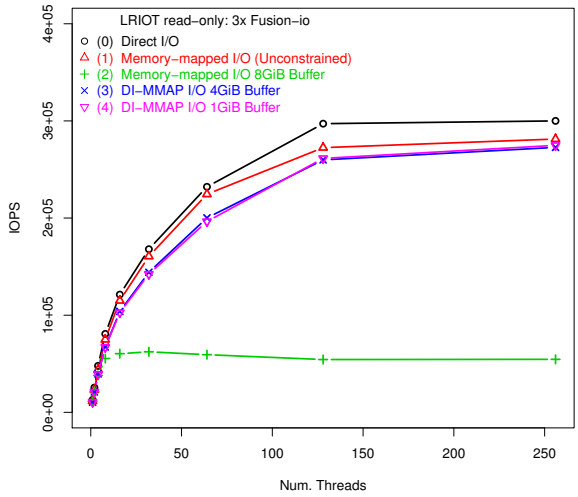


Figure 3. Read-only random I/O benchmark with uniform distribution.

NAND Flash Fusion-io ioDrive PCIe 1.1 x4 in a RAID 0 configuration. The input read sequence is constructed so that it is repeatable, has one address per page, and is unique per process. Therefore each test will fetch 6.4 million unique pages, about 24 GiB of data. The data transfer size for all I/O (direct and memory-mapped) was 4KB pages. The host system was a 16 core AMD 8356 2.3GHz Opteron system with 64 GiB of DRAM and running RHEL 6 2.6.32.

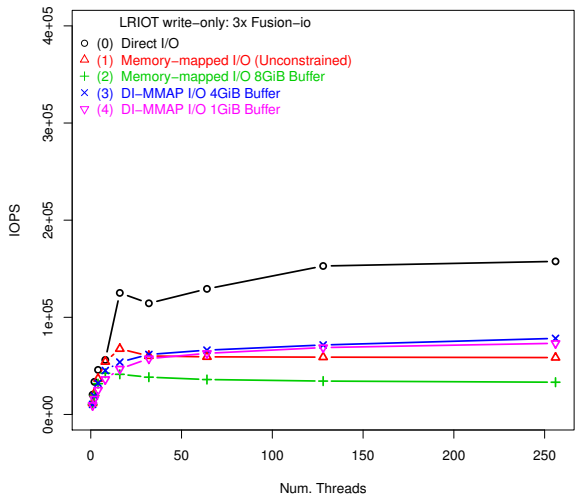


Figure 4. Write-only random I/O benchmark with uniform distribution.

Figure 3 shows the number of I/O per second (IOPs) that LRIOT achieved for the different I/O methods as concurrency increased. Note that each test used one process and the x-axis shows the number of concurrent threads. There are 5 specific test configurations shown here. The first line is for direct I/O and is typically the upper bound on achievable performance for a set of devices. The second and third lines are for the standard Linux memory-map handler when there is sufficient memory to hold all pages that are accessed in memory, *i.e.* `mmap` buffering is unconstrained, and when the page cache is constrained to hold only 8 GiB of pages. Finally, curves four and five are for DI-MMAP with a fixed buffer size of 4 GiB and 1 GiB, respectively. Figure 3 shows that the performance of DI-MMAP is very close to the performance of direct I/O and `mmap` when unconstrained, even with a very small buffer size of 1GiB. Furthermore, Figure 3 shows that standard Linux `mmap` performs well when memory is unconstrained, but performance drops significantly when system memory is constrained and the requested data exceeds the capacity of main memory. Overall, we see that DI-MMAP is able to deliver near peak performance with limited buffering resources, with only a 13% loss in IOPs compared to direct I/O at 128 threads.

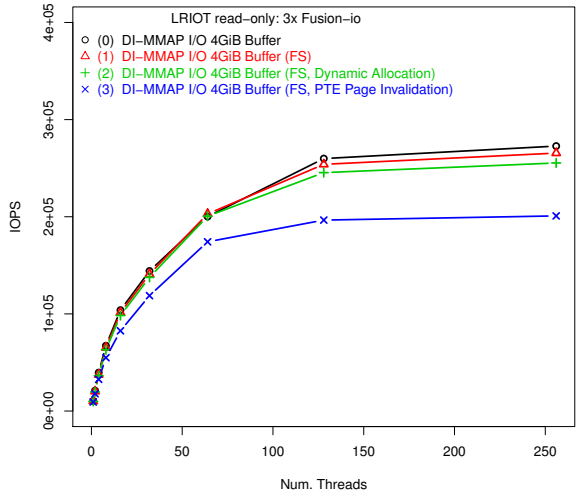


Figure 5. Impact of DI-MMAP optimizations with read-only random I/O benchmark.

Figure 4 shows the number of IOPs that LRIOT is able to achieve with a similar write-only working set. As with the read-only working set, DI-MMAP offers a bit more than double the performance of Linux memory-map when it is constrained and similar performance to the unconstrained Linux mmap. However, unlike the read-only test, the performance of DI-MMAP does not match that of direct I/O and is the subject of further investigation.

B. Testing the impact of DI-MMAP optimizations

Section II describes the DI-MMAP runtime and specifically highlights several optimizations that provide a performance advantage over the standard Linux `mmap`. Figure 5 show the impact of several of DI-MMAP's optimizations, specifically dynamic memory allocation, and bulk cleanup of the TLB. Similar to the previous figures, Figure 5 plots performance in terms of IOPs versus parallelism (# of threads). The first line in Figure 5 shows the raw performance of DI-MMAP, with all optimizations, and is the same result as the third line from Figure 3. This result

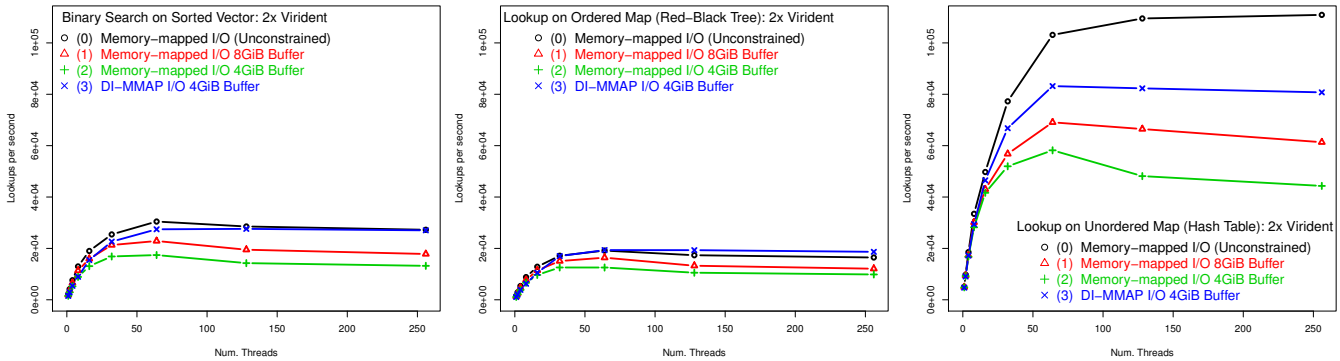


Figure 6. Micro-benchmarks: Binary Search on Sorted Vector, Lookup on an Ordered Map, and Lookup on an Unordered Map, respectively. Note that the data sets were constructed to be approximately the same size, and thus have different numbers of elements.

illustrates the peak peformance for the DI-MMAP runtime for these experiments.

The second line illustrates the cost of using DI-MMAP to access a file on an `ext2` filesystem. For this and subsequent tests, the 128GiB file is striped across all three Fusion-io cards in the RAID0 array. Figure 5 shows that at 256 threads, the cost of accessing data on `di-mmap-fs` versus a raw device is a loss of 2.6% in IOPs. The third line uses both `di-mmap-fs` to access the file and allocates a new page of data on each page fault. The Linux O/S is highly tuned for dyamic allocation of whole (raw) pages and the overhead for dynamic page allocation (vs. static allocation) was only 3.8% slower than DI-MMAP on the file system. Finally, the fourth line shows the performance of LRIOT reading a file in `di-mmap-fs` and without using the bulk TLB flush that is enabled by the victim queue. Instead, for each page that is evicted from the buffer, that specific entry is individually flushed from all TLBs. The number of IOPS achieved for DI-MMAP without bulk TLB flush is 0.756× the IOPS of LRIOT and `di-mmap-fs`, a loss of 24.4% in performance.

C. Micro-benchmarks

The three micro-benchmarks were all performed on an 8 core AMD 2378 2.4GHz Opteron system with 16 GiB of DRAM and two 200 GiB SLC NAND Flash Virident tachIOn Drive PCIe 1.1 x8. The database size for the vector and maps ranged from $\sim$ 112GiB to $\sim$ 135GiB and each micro-benchmark issued 2^{20} queries. For each of the graphs in Figure 6 performance is measured in lookups per second and the x-axis is the number of concurrent threads. In each figure, line one is the performance of Linux `mmap` with unconstrained memory, lines two and three are the performance of Linux `mmap` with 8 and 4 GiB of available buffering (respectively), and line four is the performance of DI-MMAP with 4GiB of available buffering. These figures show that the performance of DI-MMAP significantly exceeds the performance of Linux `mmap` when each is constrained to an equal amount of buffering, and in some cases the performance with DI-MMAP is able to approach the performance of `mmap` with no memory constraints.

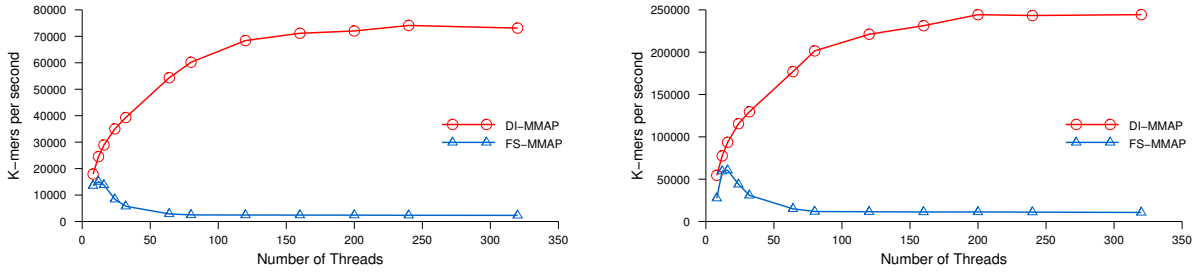
D. Raw k-mer lookup and LMAT classification

The LMAT experiments were performed on a 4 socket, 40 core, Intel E7 4850 2 GHz system, with 1 TiB of DRAM, running Linux kernel 2.6.32-279.5.2 (RHEL 6). For storage we use a software RAID over two Fusion-io 1.2 TB ioDrive2 cards, formatted with block sizes of 4 KiB, and the system was constrained to have 16 GiB DRAM available for DI-MMAP's or `mmap`'s page cache.

Figure 7(a) shows the performance of the raw k-mer lookup benchmark using the synthetic (HC1) input set with gnu hash indexing. The x-axis denotes increasing numbers of threads used for each trial and the y-axis shows k-mers per second. When using 8 threads, k-mer lookup performs better using DI-MMAP than standard `mmap` with a file system, and the performance gap increases with additional concurrency. While performance decreases with increasing threads using standard `mmap`, the opposite is true for use of DI-MMAP. Notably, the performance with standard `mmap` peaks at 16 threads and then degrades. The peak performance for DI-MMAP with 240 threads is 4.92× better than the peak performance for standard `mmap` with 16 threads.

Figure 7(b) is similar to Figure 7(a), but shows raw k-mer lookup for the two-level index with `mmap` and DI-MMAP. Again, we measure peak performance for `mmap` at 16 threads, in this case at 60,600 k-mers per second. In contrast, our peak measurement for DI-MMAP is at 244,000 k-mers per second, roughly 4× faster. Note that the peak performance for both DI-MMAP and `mmap` is significantly higher than when using the gnu hash table. In summary, Figures 7(a) and 7(b) show that that DI-MMAP performs well with two very different types of data structures: gnu hash and a two-level index. Additionally, the figures show that performance scalability (with threads) follows a similar pattern for both `mmap` and DI-MMAP using either index structures.

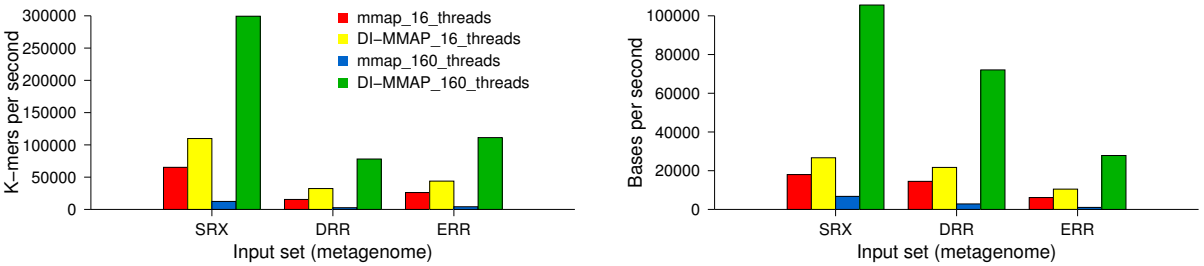
The following two Figures 8, and 9 show performance of the two applications and two indexing data structures using real metagenomes. A difference between the plots in Figures 8(a)-9(a) and 8(b)-9(b) is that the y-axis on the latter pair denotes bases per second (from the input) files rather than k-mers per second. This metric is necessary to normalize



(a) gnu hash indexing

(b) two-level indexing

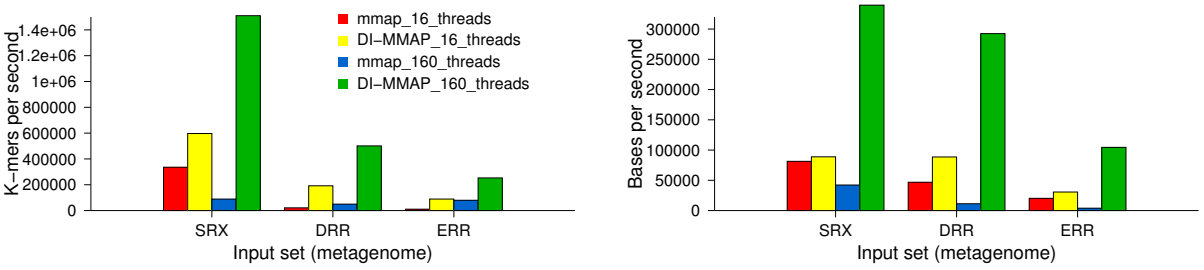
Figure 7. Performance of raw kmer lookup using k-mer identifiers extracted from the HC1 input set and using gnu hash indexing and two-level indexing.



(a) Raw K-mer Lookup

(b) LMAT

Figure 8. Performance of raw k-mer lookup and LMAT classification application using three metagenomic input sets and gnu hash indexing.



(a) Raw K-mer Lookup

(b) LMAT

Figure 9. Performance of raw k-mer lookup and LMAT classification application using three metagenomic input sets and two-level indexing.

performance among differing input sets, where the length of each read (a line of bases taken from the sequencer) may vary. These observations fit with those measured for the synthetic workload.

We observe that the performance differences between DI-MMAP and `mmap` are greater for the raw k-mer lookup benchmark than for LMAT classification. Several factors influence differences in performance between the two applications. The classification algorithm uses considerably more system memory for processing over the raw lookup, whose usage is negligible besides caching of the k-mer index and associated data. Additionally, the classification algorithm spends considerable more CPU time in the actual classification phase of processing rather than the k-mer lookup phase.

We observe differences among the different input sets for several reasons. First, each has a different percentage of redundant k-mers. Increased redundancy improves performance, since more k-mers hit in the buffer cache. For instance, the SRX input data set produces a much greater cache hit rate than the other two sets. Thus, that data set consistently shows a higher throughput in all four experiments.

Second, for the classification comparisons, the diversity of the metagenome (number of organisms represented) impacts its performance. We observe that DRR performs relatively better than ERR when comparing k-mer lookup performance with LMAT classification performance (Figure 9). We attribute this difference to the greater diversity in the ERR input set, which increases classification time but does not impact lookup time. However, DRR is also faster than ERR for raw k-mer lookup and we attribute that to greater redundancy in the DRR input set. Considering the measurements of the these data sets using gnu hash indexing, we observe DRR performing better than ERR for LMAT classification

in Figure 8(b). This result we expect given the difference in diversity. However, our measurements in Figure 8(a) show DRR performing worse, and this result we attribute to the inconsistent nature of gnu hash indexing: the k-mers in the DRR set favor chaining, which forces additional lookup time. Nonetheless, neither of these properties of the input sets have a considerable impact on the relative performance differences observed between `mmap` and DI-MMAP.

We observe a range of peak performance speedup factors for DI-MMAP vs `mmap`: from the gnu hash experiments, the greatest of 4× for the SRX input set and smallest of 2.7× for the ERR input set; from the two-level index experiments, the greatest of 3.8× for the SRX input set and smallest of 3.3× for the DRR input set.

In addition to comparing DI-MMAP with Linux `mmap`, we can observe better performance of the two-level index vs. gnu hash index through the pairs of figures: 7(a) vs. 7(b) and 8 vs. 9. The performance speedup factors differ by input data set and range from 2.3× to 6× depending on the input set. These speedups we attribute to both the improved locality and smaller overall size of the two-level index. Specifically, given the 16GiB buffer and this database, 5.6% of the two-level hash index can fit in buffer memory, while only 2.5% for the gnu hash.

E. HavoqGT: BFS Graph Analysis

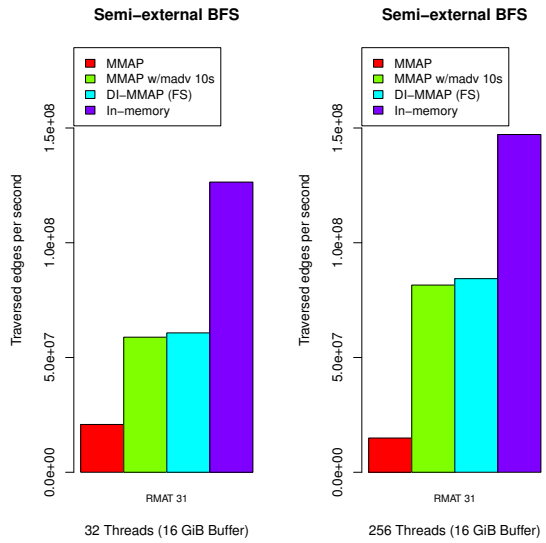


Figure 10. Comparing HavoqGT semi-external BFS on mmap, di-mmap, and tmpfs.

The HavoqGT BFS graph traversal experiments were executed in three distinct configurations: in-memory, semi-external, and fully-external. The host system was a 32 core AMD 6128 2.0GHz Opteron system with 512 GiB of DRAM, with two SLC NAND Flash Virident tachIOn Drive PCIe 1.1 x8 cards (one 200 GiB and one 300 GiB capacity), running RHEL 6 2.6.32. The Virident cards were placed in a RAID 0 configuration and then split into 4 partitions. The performance for each of these experiments is measured in Traversed Edges Per Second (TEPS), and is plotted against the number of concurrent threads. Each configuration is tested with one thread per core, and with 8× thread oversubscription at 256 threads.

The in-memory experiments placed all five algorithmic and graph data files in memory via a `tmpfs` mount. For the semi-external configurations of BFS the graph vertex and edge data was placed on one partition that was mounted with an `ext2` file system and the algorithmic data files were kept in `tmpfs`. Due to limitations in the write performance on `ext2` file systems and the current lack of support for `xfs` in DI-MMAP, the fully external configuration placed the three algorithmic data structures in each of three partitions of the software RAID and used the DI-MMAP direct mapped interface. The graph data files remained on the `ext2` partition of the RAID device.

1) Semi-external BFS execution: Figure 10 shows the performance of the HavoqGT BFS algorithm in a semi-external configuration on Linux `mmap` and DI-MMAP versus in-memory execution. The four data points in Figure 10 are for `mmap`, `mmap` with a helper thread that issues an `madvise` system call every 10 seconds using the `MADV_DONTNEED` flag, DI-MMAP, and in-memory. We see that performance with standard `mmap` is quite poor; however Figure 10 demonstrates the effectiveness of the `MADV_DONTNEED` flag with the second bar of Figure 10. By tuning the `madvise` helper thread to the application, it is possible to make standard `mmap` perform very close to DI-MMAP. DI-MMAP achieves about half of (0.57×) the in-memory performance.

Figure 10 shows that the tuned BFS semi-external algorithm with `mmap` and `madvise` perform quite well, almost as well as DI-MMAP, with sufficient buffering available. One of the advantages previously demonstrated for DI-MMAP was the ability to work with less page buffering. Figure 11 shows the performance of both `mmap` and DI-MMAP as the amount of system buffering is reduced from 16GiB down to 3GiB. We see that DI-MMAP performs significantly better than Linux `mmap` when the buffer size is scaled down.

2) Fully-external BFS execution: An alternative execution environment for the HavoqGT BFS algorithm is to place all of the memory-mapped data files into Flash memory, creating a fully external memory execution. The key difference of this environment versus the semi-external environment is that the three memory-mapped files of algorithmic data are written as well as read, and that some of the data access patterns exhibit good temporal locality and are amenable to traditional caching techniques. Note that the `madvise` system call is only used on the memory-mapped (read-only) vertex and edge graph data, not the algorithmic data, and is issued every 10 seconds. Figure 12 shows that the performance of the DI-MMAP is 7.44× better than `mmap` on the fully-external execution of the BFS algorithm when using 256 threads.

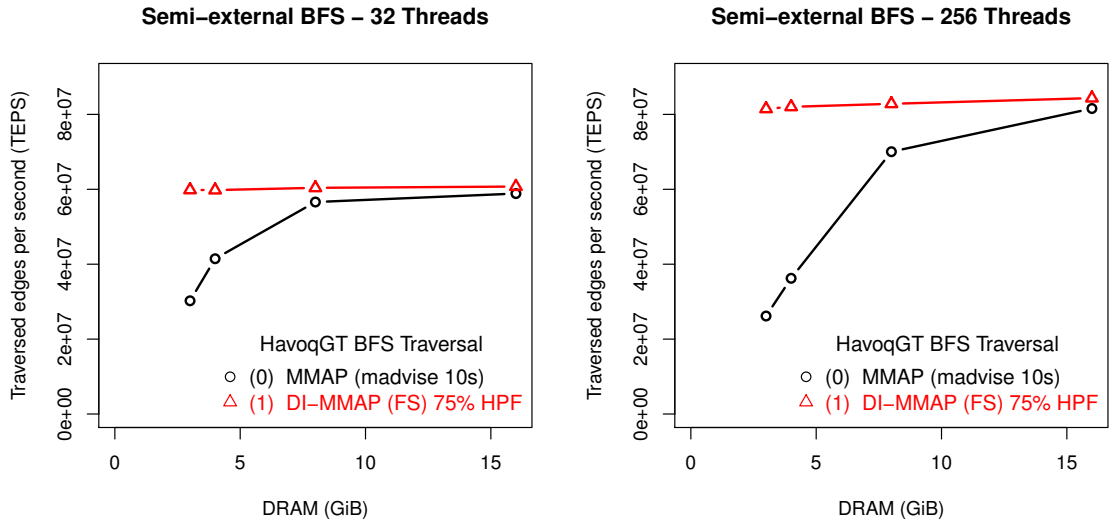


Figure 11. Impact of reducing system memory on HavoqGT semi-external BFS algorithm. DI-MMAP allocated 75% of its page cache to the hot page FIFO (HPF)

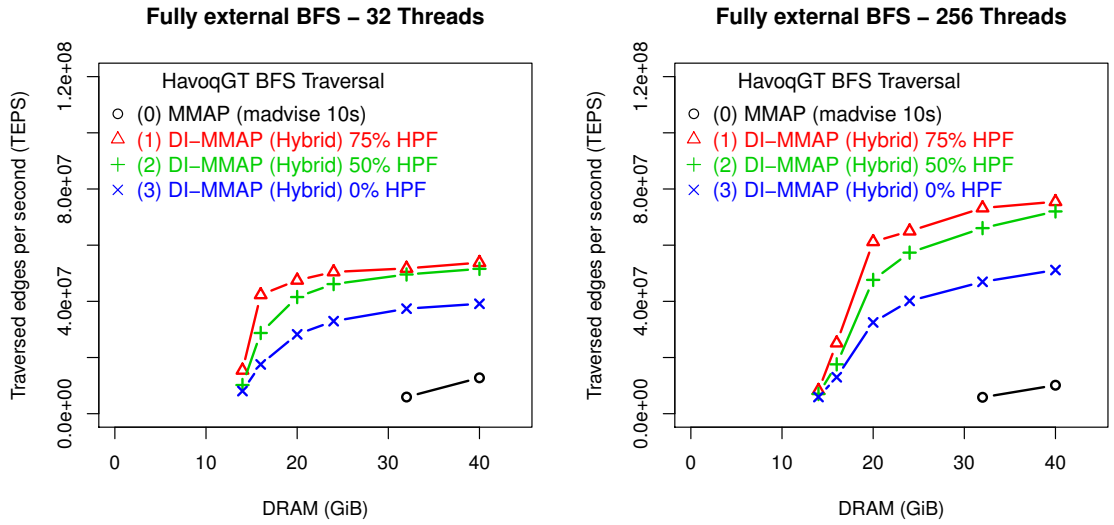


Figure 13. Impact of reducing system memory on HavoqGT fully external BFS algorithm. Additionally, a comparison of DI-MMAP with differing quantities of the buffer dedicated to the hot page FIFO (HPF).

System memory is frequently a limiting factor in the size of problem that a data-intensive application is able to solve. Figure 13 shows the performance of the fully external BFS alogrithm on DI-MMAP and `mmap` as the size of the page cache is reduced. For DI-MMAP the trade-off in performance versus buffer size is more dramatic than for the semi-external algorithm due to the access pattern of the algorithmic data and the need to write out dirty pages of algorithmic data. When using 75% of the buffer for a hot page FIFO, as the buffer size is scaled from 40 GiB down to 20 GiB the performance is only 1.23× slower when executing with 256 threads. The ability to support such a dramatic reduction in main memory requirement provides the opportunity for a system of a fixed size to solve a much larger problem. When using Linux `mmap` and the buffer size is scaled from 40 GiB down to 32 GiB the performance is already 1.74× slower when executing with 256 threads. Note that Linux `mmap` was not tested with even less memory, as the performance had already dropped off dramatically.

Figure 13 also demonstrates the efficacy of a hot page buffer. As previously noted in [2] the asymmetric access patterns of the fully external BFS search does benefit from a hot page buffer that will catch the temporal locality of the algorithmic data. In Figure 13 line 1 has 75% of the buffer allocated to the hot page buffer, line 2 has 50%, and line 3 has no hot page buffer. Using 75% of the buffer for hot pages, a 40GiB buffer, and 256 threads provides a 1.48× improvement in performance over not having a hot page buffer.

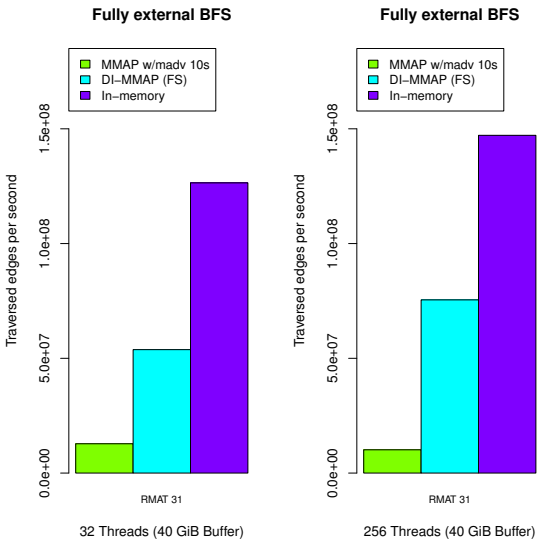


Figure 12. Comparing HavoqGT fully-external BFS on mmap and di-mmap.

VII. Conclusions

The goal of the data-intensive memory-map (DI-MMAP) runtime is to provide scalable, out-of-core performance for data-intensive applications. We show that the performance of algorithms using DI-MMAP scales up with increased concurrency, and does not significantly degrade with smaller memory footprints. As such, DI-MMAP provides a viable solution for scalable out-of-core algorithms. DI-MMAP offloads the explicit buffering requirements from the application to the runtime, allowing the application to access its external data through a simple load/store interface that hides much of the complexity of the data movement.

We demonstrate the performance of DI-MMAP over Linux's existing memory-map runtime with a simple random I/O workload, three micro-benchmarks, a metagenomics classification application, and a level-asynchronous breadth-first search graph traversal. Our results show that as the tests increase in complexity the performance of DI-MMAP can be 4× to 2.7× better than standard Linux `mmap` for the metagenomics sample classification application, and up to 7.44× better for a fully external BFS traversal. Furthermore, the use of DI-MMAP alleviates the need to implement a custom, user-level buffer caching algorithm and infrastructure to achieve high performance.

Acknowledgments This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-JRNL-612114). Funding partially provided by LDRD 11-ERD-008, LDRD 12-ERD-033, and the ASCR DAMASC project. The metagenomic classification algorithm was developed by Jonathan Allen, David Hysom, and Sasha Ames, all of LLNL. Portions of experiments were performed at the Livermore Computing facility resources, with special thanks to Dave Fox and Ramon Newton.

References

[1] R. Pearce, M. Gokhale, and N. M. Amato, "Multithreaded asynchronous graph traversal for in-memory and semi-external memory," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

[2] B. Van Essen, R. Pearce, S. Ames, and M. Gokhale, "On the role of NVRAM in data intensive HPC architectures: an evaluation," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Shanghai, China, May 2012, pp. 703–714.

[3] J. S. Vitter, "Algorithms and data structures for external memory," *Foundations and Trends in Theoretical Computer Science*, vol. 2, no. 4, pp. 305–474, 2006.

[4] B. Van Essen, H. Hsieh, S. Ames, and M. Gokhale, "DI-MMAP: A high performance memory-map runtime for data-intensive applications," in *International Workshop on Data-Intensive Scalable Computing Systems (DISCS-2012)*, Nov. 2012.

[5] "Data-centric Computing Architectures Research Group," https://computation.llnl.gov/casc/dcca-pub/dcca/Data-centric_architecture.html.

[6] X. Wu and A. L. N. Reddy, "Scmfs: a file system for storage class memory," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 39:1–39:11.

[7] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 364–373.

[8] C.-H. Lee, M. C. Chen, and R.-C. Chang, "HiPEC: high performance external virtual memory caching," in *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, ser. OSDI '94. Berkeley, CA, USA: USENIX Association, 1994.

[9] Y. Park, R. Scott, and S. Sechrest, "Virtual memory versus file interface for large, memory-intensive scientific applications," in *Proc. ACM/IEEE Conf. Supercomputing*, 1996.

[10] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proceedings of the 34th annual international symposium on Computer architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 381–391.

[11] G. Marcais and C. Kingsford, "A fast, lock-free approach for efficient parallel counting of occurrences of k-mers," *Bioinformatics*, Jan. 2011.

[12] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Fourth SIAM International Conference on Data Mining*, April 2004.

[13] "Graph500," www.graph500.org.