Towards Optimizing Memory Mapping of Persistent Memory in UMap

Karim Youssef*†, Keita Iwabuchi†, Wu-chun Feng*, Maya Gokhale†, Roger Pearce†
*Virginia Tech
{karimy,feng}@cs.vt.edu
† Lawrence Livermore National Laboratory
{iwabuchi, gokhale2, pearce7}@llnl.gov

***Abstract*—The exponential growth in data set sizes across multiple domains creates challenges in terms of storing data efficiently as well as performing scalable computations on such data. Memory mapping files on different storage types offers a uniform interface as well as programming productivity for applications that perform in-place or out-of-core computations. However, multi-threaded applications incur an I/O contention on mapped files, hampering their scalability. Also, many applications handle sparse data structures, rendering storage efficiency a desirable feature. To address these challenges, we present *SparseStore*, a tool for transparently partitioning a memory-mapped persistent region into multiple files with dynamic and sparse allocation. We provide *SparseStore* as a part of UMap, a user-space page management tool for memory mapping [1]. Our experiments demonstrated that using UMap with *SparseStore* yielded up to 12x speedup compared to system-level mmap, and up to 2x speedup compared to the default UMap that maps a single file.**

***Index Terms*—dataset sizes, persistent memory, memory mapping, user-space, parallel I/O, sparse data structures**

I. Introduction

The exponential growth in data set sizes across many scientific domains presents multiple challenges for high-performance computing systems to store and retrieve data efficiently as well as to perform scalable computations on such datasets [2]. To cope with these needs, leadership supercomputers began incorporating new storage technologies such as node-local persistent memory, NVMe SSDs, and network interconnected flash memory. These storage technologies optimize speed, cost, and data persistence, benefiting applications that perform out-of-core computations.

Memory-mapping of files on different types of storage devices provides a simplified and portable interface to applications. However, system-level memory mapping, e.g., *mmap*, rely on the operating system for page management, which lacks flexibility for application-specific optimizations. This fact motivated the design of user-space solutions for page management [3].

UMap is a tool for memory mapping using user-space page management [1]. It enables the configuration of multiple page management parameters such as page size, eviction strategy, degree of concurrency, and prefetching. This design enables application-specific performance optimizations, however, parallel I/O performance is a crucial factor that was not addressed. Since most data analytics and scientific computing applications are multi-threaded, the I/O contention on mapped files represents a bottleneck to their scalability. Moreover, many applications handle sparse data structures such as graphs, rendering storage efficiency of memory-mapped files a desirable feature. Storage efficiency is particularly important for applications that allocate and store data structures on persistent memory for later reuse [4].

To address these challenges, we augment UMap with *SparseStore*, a backing store handler that transparently partitions a memory-mapped persistent region into multiple files and assigns each file to a virtual memory address range. The *SparseStore* handler also uses a dynamic and sparse allocation strategy that only creates backing files on-demand. The goal of this design is to improve the I/O performance of multi-threaded applications, as well as the storage efficiency of sparse data structures on persistent memory devices. We provide a detailed performance analysis of UMap with our store handler for two applications, a dynamic graph construction on persistent memory, and a 512GB out-of-core sorting. Using UMap with *SparseStore* yielded up to 12x speedup compared to system-level *mmap*, and up to 2x speedup compared to UMap with the default store handler that maps a single file.

II. Design

UMap offers an extensible design that allows application developers to implement application-specific or storage-specific backing store handlers by extending a *Store* object. A *Store* object defines methods for interfacing a backing store, i.e., reading and writing. We leveraged this extensible design to implement the sparse multi-file store handler, called *SparseStore*. Since our design would benefit various applications and storage types, we provided *SparseStore* implementation as a part of the UMap library.

Figures 1 illustrates the design and usage of the *SparseStore* handler. the *SparseStore* handler transparently partitions a mapped persistent region into segments, and creates a separate file for each segment. The segment granularity is a configurable parameter. When memory-mapping a virtual memory region using UMap and SparseStore for the first time (i.e., mapping an empty file), no file descriptor is actually mapped. Instead, the *SparseStore* handler is linked with a directory on the backing store. The backing files are then created dynamically and on-demand. When a page fault occurs, UMap

invokes the read or write routines from the *SparseStore* object. The *SparseStore* maps the starting address of the requested page to a file index and a file offset. If the file with the mapped index does not exist, it is created and opened; otherwise, it is opened directly. The *SparseStore* object keeps a list of opened file descriptors and closes all files when their mapping virtual memory region is un-mapped. The *SparseStore* also keeps metadata information such as the segment granularity to enable future mapping of a sparse multi-file persistent region.

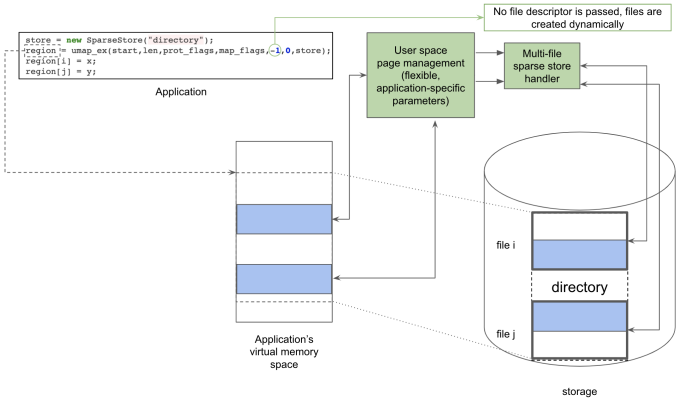

Fig. 1: *A SparseStore object partitions the mapped persistent region into multiple files. Initially, the virtual memory region does not map any file, as multiple files are created dynamically. Previously created persistent regions using SparseStore can also be mapped and grown dynamically.*

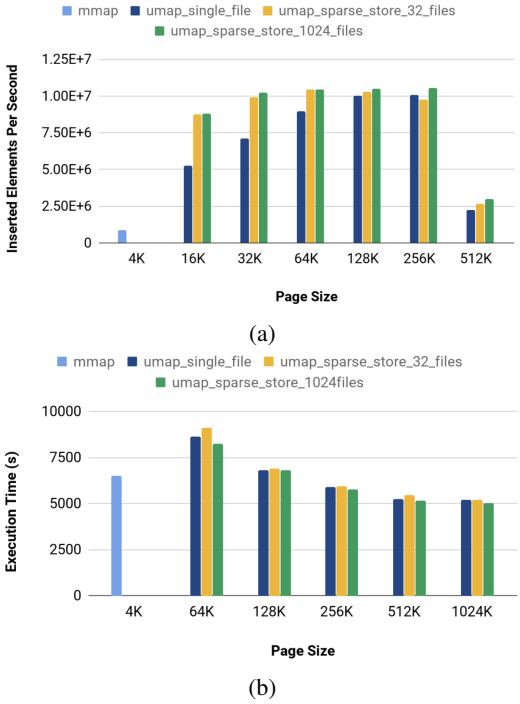

Fig. 2: *Performance evaluation of UMap with SparseStore for dynamic graph construction (a), and 512GB out-of-core sorting (b). The dynamic graph construction performance is measured in inserted elements per second, while sorting performance is measured in sorting time in seconds.*

III. Evaluation

We evaluated the performance of UMap with the *SparseStore* handler using two applications, a dynamic graph construction on persistent memory, and an out-of-core sorting of 512GB. Experiments were conducted on an AMD testbed consisting of 2 AMD EPYC 7401 processors with 24 cores and 48 hardware threads each, 256GB DDR4 DRAM, and 1.8T local NVMe SSD. The platform runs Centos 7 with Linux kernel 5.6.0-rc2-uffd-wp-v6-r1-amd-g42355f8.

We configured each benchmark to use either mmap, UMap with the default store handler, or UMap with *SparseStore*. For the *SparseStore* configuration, we changed the number of backing files to be either 32 or 1024 files. We changed the UMap page size between 16KB and 1MB. A separate experiment was performed to obtain the optimal number of threads to use for each benchmark. The graph construction used 48 threads, while the sort used 96 threads. Results are shown in figure 2.

For the graph construction benchmark, it was observed that using UMap with the *SparseStore* handler, 1024 files, and 32KB page size, yielded nearly a 12x speedup over system-level *mmap*. Also, using 16KB page size, the *SparseStore* configuration yielded nearly 2x speedup compared to UMap with the default store handler that maps a single file. This observation could be explained by the fact that smaller page sizes lead to more reads and writes to the backing store and hence more I/O contention for the single file configuration.

For the sorting benchmark, a slight 6% improvement was observed for the *SparseStore* configuration with 32 files at 64KB page sizes. Since the sorting benchmark does not start benefiting from UMap before 128KB page size, and since increasing the number of files was observed to benefit page sizes of 64K and smaller, no significant difference was observed between the single file and the *SparseStore* configurations for this particular benchmark. For larger page sizes, it was observed that the performance of all UMap configurations converge.

Finally, it was noticed that increasing the UMap page size improved the performance of the sorting benchmark for up to 1MB , while the graph construction benchmark performed best at 128KB, then at 512KB, performance degraded significantly. This may be due to the irregular memory access pattern of the graph construction which leads it to stop benefiting from larger page sizes.

IV. Acknowledgment

Prepared by LLNL under Contract DE-AC52-07NA27344 (LLNL-ABS-813826). Experiments were performed at the Livermore Computing facility. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

REFERENCES

[1] I. Peng, M. McFadden, E. Green, K. Iwabuchi, K. Wu, D. Li, R. Pearce, and M. Gokhale, "Umap: Enabling application-driven optimizations for memory mapping persistent store," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2019.

[2] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson, "Big data: astronomical or genomical?" *PLoS biology*, vol. 13, no. 7, p. e1002195, 2015.

[3] J. Corbet, "Page faults in user space: Madv_userfault, remap_anon_range(), and userfaultfd()," *http://lwn.net/Articles/615086*, vol. 2, 2014.

[4] K. Iwabuchi, L. Lebanoff, M. Gokhale, and R. Pearce, "Metall: A persistent memory allocator enabling graph processing," in *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 2019, pp. 39–44.