Large-Scale Adaptive Mesh Simulations Through Non-Volatile Byte-Addressable Memory

Bao Nguyen
School of Engineering and Computer Science
Washington State University
Vancouver, WA 98686
bao.nguyen@wsu.edu

Hua Tan
School of Engineering and Computer Science
Washington State University
Vancouver, WA 98686
hua.tan@wsu.edu

Xuechen Zhang
School of Engineering and Computer Science
Washington State University
Vancouver, WA 98686
xuechen.zhang@wsu.edu

ABSTRACT

Octree-based mesh adaptation has enabled simulations of complex physical phenomena. Existing meshing algorithms were proposed with the assumption that computer memory is *volatile*. Consequently, for failure recovery, the in-core algorithms need to save memory states as snapshots with slow file I/Os. The out-of-core algorithms store octants on disks for persistence. However, neither of them was designed to leverage unique characteristics of non-volatile byte-addressable memory (NVBM). In this paper, we propose a novel data structure **P**ersistent **M**erged octree (PM-octree) for both meshing and in-memory storage of persistent octrees using NVBM. It is a multi-version data structure and can recover from failures using its earlier persistent version stored in NVBM. In addition, we design a feature-directed sampling approach to help dynamically transform the PM-octree layout for reducing NVBM-induced memory write latency. PM-octree has been successfully integrated with Gerris software for simulation of fluid dynamics. Our experimental results with real-world scientific workloads show that PM-octree scales up to 1.1 billion mesh elements with 1000 processors on the Titan supercomputer.

CCS CONCEPTS

• **Software and its engineering** → **Memory management**; • **Mathematics of computing** → *Mesh generation*;

KEYWORDS

Octree, Adaptive Mesh Refinement, Non-volatile Byte-addressable Memory

ACM Reference format:
Bao Nguyen, Hua Tan, and Xuechen Zhang. 2017. Large-Scale Adaptive Mesh Simulations Through Non-Volatile Byte-Addressable Memory. In *Proceedings of SC17, Denver, CO, USA, November 12–17, 2017,* 12 pages.
DOI: 10.1145/3126908.3126944

SC17, Denver, CO, USA

DOI: 10.1145/3126908.3126944

1 INTRODUCTION

Octree-based adaptive mesh refinement on high-performance computing (HPC) clusters has enabled simulation of complex physical phenomena with 3D modeling, e.g., droplet ejection in inkjet technology [21, 43], droplet impact on a solid surface [24, 59], and rapid boiling flow [34], among many others. Computational scientists seek to run their physics models at ever-larger length- and time-scales such that memory demands for running such simulation is significant even on supercomputers [23]. At the same time, it has become increasingly difficult to scale DRAM to higher capacities because of associated capital costs and power consumption: the price of DRAM larger than 128GB per node can quickly go beyond tens of thousands of dollars, and the power consumed by DRAM can be up to 40% [51] of that of the entire HPC cluster. Further, considering the fact that per-core DRAM capacity has been shrinking in recent years, we must seek more cost-effective solutions to extend memory capacity available to large-scale meshing and visualization using octree data structures. The *non-volatile, byte-addressable memory* (NVBM) technologies, e.g., Phase Change Memory (PCM) [10], STT-MRAM [6], and Memristor [58], are promising enabling technologies for alleviating these problems.

However, on the software side, no existing meshing algorithms can take full advantage of NVBM. (1) The existing in-core meshing algorithms [41, 42, 46] were proposed based on the assumption that computer memory is *volatile*. These algorithms did not explore *non-volatility* of NVBM. For example, Gerris [4] uses an in-core meshing algorithm for simulations of computational fluid dynamics. It needs to save memory states to snapshot files for failure recovery. The incurred I/O operations can cause a severe performance bottleneck on storage systems and a waste of CPU cycles [62, 63]. (2) The out-of-core meshing algorithms (e.g., Etree [44]) were designed for non-volatile storage mediums (e.g., solid-state disks and hard disks) on I/O bus. They should not be directly used for meshing on NVBM because I/O bus is much slower than memory bus. I/O optimization techniques used in these algorithms (e.g., data indexing) only incur additional memory latency, which may offset the benefits of NVBM.

In this work we propose a novel meshing algorithm, that allows an octree data structure resident in both DRAM and NVBM via a memory interface supported by operating systems/runtime[1]. However, there are three challenges in the design of an NVBM-aware octree because some physical aspects (e.g., write latency and endurance) of NVBM

[1] Assume NVBM is attached to the CPU bus alongside DRAM, and the OS/R is able to support dynamic allocation from NVBM devices [3, 17].

9-11 3 notes:

are so much different from those of DRAM. Existing solutions have been oblivious of these differences.

First, NVBM writes may incur high latency. On the one hand, the read latency of NVBM is comparable to DRAM. On the other hand, its write latency is 2.5X greater than that of DRAM [29, 39, 50]. At the same time, octree meshing operations can be write-intensive. For the cases related to our research of fluid dynamic simulations, memory writes account for up to 7[illegible] and 41% on average, of the total number of memory accesses. Therefore, placing the octrees in NVBM may expose the long write latency to mesh operations and result in significant loss of performance.

12 chen ping

13 chen ping

Second, the existing octree data structure is not durable for NVBM. Corruption is likely when updating the octree in NVBM in the face of hardware or software failures. For example, if an octree node (octant) needs to be refined, we can initialize a new octant and then add a pointer linking the two octants. Because CPU cache does not guarantee the order of writing the octant and writing the pointer for optimization of memory access, the pointer might be written to NVBM before the new octant is written. A failure between the two writes can cause the pointer to link to an undefined region in NVBM. Although it is possible to enforce the order by issuing *mfence* and *clflush* CPU instructions [50], such solutions suffer from two major limitations: (1) it is not sufficient to provide atomic writes on data larger than 8 bytes; and (2) executing these instructions after every write can incur high overhead [33].

14 chen ping

15-16 2 notes:

17-18 2 notes:

Last but not least, the legacy octree solutions do not handle special pointers linking persistent octants in NVBM and volatile octants in DRAM. When a program exits, all the volatile octants should become persistent and pointers from the volatile octants to the persistent octants should be destroyed. When a program recovers from a failure, pointers from the persistent octants to the volatile octants should be updated according to the addresses of newly allocated memory regions in DRAM. In addition, we need a library that can automatically handle such special pointers during failure recovery without introducing extra complexity for application developers.

19-20 2 notes:

In this paper, we propose a novel data structure *persistent merged octree* (PM-octree) for both meshing and in-memory storage of persistent version of octrees. It is a multi-version data structure. Special instructions for enforcing ordering of memory access is needed because our algorithms can guarantee at least one version of the octree is consistent while updating its newer version. If a failure happens, the consistent version of the octree will be accessed for restarting a program. The octree is partitioned so that NVBM-induced additional memory latencies can be effectively shielded. To this purpose, we propose a feature-directed sampling approach to identify popular sub-domains using application-level knowledge about data features, realized as methods for refinement, coarsening, and solving. In addition, we also dynamically adjust the size of sub-trees in DRAM and NVBM to improve memory efficiency according to the memory utilization tracked by operating systems. Specifically, we made the following contributions.

21 chen ping

22-23 2 notes:

24 chen ping

- We propose a novel multi-version data structure PM-octree which not only effectively extends memory capacity using NVBM, but also supports near-instantaneous failure recovery through exploring non-volatility of NVBM.

25 chen ping

- We design algorithms using feature-directed sampling approach to detect the condition of transforming PM-octree layout for reducing NVBM-induced memory write latency and the number of writes to NVBM.
- PM-octree supports orthogonal persistence for applications and provides an easy-to-program interface, with which users are freed from error-prone and tedious tasks of persistent pointer management.
- We implemented a software prototype of PM-octree and its algorithms and integrated them with open-source Gerris flow solver. Our evaluation results with one case study on the simulation of droplet ejection validate the correctness of the algorithms and show that PM-octree scales up to 1.1 billion elements with 1000 processors on the Titan supercomputer.

The rest of the paper is organized as follows. In Section 2 we introduce background and related work. Section 3 presents operations of PM-octree, layout transformation of PM-octree for efficient data access to NVBM, and programming interface of PM-octree for failure recovery. Section 4 discusses the integration of PM-octree with Gerris. Section 5 describes and analyzes experimental results and Section 6 concludes.

2 BACKGROUND AND RELATED WORK

An octree [30] refers to a class of hierarchical spatial tree structures: quadtree for 2-dimensional space, octree for 3-dimensional space, and hyperoctree for dimensions higher than three. We will focus our discussion on the 3-dimensional octree as an example because it is extremely prevalent in scientific simulation and analysis work. The octree index recursively partitions the 3D space into 8 sub-regions (also called octants or "cells") using separators parallel to the coordinate axes. The spatial partition stops at a pre-defined granularity or condition, which is determined by the simulation. Each internal node of an octree has at most 8 entries, each corresponding to one of the 8 cells at that level of partitioning granularity. A leaf node in an octree only has one entry and corresponds to one cell. Figure 1 illustrates a quadtree example, which is explicitly stored in DRAM and used to represent a decomposed square domain for fluid dynamics simulation.

The recursive partitioning using octree may stop at a node as long as a certain pre-specified condition is met; for example, the difference between the minimum and maximum values recorded within the volume is less than a target error. This makes octree very useful to represent multi-resolution datasets: regions of interest can be partitioned at finer granularity to gain a deeper level of detail, while un-interesting or slowly varying regions can be quickly summarized without further lookup. Consequently, the octree is widely used with adaptive mesh refinement in large-scale simulations of fluid dynamics, which have extremely high memory demands.

Octree meshing is typically composed of the 5 major routines: (1) creating a new octree on each processor (*Construct*); (2) adding and/or deleting octants to refine and/or coarsen a domain (*Refine &Coarsen*); (3) enforcing 2:1 constraint on the entire parallel octree (*Balance*); (4) redistributing octants among processors (*Partition*);

[2]We use a 2D quadtree to illustrate the concepts, but all techniques are designed and implemented for 3D cases.

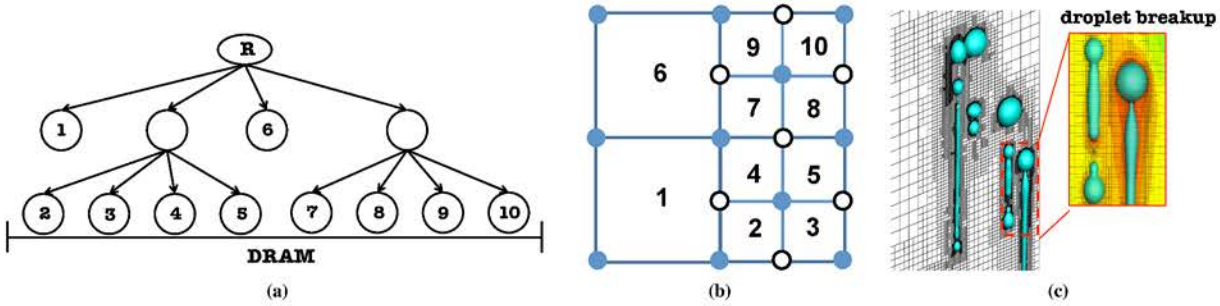


Figure 1: An example of quadtree mesh[2]. (a) Tree representation of a quadtree. *R*: root node of the quadtree in DRAM. (b) Decomposition of a domain using the quadtree. An element of the domain is represented as a quadrant in the quadtree. Vertices map to mesh nodes, which can be anchored nodes (dark colored) or dangling nodes (light colored). (c) A real example of 3D mesh extracted from PM-octree used for modeling droplet ejection of inkjet process. After the liquid jet pinches off at nozzle, the jet eventually breaks up into multiple droplets due to capillary instability.

(5) creating a mesh data structure for visualization (*Extract*). *Balance* is enforced on the fly to guarantee two neighboring octants should be at most twice as large or small. *Extract* is used with data analytics and visualization and is typically executed on demand. These routines may require massive DRAM space to store unstructured data using a complex pointer-based octree data structure.

To extend memory space using NVBM for memory-intensive applications, a wide range of systems and libraries have been developed to address performance and consistency issues caused by slow writes of NVBM and data corruption under failures. The work related to PM-octree is discussed below.

Durable and Consistent Data Structures: Ordinary octree data structure is *ephemeral* because the old version can be destroyed when a change is made to the tree, leaving only the new one. When an application fails, it is not possible to access the previous versions for recovery using an ephemeral octree. In general, applications designed with tree data structure may use three techniques to ensure data reliability: journaling [20] or leveraging persistent data structure [16] for transactional applications and checkpointing for non-transactional applications [8].

Journaling (write-ahead logging) is mostly used by database systems for updating B/B+-Tree data structures. The updates are written to log files sequentially before writing at its primary location. The logs are used to access an old version of B/B+-Tree when transactions fail. The main disadvantage of this technique is *high I/O overhead* because it requires two disk writes for every update. Leveraging multi-version data structures/shadow paging [31, 49], applications do not need to maintain separate log files for data reliability. Instead, they use copy-on-write to perform all updates, so that the original version of the data is not mutated until the newer version of data becomes persistent. Compared to journaling, this approach requires mutating fewer data. However, it is a non-trivial task to maintain the accessibility of multiple data versions in persistent data structures. In fact, its overhead can be high [15, 31], overshadowing its benefit, especially when data structures mainly reside on slow persistent storage devices, e.g., hard disks.

As NVBM is emerging with many attractive features, novel persistent data structures have been proposed to explore its byte-addressability and non-volatility. For example, CDDS B-Tree [50] was designed to provide both durability and consistency and used for implementation of persistent key-value stores. CDDS B-Tree is derived from multi-version B-Tree data structure [49]. Other researchers have been rethinking the design of B/B+-Tree to embrace NVBM technology when it resides in NVBM. As an example, persistent B+-Tree [12, 39] and NV-tree [57] were proposed to reduce pointer mutations, which are prone to data corruption. However, octree has its unique structure and access pattern, making these works not directly applicable.

To ensure data reliability, scientific applications need to periodically save updates to main data structures (e.g., octree [4] and array [54]) of simulations to snapshot files on a persistent storage medium. These files can then be used for failure recovery. For large-scale scientific applications, parallel I/Os of writing snapshots can easily overload a disk-based storage system and cause a severe performance bottleneck [8, 60, 61]. Simply replacing disks with NVBM cannot explore its byte-addressability [25]. Caulfield et al. [11] studied the impact of non-volatile memory on scientific applications. However, they did not address any NVBM-specific issues, e.g., data consistency. In this paper, PM-octree is designed to serve two purposes: (1) storing persistent versions of data structures in NVBM and (2) effectively extending memory capacity for large-scale meshing. It is inspired by a multi-version octree data structure [38]. However, our focus on hiding NVBM-induced latency and easy-to-program required changes (e.g., re-layout and pointer management) to the design and impacted our implementation.

Adaptive Meshing using Octree-based Data Structures: Octree-based adaptive meshing approaches have been widely used in simulations [4, 23, 26] of unstructured grids to reduce the cost of mesh partitioning, setup, and access. Many existing in-core and out-of-core algorithms of octree meshing have been implemented. However, none of them is optimized for NVBM.

When an octree resides in only DRAM, there are two approaches for *in-core* octree construction. Tu et al. [46] used a top-down approach for creating a Morton-ordered octree, which is optimized for

data access locality of octree traversal. Sundar et al. [41, 42] used a bottom-up approach for octree construction leading to less time to build the tree. In addition, they also proposed a mesh compression technique to reduce memory overhead and the time to perform finite element calculations using the octree. However, they focused only on linear octrees, which do not store pointers to siblings or neighbors. It cannot be applied to general flow solvers (e.g., Gerris [1]) because they require an implementation of *multi-threaded octrees*, which store the pointers for traversal efficiency. PM-octree is a derived version of multi-threaded octree. An end-to-end approach [47] was proposed to reduce I/O overhead by providing in-situ data access to in-memory octrees for all major routines of octree meshing, e.g., mesh generation and visualization. All the existing octree data structures for in-core meshing are ephemeral and cannot provide durability and consistency when they resident on NVBM.

For out-of-core meshing, previous work proposed various approaches to storing octrees on slow persistent storage mediums, e.g., hard disks. For example, Salmon et al. designed an octree-based out-of-core algorithm for solving n-body problems. They stored octants on disks in an order determined by a space-filling curve [18]. Recently, Etree [44, 45] library was designed to store octants as pages on disks. It needs to maintain a B-Tree for indexing the pages. Each octant is assigned a unique key (Z-value) calculated using its locational data and Morton code. An octree was also used as an indexing data structure for out-of-core visualization [48]. These index-based algorithms are not necessary for NVBM because it has short write/read latencies, which are very close to those of DRAM and 4-5 orders of magnitude smaller than disks.

Our Work in Context: As NVBM is nearing its deployment, researchers have been developing various system software and programming tools for NVBM. PM-octree and its library is the only one specifically designed for applications using octree as main data structures in memory, while other work focused on B/B+-Tree and log structure for database and file systems, hashmap and link list for memory cache, and heap managed by programming tools.

In novel database systems/data stores, researchers designed optimized B/B+Tree data structures [12, 39, 50, 57] for NVBM. These systems can guarantee data reliability and consistency without logging, which can be slow due to I/O overhead. For log-based database systems, software overhead can be high [22, 27] for writing log files on NVBM. Therefore, novel logging approaches have been proposed to attack this issue. As an example, NV-logging [22] was designed with a decentralized per-transaction log buffer to reduce lock contention. Kim et al. [27] proposed a new byte-addressable log structure to eliminate I/O overhead of filesystem-based logging.

The file systems [15, 33, 56] designed for NVBM can significantly improve I/O performance by leveraging its low latency and high bandwidth. BPFS [15] can guarantee data consistency for file systems using a multi-version tree structure to manage file blocks. It uses short-circuit shadow paging to reduce the overhead of version management of the tree. Instead of using a tree structure as BPFS, SCMFS [56] was designed to utilize the memory management component in operating systems to map file address to physical address on NVBM. HiNFS [33] focused on reducing NVBM-induced write latency, which can be hidden from critical paths by delaying writes that are allowed to be persisted lazily by file systems.

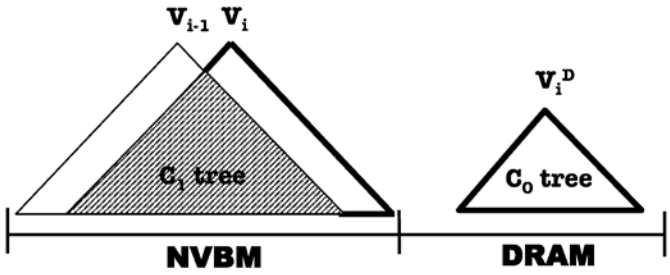


Figure 2: An illustration of PM-octree. V_{i-1} and V_i denote the root nodes of the octrees operated on time steps $i-1$ and i respectively. While V_{i-1} is entirely stored in NVBM, V_i is partitioned into two segments according to data access locality. The C_0 tree composes of frequently accessed subtrees and is stored in DRAM. V_i^D is the root node of the C_0 tree. The C_1 tree composes of subtrees which are less frequently accessed and is stored in NVBM. V_{i-1} and V_i share octants (in the shaded area) not being updated since the completion of T_{i-1}. Only V_i is visible to applications during a normal execution. V_{i-1} is used for restarting applications upon failures.

Other software systems were developed to embrace NVBM technology. Whole system persistence [32] provides near-instantaneous recovery of in-memory state leveraging OS support. Data structures used in memory cache software have been enhanced [37, 55] for data consistency. NVBM has been used in the hypervisors [36] of virtual machines to protect them from failure.

Many general libraries and programming interfaces [13, 14, 28, 53] have been proposed to provide a persistent abstraction for the development of transactional software using NVBM. However, they require the developers to explicitly identify individual allocations as persistent or not and track and manage changes to these within transactions. PM-octree is implemented in a library supporting non-transactional applications by enabling the *orthogonal persistence* [7, 19]. It can identify what data needs to be persistent and implement the persistence for the applications.

3 DESIGN OF PM-OCTREE

The design objective of PM-octree is to effectively utilize NVBM for memory extension and failure recovery. This is achieved by introducing a ***persistent merged octree*** (PM-octree) for meshing and solving using multi-versions of octrees, as shown in Figure 2. In this section, we will first present the overview of PM-octree and then describe major operations of PM-octree, such as tree building, insertion, deletion, traversal, garbage collection, and merging. Further, we will discuss how to use a feature-directed sampling approach to identify frequently accessed subtrees of V_i and its re-layout for reducing NVBM-induced write latency. Finally, we will describe program interface and how to utilize it to restart an application.

3.1 Overview

As shown in Figure 2, PM-octree is a persistent octree data structure, which keeps two of its latest versions V_{i-1} and V_i. V_{i-1} is a persistent version of the octree saved at the end of the simulation of the time

step T_{i-1} and resides entirely in NVBM leveraging its non-volatility for failure recovery. V_i is a version of the octree being actively accessed at the time step T_i. For V_i, its frequently accessed subtrees (C_0) are stored in DRAM and the rest of the tree (C_1) resides in NVBM.

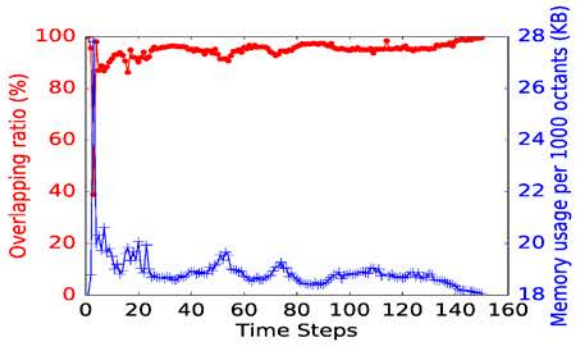


Figure 3: Octant overlapping ratio of V_{i-1} and V_i and memory usage per 1000 octants over 150 time steps in the simulation of droplet ejection.

PM-octree supports shared octants between V_{i-1} and V_i to reduce redundant octants stored in memory, therefore, reducing memory footprint and improving memory utilization. In this paper, we define overlapping ratio as the ratio between the number of octants in the shaded area and the number of octants in V_i. Our study on the simulations of droplet ejection shows that the overlapping ratio of V_{i-1} and V_i ranges from 39% to 99% (shown in Figure 3), indicating a large number of spatial domains does not change in adjacent time steps of the simulation. Figure 3 also shows that PM-octree can reduce the memory usage per 1000 octants by up to 1.98X when the overlapping ratio is increased in the simulation. The factor increase of the memory usage with PM-octree compared with that of storing a single copy of V_i is 1.01 for the last time step when the overlapping ratio is 99.5%, the highest among all the steps.

3.2 Operations of PM-octree

Insertion and updating of an octant: An octant can be inserted into either the C_0 tree or C_1 tree, determined by its locational code. If it is inserted into C_0 in PM-octree, it will be eventually merged out to C_1 in NVBM. The operation of inserting an octant into C_0 does not incur long write latency. However, the size of C_0 is constrained by the size of DRAM. We need to migrate octants of C_0 out to C_1 before OS page swapping [40] starts due to low memory efficiency. For this purpose, we track the percentage of available DRAM space. When it is smaller than a threshold $threshold_{DRAM}$, a least-frequently-accessed subtree will be removed from C_0 and merged with C_1.

If an octant is inserted into C_1 in NVBM, multiple copying steps might be needed to propagate the updates towards the root. An example of inserting octant 11 as a child of octant 9 is illustrated in Figure 4(a). For consistency, we do not directly add a pointer to octant 9. Instead, we need to create a copy (octant 9′) of the octant 9. Then octant 11 is linked to its parent octant 9′. Then we need to set the parent octant u of octant 9 pointing to octant 9′. Again, we must create a copy u' of u and set the initial 4 children of octant u'

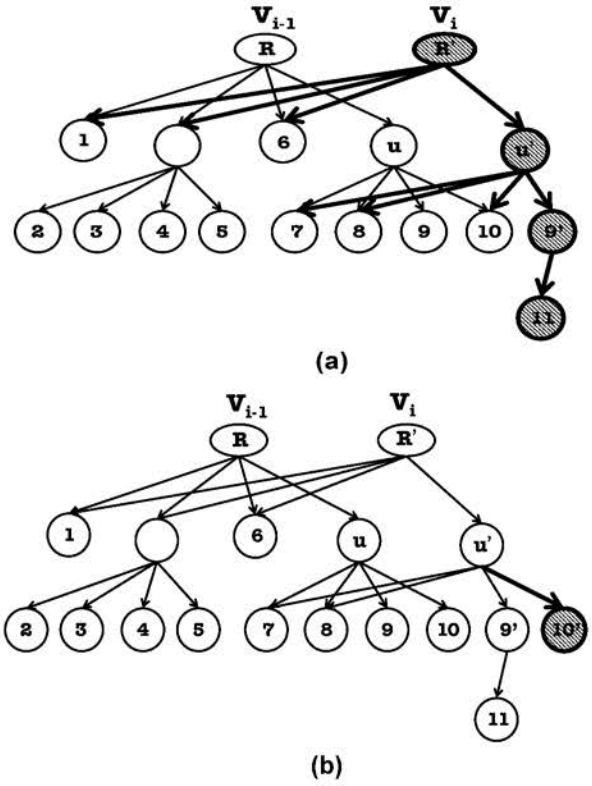


Figure 4: (a) The octree after inserting octant 11; (b) The octree after updating octant 10. Note: the dark octants are newly inserted octants due to the insertion of octant 11 or updating of octant 10.

with their latest values of octant u. Then octant 9′ is linked to octant u'. While the updates being propagated to the root octant R, a new root node R' will be created for V_i. We will also set up pointers to parents if needed.

For updating operations, only octants of C_0 and the octants of C_1 but not shared with V_{i-1} can be updated in place. To update an octant shared by V_i and V_{i-1}, we need to first create a copy of the octant and then update it, as illustrated in Figure 4(b).

Both insertion and updating use a copy-on-write approach to recursively create a new version of an octree. In practice, these operations can be quite efficient for three reasons. First, C_1 only serves a small ratio of writes on NVBM during meshing due to dynamic transformation of PM-octree layout (Section 3.3). Second, the insertion/updating operations are executed in batch and infrequently. Therefore, the overhead of copying octants can be amortized. Third, not as the design of other multi-version persistent data structures [49], we only need to track two versions of the octants. Therefore, copying overhead is significantly reduced.

Deletion of an octant: An octant cannot be removed directly from V_{i-1}. Instead, a copy of its parent octant will be created with an empty pointer to replace the pointer which linked to the removed octant. We can directly delete an octant in C_0 as it resides in DRAM. However, if an octant needs to be removed from C_1 in NVBM, we

will only mark the octant as "deleted". It does not require deleting any octants because deletion can cause a long write latency just as an insertion for NVBM. The real deletion is only handled by garbage collection (GC). Before GC is executed, the allocated NVBM regions will be not released and can be reused for inserting new octants. This optimization will reduce the number of writes to NVBM and improve its efficiency.

Garbage collection: As the size of C_1 does not decrease when octants are deleted and can increase due to a creation of its parent octants, GC is required to be executed before the execution of a new time step of a simulation. To ensure NVBM efficiency, we track the percentage of available NVBM space. When it is smaller than a threshold $threshold_{NVBM}$, GC will also be executed on demand. We use a mark-and-sweep garbage collector [9] for its implementation. Specifically, the GC routine starts from the child octants of the root node of C_1 and deletes octants which are marked as "deleted" and unreferenced the octants by invalidating the parent pointers to the deleted octants.

Traversing a PM-octree: To traverse a PM-octree, we need to first identify the root nodes of the tree V_{i-1} and V_i. They are stored in two known memory addresses, $ADDR(V_{i-1})$ and $ADDR(V_i)$. As mentioned earlier, V_{i-1} is used to restart an application and V_i is accessed during its normal execution. Both of the trees can be traversed as regular octrees. Therefore, all existing in-core algorithms for tree construction, mesh refinement and coarsening using the tree, tree balancing, and mesh extraction can be easily adapted to the new system with few changes. The major difference lies in the implementation of routines for checkpointing and failure recovery. We will discuss them in detail in Section 3.4.

Merging of PM-octree components: The merging routine is executed in two scenarios. (1) The size of C_0 is large so that the remaining DRAM space is smaller than $threshold_{DRAM}$. To maintain efficiency, a subtree of C_0 is trimmed and merged out with C_1 by inserting new octants or updating existing octants of C_1. (2) Simulation of time step i is completed and a persistent point of the octree needs to be saved. In this scenario, we need to first merge C_0 with C_1 and then mark all the nodes existing only on V_{i-1} as "deleted". The GC routines executed later will delete those octants. In the end, we need to swap $ADDR(V_i)$ and $ADDR(V_{i-1})$. Only after this point, the new persistent data structure becomes available and simulation of time step $i+1$ can begin. If the applications fail during merging, the octants marked as "deleted" should be recovered and used to restart the program. To ensure data consistency, GC is disabled during the merging of PM-octree.

3.3 Dynamic Transformation of PM-octree

Access to data in NVBM may incur high read/write latencies. To achieve optimal performance, we need to use DRAM to store frequently accessed (hot) subtrees of PM-octree and use NVBM to store less frequently accessed (cold) ones. Figure 5 shows a simplified octree which resides in both DRAM and NVBM. We use this example to demonstrate the importance of dynamic transformation of PM-octree layout. The subdomains denoted by octants 2-5 are accessed much more frequently than octants 7-10 at time step i. They are hot octants, while octants 1 and 6 are cold octants. However, with a brute-force approach without considering data access pattern,

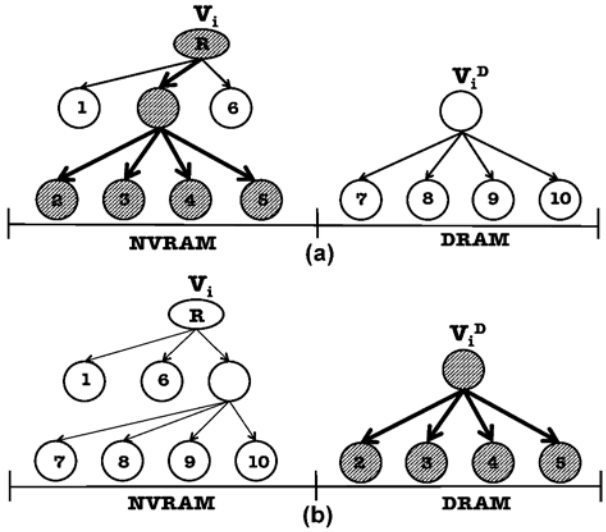


Figure 5: (a) Locality oblivious layout; (b) Locality aware layout. Note: the dark octants denote subdomains being actively accessed.

octants 2-5 might be stored in NVBM as shown in Figure 5(a). This suboptimal layout can result in 89% more writes being served in NVBM for a simple refinement operation (e.g., those used in droplet ejection simulation) than the optimal layout shown in Figure 5(b).

To alter the layout of PM-octree, we need a mechanism to determine when a transformation should be executed. However, we cannot simply use the history of octant accesses to predict future access pattern because a simulation with adaptive mesh refinement may compute on two different domains in two consecutive time steps. We propose a *feature-directed sampling technique* for this purpose. The application features are application-level knowledge and realized as functions for octant refinement/coarsening or solver functions traversing an octree in computations. These functions are provided as inputs to PM-octree library for identification of hot octants. We consider the additional programming burden imposed on developers are trivial because those functions already exist and the library supporting PM-octree simply pre-executes them for generating application-level hints on data access patterns (e.g., access frequency of subtrees) of major meshing routines.

A transformation should be executed only when the access frequency of subtree i in NVBM ($Freq_i^{NVBM}$) is significantly larger than the access frequency of subtree j in DRAM ($Freq_j^{DRAM}$). We use the following steps to predict the access frequency of a subtree. (1) Randomly select N_{sample} octants in a subtree. By default, N_{sample} is set to min(100, $Size_{octant}$), where $Size_{octant}$ is the total number of octants in a subtree. (2) Pre-execute feature functions based on the locational code or value of data associated to the octants. The feature functions return 1 if the corresponding domain is of interest to the application (e.g., the refining condition is satisfied) or 0 if not; (3) The access frequency of the subtree is computed as the total number of 1s returned by the feature functions. Then we compute the ratio of access ($Ratio_{access}$) as the ratio of $Freq_i^{NVBM}$ and $Freq_j^{DRAM}$. (4) Check whether $Ratio_{access}$ is larger than $T_{transform}$, where $T_{transform}$

Routine	*Description*
pmoctree ⋆ pm_create(octree ⋆ tree)	create a new PM-octree; return a pointer to V_i
void pm_persistent(pmoctree ⋆ tree)	create a persistent version of octree
pmoctree ⋆ pm_restore(void)	restore a PM-octree; return a pointer to V_i
void pm_delete(pmoctree ⋆ tree)	delete all octants on NVBM and DRAM

Table 1: The program interface of PM-octree.

is a predefined threshold and its default value is set empirically in our prototype system. If it is true, then it indicates that the data access pattern shall be changed and it is necessary to re-layout PM-octree.

To reduce the overhead of sampling, we only carry out the transformation detection on subtrees, whose size (number of octants) is similar to that of the C_0 tree in DRAM. In the paper, we define *subtree level* as the level of the root node of a subtree in PM-octree. The maximum level of a subtree (L_{sub}) is constrained by the size of DRAM available for C_0. Therefore, L_{sub} can be determined using Equation 1.

$$L_{sub} = Depth_{octree} - \left\lfloor \log_{Fanout} Size_{DRAM} \right\rfloor \quad (1)$$

In the equation, $Depth_{octree}$ is the current depth of PM-octree, *Fanout* is the number of children a node has (8 for octree), and $Size_{DRAM}$ is the size of *DRAM* configured for C_0. A larger $Size_{DRAM}$ indicates that we can select and move a subtree with more octants. To change the layout of PM-octree, we need to swap the subtree having the maximum $Ratio_{access}$ in NVBM with C_0 in DRAM. To further reduce the overhead of layout transformation, octants are *copied on write* and *incrementally*. Dynamic transformation is only triggered after the completion of the merging operations.

3.4 Program Interface and Failure Recovery

We provide orthogonal persistence [7, 19] in the design of PM-octree so that users do not need to control what octants become persistent on NVBM. They can be persistent and restored simply by calling the program interface (e.g., *pmoctree_persistent()* and *pmoctree_restore()*) described in Table 1, just as how they use legacy APIs of creating snapshots on block devices, e.g., file systems. As an example, we can replace gfs_output_write() and gfs_output_read() used in Gerris with pm_persistent() and pm_restore(). Essentially, we implement a *persistent memory abstraction* for octrees so that developers can be freed from the tedious and error-prone task of NVBM allocation and pointer management at failure recovery.

Creating persistent versions of octree and failure recovery using these routines are fast because they operate at memory bandwidth speeds. In the scenario that the crashed compute nodes are available after rebooting and can be used to serve the failed applications, instantaneous failure recovery can be achieved because PM-octree simply marks the octants only in V_i as "deleted" and returns the address of V_{i-1} for a simulation to access a consistent and persistent octree on NVBM. The "deleted" octants are recycled by GC asynchronously in the background.

In the scenario that the crashed node will not be available for serving the same applications, PM-octree supports storing remote replicas for higher data reliability and availability. The feature must be turned on/off by users in the prototype of PM-octree. We wish to leave the automated approach for remote replica scheduling as the future work. Currently, when the feature is enabled by users, PM-octree creates two copies of V^H_{i-1} and V^P_{i-1}. V^H_{i-1} is stored on the host node where processes are scheduled to run. And V^P_{i-1} is stored on other compute nodes or staging nodes selected by job schedulers according to their NVBM utilization. PM-octree only stores the differences of V^P_{i-1} and V^P_i rather than complete octrees. Because of the high overlapping ratio of octrees between adjacent time steps in the simulations studied in this paper, the differences are small and therefore the octant transmitting overhead is small for maintaining the consistency of octant replicas. In this case, PM-octree can still achieve near-instantaneous failure recovery as shown in Section 5.6.

Using the operations of PM-octree described in Section 3, we develop a library to implement these routines as follows.

- **pm_create**: It starts from root octant V_i that encloses the entire domain and then expands PM-octree by inserting octants. V_i is further partitioned to C_0 and C_1.
- **pm_persistent**: It merges V_i with V_{i-1} and then executes a dynamic transformation of the layout of PM-octree data structure. When the crashed node will not be available, delta octants need to be copied to other compute nodes.
- **pm_restore**: It ensures that PM-octree is identical to the most recent consistent version V_{i-1}. To achieve this, it marks "deleted" for the octants referenced by V_i but not referenced by V_{i-1} and then returns ADDR(V_{i-1}). When the crashed node will not be available, V_{i-1} can be recovered from its replicas stored on other compute nodes.
- **pm_delete**: It deletes all the octants and enables GC to free memory spaces.

4 IMPLEMENTATION

We implement PM-octree and its auxiliary library in Gerris [4, 5] flow solver to evaluate its correctness, performance, and scalability. Gerris is developed for simulations of surface-tension dominant multiphase flows. It is one of the most widely used open-source Computational Fluid Dynamics (CFD) software. The code has been extensively adopted and validated for a large variety of multiphase flows. Gerris uses the Cartesian mesh based finite volume method to solve the governing equations of multiphase flows. It uses the MPI-based domain decomposition method to achieve highly scalable parallelism with load-balancing capability.

PM-octree is implemented with 4K lines of code in C++. It has two major components, realizing the operations of PM-octree and persistence management using NVBM. Specifically, the operations of PM-octree include octant insertion/updating, garbage collection, traversing, merging, and dynamic layout transformation. During simulation, these operations are then used by the internal functions of Gerris such as *ftt_cell_traverse()*, *ftt_cell_neighbor()*, *ftt_cell_refine()*, *ftt_cell_write()*, and *ftt_cell_read()*. For snapshot

management, Gerris uses POSIX I/O library in its current functions, such as *gfs_simulation_read()*, *gfs_output_file_new()*, *gfs_output_file_open()*, and *gfs_output_file_close()*. We implement another set of functions as described in Table 1 in Gerris. When NVBM is configured for simulation, these functions are called to replace the existing snapshot functions.

5 EVALUATION

We conduct an extensive performance study for PM-octree to experimentally answer the following questions.

- What is the time distribution among major routines while generating a mesh using PM-octree?
- Is PM-octree effective and scalable for real scientific workloads with diverse data access patterns?
- What are the performance implications for PM-octree when parameters such as the size of DRAM configured for the C_0 tree is changed?
- What is the impact of dynamic transformation of PM-octree layout on the reduction of NVBM writes?

5.1 Experimental Setup

	DRAM	*NVBM*
Read Latency(*ns*)	60	**100**
Write Latency(*ns*)	60	**150**
Endurance(writes/bit)	$> 10^{16}$	$10^6 - 10^8$

Table 2: Characteristics of DRAM and NVBM. Note: The table contents are based mainly on [29, 39, 50].

Modeling DRAM-fast NVBM We model NVBM using DRAM on real computer servers using an emulation based approach, which is similar to those in other projects [22, 33, 52, 53]. Specifically, our NVBM emulator introduces extra latency for NVBM write and read through routines that write to or read from NVBM. The delay is determined according to the performance parameters listed in Table 2. We create delays using a software spin loop [33, 52] that uses the x86 RDTSP instruction to read the processor timestamp counter and spins until the counter reaches the intended delay.

We evaluated the performance of PM-octree on the Titan supercomputer at Oak Ridge National Laboratory. Titan consists of 18,688 nodes, each of which is configured with a 16-core AMD Opteron 6274 CPU and 32GB memory. Each node runs Cray Linux Environment operating system. All nodes were interconnected with a Gemini network.

Driving Scientific Problem: To evaluate the correctness and scalability of PM-octree based adaptive meshing algorithms, we develop a program to simulate *droplet ejection* [21, 43], as the driving scientific workload. It uses the Cartesian mesh based finite volume method to simulate the flow. The simulation can accurately predict droplet breakups, which happen after the liquid jet pinches off at nozzle as shown in Figure 1(c). The jet eventually breaks up into multiple droplets due to capillary instability. The accurate prediction of droplet breakups in inkjet printing is extremely challenging, because the solution of singularity at breakup location requires the mesh resolution under micrometer, whereas the inkjet device is often at centimeter scale. So there is at least four orders-of-magnitude length difference across the fluid domains.

The simulation consists of multiple time steps. In the experiments, we run the simulation for 100 time steps. 8GB DRAM is configured to stored the octants of the C_0 tree unless otherwise stated. It is equivalent to set the $threshold_{DRAM}$ to 24GB. We generate meshes of various sizes through adjusting the number of mesh elements.

Octree Implementations: The simulation of the droplet ejection is implemented using three octree implementations. (1) *In-core-octree*: it is the existing octree data structure used in the meshing algorithm in Gerris. During its simulation all octants are stored in DRAM. The octree data structure is saved as a snapshot file stored on NVBM every 10 time steps and accessed via file-system interface. (2) *Out-of-core-octree*: it is a data structure used in the Etree library [44] for meshing using slow storage devices, e.g., hard disks. Specifically, all octants are stored on disks and data indexes are created to reduce I/O latency of accessing the octants. We modified the Etree library to support parallel meshing over octants across thousands of compute nodes. Furthermore, we use NVBM instead of disks to study the performance of Etree in a computing environment configured with NVBM. The Etree octants are stored in NVBM and accessed via file-system interface. (3) *PM-octree*: it is the data structure described in Section 3 with the dynamic transformation of PM-octree layouts during execution to reduce NVBM writes by exploring the locality of octree access.

5.2 Weak Scaling

We first study weak scaling of the simulation with three octree implementations, in-core-octree, PM-octree, and out-of-core-octree. We measure their execution times [3] while increasing the problem size from 1.2M elements to 1077M elements and the number of processors from 1 to 1000. The number of elements per processor is $\sim$1M. Because the program uses unstructured meshes, it is impossible to guarantee the number of elements per processor to be exactly the same over different runs. However, we found that the different is no more than 7% across the runs.

Figure 6 shows the execution time with PM-octree, compared to in-core-octree and out-of-core-octree respectively. We have three observations from the results. (1) None of the implementations can achieve an optimal speedup because of the communication overhead and tree partitioning overhead. As shown in Figure 7, the time of *Partition* is 0s on a single processor with 1.2M elements. However, it is increased to 13sec (19%) when the mesh size is 7.1M elements on 6 processors, compared to 1409sec (56%) when the mesh size is increased to 1077M elements on 1000 processors. (2) PM-octree achieves similar weak scaling as in-core-octree. The times on refining and balancing increase as a logarithm of problem size. For example, as the problem size is increased from 1.2M to 1077M (897X), the time on refining octants only increased 16X. It suggests that PM-octree has good scalability. (3) Out-of-core-octree can only achieve a suboptimal performance because of the overhead from the Etree library and file system software. For example, the Etree library uses B-tree for data indexing on disks. However, it may cause

[3] Mesh extracting time is excluded from the execution time because *Extract* is executed offline and on demand for our research.

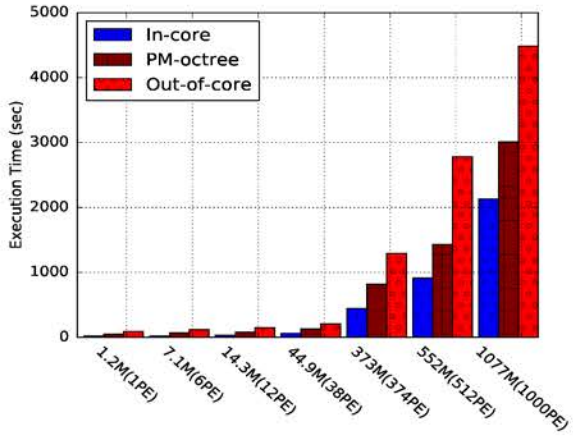


Figure 6: Execution time of the simulation as the problem size increases from 1.2M to 1077M elements along with the number of processors increases from 1 to 1000. We compare the execution time with PM-octree to that with in-core-octree and out-of-core-octree respectively.

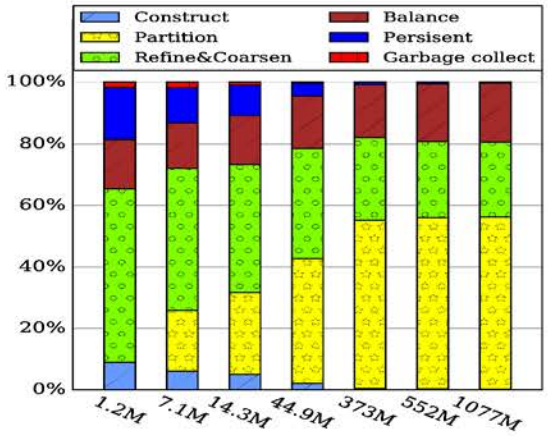
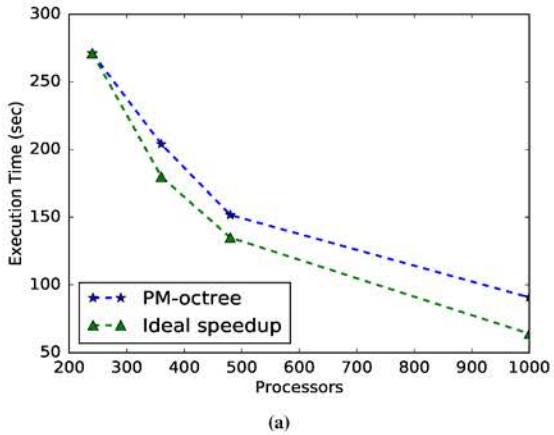
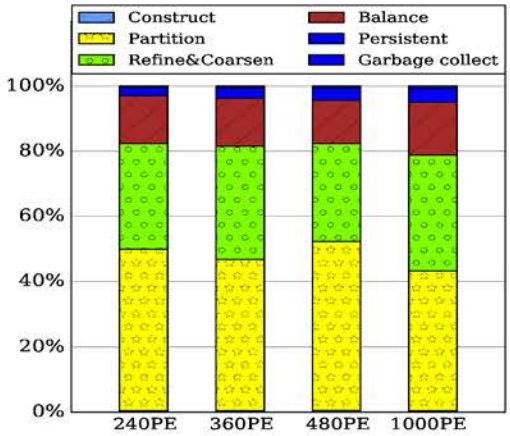


Figure 7: Execution time percentage breakdown across major simulation routines for weak scaling study.



(a)



(b)

Figure 8: (a) Execution time of the simulation with PM-octree, compared to that with ideal speedup. (b) Execution time percentage breakdown across major simulation routines for strong scaling study.

extra memory latency if the B-tree is used for indexing data stored in NVBM.

5.3 Strong Scaling

To study strong scaling of the simulation with PM-octree, we compute the speedup when the problem size is kept constant and the number of processors is increased. Specifically, we fix the problem size to be 150M elements while increasing the number of processors from 240 to 1000. The execution time is shown in Figure 8(a). A finding from this figure is that the execution time speedup with PM-octree is close to the ideal speedup on 240-1000 processors. The major overhead in a large scale is caused by tree partitioning. However, no major fluctuation is observed in the execution breakdowns as shown in Figure 8(b) as we increase the number of processors. It indicates that program has no scalability issue.

In addition, we compared the execution time with PM-octree to that with in-core-octree and out-of-core-octree respectively. Figure 9 shows the results. We found that (1) the execution times of the three schemes are linearly decreased while the number of processors is increased from 240 to 1000 because fewer mesh domains need to be computed on a processor; (2) The performance gap between in-core-octree and PM-octree is reduced. For example, when the number of processors is 240, the speedup of using in-core-octree is 48%. It is reduced to 36% when the number of processors is increased to 1000. This is because, with fewer octants assigned to each processor, more octants can be stored in the C_0 tree in DRAM. Then it is more likely that the simulation with PM-octree could achieve the performance as that with in-core-octree.

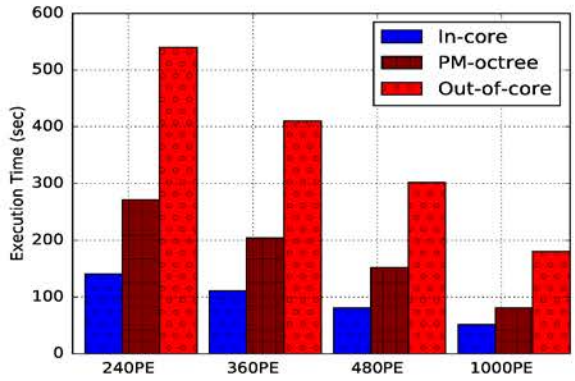


Figure 9: Execution time of the simulation with fixed problem size (150M elements) and varying number of processors from 240 to 1000. We compare the execution time with PM-octree to that with in-core-octree and out-of-core-octree respectively.

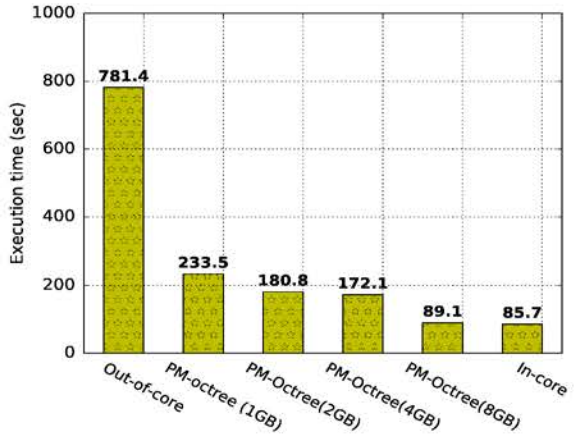


Figure 10: Execution time of the simulation with out-of-core-octree, PM-octree with different DRAM sizes (1GB, 2GB, 4GB, and 8GB) configured for its C_0 tree, and in-core-octree.

5.4 Impact of DRAM Size

Memory size is inversely correlated to the amount of octants stored in NVBM. We can vary the memory size to influence the merging frequency of the C_0 tree in DRAM and the C_1 tree in NVBM and then correspondingly affect the execution time of the simulation with PM-octree. In the following experiments, we ran the simulation with different DRAM sizes to determine whether the program's execution time is substantially affected by the DRAM size. The generated mesh has 6.75M elements and is generated on 100 processors. The maximum DRAM demand is 20GB for simulation with in-core-octree.

We increase the size of DRAM configured for the C_0 tree of PM-octree from 1GB, 2GB, 4GB, to 8GB. As shown in Figure 10, the execution time of the simulation is decreased from 233.5sec to 89.1sec when the DRAM for the C_0 tree is increased from 1GB to 8GB. When DRAM size is small, the C_0 tree needs to be frequently merged with the C_1 tree. Specifically, C_0 and C_1 were merged 491 times when DRAM size is 1GB. However, as the DRAM size increases, more octants can be stored in C_0 in DRAM, resulting in less merging operations and fewer NVBM accesses. When the DRAM size is 8GB, the C_0 tree only needs to be merged with the C_1 tree at the end of each time step for data persistence.

From Figure 10 we have two more observations. (1) When DRAM is large enough to store the V_i tree the execution time with PM-octree is very close to that with in-core-octree. This is because of the snapshot overhead. The simulation with in-core-octree needs to write all octants to NVBM for persistence, while PM-octree only needs to write new and updated octants with much fewer writes to NVBM. (2) When the DRAM size is 1GB, the performance with PM-octree is still significantly better than that with out-of-core-octree. There are three reasons. First, the octants of out-of-core-octree are not byte-addressable. To improve the I/O performance, its minimum I/O unit is a page (4KB), which consists of multiple octants. Second, it takes multiple memory access to search the data index to find a page where an octant is located. Third, out-of-core-octree is designed as a linear octree. Its balancing operation is very time-consuming because no pointers to neighbor octants are maintained in its data structure. Therefore, for a single octant, it needs to search all its 26 neighbors, resulting in very high I/O overhead when balancing an octree of millions of octants.

5.5 Effectiveness of Dynamic Transformation

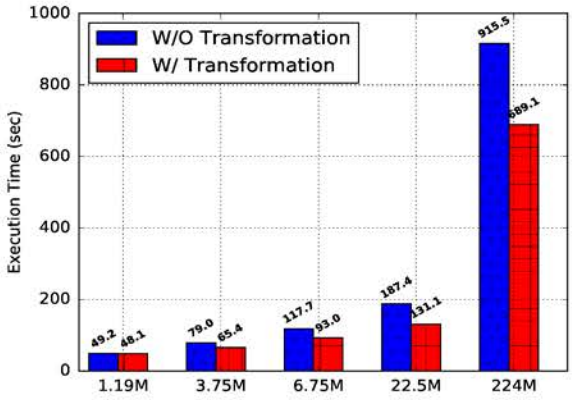


Figure 11: Execution time without and with the dynamic transformation of a PM-octree layout.

The dynamic transformation of a PM-octree layout can reduce the number of writes to NVBM by moving its frequently accessed (hot) subtrees from NVBM to DRAM, therefore saving the execution time of a simulation. The hot subtrees are identified using the feature-directed sampling approach described in Section 3.3. We study the impact of the dynamic transformation on the effectiveness of PM-octree in this section. We ran the simulation with varying number of mesh elements from 1.19M, 3.75M, 6.75M, 22.5M, to 224M. The meshes are generated on 100 processors. Figure 11 shows the execution time of the simulation without and with the dynamic transformation.

When the mesh size is small, the octants which are frequently accessed in the hot subtrees can be stored in DRAM. Therefore, the execution time (48.1sec) with the dynamic transformation scheme is almost the same as that without it. When we increase the mesh size to 224M elements, the dynamic transformation can reduce the execution time by 24.7% from 915.5sec to 689.1sec and the number of writes to NVBM by 31%. The reason is that the C_0 tree in DRAM can store only 7% of the total number of octants. When the simulation needs to refine octants stored in NVBM, its higher read/write latency lead to the performance degradation. When the dynamic transformation is enabled, it uses the features (e.g. locational predicates for mesh refinement) to determine the hot subtrees and then replace the less frequently subtrees in DRAM with the hot ones. The octant copy overhead is well amortized by the benefits of reducing writes to NVBM. As a result, the dynamic transformation can generate a more NVBM-friendly layout and therefore effectively reduce the execution time of the simulation and extend the lifetime of NVBM.

5.6 Failure Recovery

In this section, we compare the time to restart the simulation after failures with in-core-octree, PM-octree, and out-of-core-octree. We use a local Kamiak cluster [2] hosted at Washington State University for the experiments because we need to control when and where failures occur. Kamiak includes 87 20-core Intel Xeon E5-2660 and E5-2680 processors running Redhat Linux of kernel-3.10.0. All nodes are interconnected with a 56 GB/s InfiniBand network. In the experiments, we fix the problem size to be 6.75M elements and run the simulation with 100 processes. We kill the processes at the time step 20 and then restart the simulation. We measure the restarting time in two scenarios: (1) the same set of compute nodes are used to serve the simulation; and (2) a new compute node is used to serve the simulation, replacing one of the crashed nodes.

In the first scenario, with out-of-core-octree, the program can immediately access octants in NVBM because Etree is essentially an octant database and can guarantee data consistency after failures. The time to restart the simulation is 42.9sec with in-core-octree and 2.1sec with PM-octree respectively. PM-octree only needs to mark the octants in V_i as "deleted" and returns V_{i-1} for the simulation to access the consistent octree. In contrast, the program with in-core-octree needs to read all the data from the snapshot file stored in the NVBM based file system.

In the second scenario, the out-of-core-octree cannot recover from failures because the octants stored in the Etree database are not replicated. The time with in-core-octree is the same as that in the first scenario because snapshot files are read from a shared parallel file system, which does not suffer from failures of compute nodes. The time with PM-octree is 3.48sec because it needs additional 1.38sec for moving the octant replica to the new compute node when the copy on one of the compute nodes is no longer available after the crash.

6 CONCLUSIONS

In this paper, we describe the design and implementation of PM-octree data structure for large-scale simulations using octrees. It not only can effectively extend memory capacity using NVBM but also support near-instantaneous failure recovery through exploring non-volatility of NVBM. Furthermore, to make the algorithms truly effectively, the layout of PM-octree is dynamically transformed according to the octant access frequency obtained using a feature-directed sampling approach. Last but not least, it supports orthogonal persistence for applications and provides an easy-to-program interface, with which users are freed from error-prone and tedious tasks of persistent pointer management. A software prototype of PM-octree is developed and integrated with Gerris. And one real-world flow simulation program is developed using PM-octree to simulate droplet ejection in inkjet printing. Our experimental results show that simulations implemented using PM-octree have good scalability up to 1.1 billion elements on 1000 processors on the Titan supercomputer.

As future work, we plan to automate the setting of DRAM size for the C_0 tree in order to provide better memory efficiency under high concurrency and test PM-octree with other flow solvers and simulations requiring adaptive mesh refinement using the octree data structure.

ACKNOWLEDGMENTS

We would like to thank our shepherd Tom Peterka as well as anonymous reviewers for their helpful comments and feedback. This research was supported in part by NSF ACI-1565338, WSUV Research Grant and WSU New Faculty Seed Grant.

REFERENCES

[1] Gerris Flow Solver. http://gfs.sourceforge.net/wiki/index.php/Main_Page.
[2] The Kamiak Cluster at WSU. https://hpc.wsu.edu.
[3] pmem.io: Persistent Memory Programming. http://pmem.io/nvml/.
[4] 2003. Gerris: a tree-based adaptive solver for the incompressible Euler equations in complex geometries. *J. Comput. Phys.* 190, 2 (2003), 572 – 600.
[5] 2009. An accurate adaptive solver for surface-tension-driven interfacial flows. *J. Comput. Phys.* 228, 16 (2009), 5838 – 5866.
[6] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, Alexander Driskill-Smith, and Mohamad Krounbi. 2013. Spin-transfer Torque Magnetic Random Access Memory (STT-MRAM). *J. Emerg. Technol. Comput. Syst.* 9, 2 (May 2013).
[7] Malcolm P. Atkinson and O. Peter Buneman. 1987. Types and Persistence in Database Programming Languages. *ACM Comput. Surv.* 19, 2 (June 1987).
[8] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. 2009. PLFS: A Checkpoint Filesystem for Parallel Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*.
[9] Hans-Juergen Boehm and Mark Weiser. 1988. Garbage Collection in an Uncooperative Environment. *Softw. Pract. Exper.* 18, 9 (Sept. 1988).
[10] Geoffrey W Burr, Matthew J Breitwisch, Michele Franceschini, Davide Garetto, Kailash Gopalakrishnan, Bryan Jackson, Bülent Kurdi, Chung Lam, Luis A Lastras, Alvaro Padilla, and others. 2010. Phase change memory technology. *Journal of Vacuum Science & Technology B* 28, 2 (2010), 223–262.
[11] Adrian M. Caulfield, Joel Coburn, Todor Mollov, Arup De, Ameen Akel, Jiahua He, Arun Jagatheesan, Rajesh K. Gupta, Allan Snavely, and Steven Swanson. 2010. Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*.
[12] Shimin Chen and Qin Jin. 2015. Persistent B+-trees in Non-volatile Main Memory. *Proc. VLDB Endow.* 8, 7 (Feb. 2015).
[13] Joel Coburn, Adrian Caulfield, Laura Grupp, Ameen Akel, and Steven J Swanson. 2009. NVTM: A transactional interface for next-generation non-volatile memories. (2009).
[14] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. *SIGPLAN Not.* 46, 3 (March 2011).
[15] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. 133–146.

[16] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. 1989. Making Data Structures Persistent. *J. Comput. Syst. Sci.* 38, 1 (Feb. 1989).

[17] B. V. Essen, H. Hsieh, S. Ames, and M. Gokhale. 2012. DI-MMAP: A High Performance Memory-Map Runtime for Data-Intensive Applications. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. 731–735.

[18] C. Faloutsos and S. Roseman. 1989. Fractals for Secondary Key Retrieval. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '89)*.

[19] Jorge Guerra, Leonardo Marmol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. 2012. Software Persistent Memory. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*.

[20] R. Hagmann. 1987. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles (SOSP '87)*.

[21] Stephen D. Hoath. 2016. *Fundamentals of inkjet printing: the science of inkjet and droplets*. Wiley-VCH Verlag GmbH &Co.

[22] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. 2014. NVRAM-aware Logging in Transaction Systems. *Proc. VLDB Endow.* 8, 4 (Dec. 2014).

[23] Tong Jin, Fan Zhang, Qian Sun, Hoang Bui, Manish Parashar, Hongfeng Yu, Scott Klasky, Norbert Podhorszki, and Hasan Abbasi. 2013. Using Cross-layer Adaptations for Dynamic Data Management in Large Scale Coupled Scientific Workflows. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*.

[24] C. Josserand and S.T. Thoroddsen. 2016. Drop Impact on a Solid Surface. *Annual Review of Fluid Mechanics* 48, 1 (2016), 365–391.

[25] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic. 2013. Optimizing Checkpoints Using NVM as Virtual Memory. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. 29–40.

[26] E. Kim, J. Bielak, O. Ghattas, and J. Wang. 2002. Octree-based finite element method for large-scale earthquake ground motion modeling in heterogeneous basins. *AGU Fall Meeting Abstracts* (Dec. 2002).

[27] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. 2016. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*.

[28] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*.

[29] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory As a Scalable Dram Alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*.

[30] Dinesh P. Mehta and Sartaj Sahni. 2004. *Handbook of Algorithms and Data Structures*. Chapman and Hall/CRC.

[31] C. Mohan. 1999. Repeating History Beyond ARIES. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*.

[32] Dushyanth Narayanan and Orion Hodson. 2012. Whole-system Persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*.

[33] Jiaxin Ou, Jiwu Shu, and Youyou Lu. 2016. A High Performance File System for Non-volatile Main Memory. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*.

[34] Carey P.V. 2008. *Liquid-vapor phase-change phenomena*. CRC press, New York.

[35] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*.

[36] V. A. Sartakov and R. Kapitza. 2014. NV-Hypervisor: Hypervisor-Based Persistence for Virtual Machines. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*.

[37] David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. 2015. NVC-Hashmap: A Persistent and Concurrent Hashmap For Non-Volatile Memories. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics (IMDM '15)*. Article 4, 4:1–4:8 pages.

[38] Q. Shi and J. JaJa. 2006. Isosurface Extraction and Spatial Filtering using Persistent Octree (POT). *IEEE Transactions on Visualization and Computer Graphics* (Sept 2006), 1283–1290. DOI:https://doi.org/10.1109/TVCG.2006.157

[39] Vinay Nath Shimin Chen, Phillip B. Gibbons. 2011. Rethinking Database Algorithms for Phase Change Memory. In *CIDR'11: 5th Biennial Conference on Innovative Data Systems Research*.

[40] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. 2008. *Operating System Concepts* (8th ed.). Wiley Publishing.

[41] H. Sundar, R. S. Sampath, S. S. Adavani, C. Davatzikos, and G. Biros. 2007. Low-constant parallel algorithms for finite element simulations using linear octrees. In *Supercomputing, 2007. SC '07. Proceedings of the 2007 ACM/IEEE Conference on*. 1–12.

[42] Hari Sundar, Rahul S Sampath, and George Biros. 2008. Bottom-up construction and 2: 1 balance refinement of linear octrees in parallel. *SIAM Journal on Scientific Computing* 30, 5 (2008), 2675–2708.

[43] H. Tan, E. Torniainen, D. P. Markel, and R. N. K. Browning. 2015. Numerical simulation of droplet ejection of thermal inkjet printheads. *International Journal for Numerical Methods in Fluids* 77 (March 2015), 544–570.

[44] T. Tu, J. Lopez, and D. O'Hallaron. 2003. *The Etree Library: A System for Manipulating Large Octrees on Disk*. Technical Report CMU-CS-03-174. Carnegie Mellon School of Computer Science.

[45] Tiankai Tu and D. R. O'Hallaron. 2004. A Computational Database System for Generatinn Unstructured Hexahedral Meshes with Billions of Elements. In *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*. 25–25.

[46] Tiankai Tu, D. R. O'Hallaron, and O. Ghattas. 2005. Scalable Parallel Octree Meshing for TeraScale Applications. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*. 4–4. DOI:https://doi.org/10.1109/SC.2005.61

[47] T. Tu, H. Yu, L. Ramirez-Guzman, J. Bielak, O. Ghattas, K. l. Ma, and D. R. O'Hallaron. 2006. From Mesh Generation to Scientific Visualization: An End-to-End Approach to Parallel Supercomputing. In *SC 2006 Conference, Proceedings of the ACM/IEEE*.

[48] Shyh-Kuang Ueng, C. Sikorski, and Kwan-Liu Ma. 1997. Out-of-core streamline visualization on large unstructured meshes. *IEEE Transactions on Visualization and Computer Graphics* 3, 4 (1997), 370–380.

[49] P. J. Varman and R. M. Verma. 1997. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering* 9, 3 (May 1997), 391–DOI:https://doi.org/10.1109/69.599929

[50] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST'11)*.

[51] J. S. Vetter and S. Mittal. 2015. Opportunities for Nonvolatile Memory Systems in Extreme-Scale High-Performance Computing. *Computing in Science Engineering* 17, 2 (2015), 73–82.

[52] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. 2014. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*.

[53] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. *SIGPLAN Not.* 47, 4 (March 2011), 91–104.

[54] W. X. Wang, Z. Lin, W. M. Tang, W. W. Lee, S. Ethier, J. L. V. Lewandowski, G. Rewoldt, T. S. Hahm, and J. Manickam. 2006. Gyro-Kinetic simulation of global turbulent transport properties in tokamak experiments. *Physics of Plasmas* 13, 9, 092505.

[55] Xingbo Wu, Fan Ni, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, Zili Shao, and Song Jiang. 2016. NVMcached: An NVM-based Key-Value Cache. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '16)*.

[56] Xiaojian Wu and A. L. Narasimha Reddy. 2011. SCMFS: A File System for Storage Class Memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*.

[57] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*.

[58] J. Joshua Yang and R. Stanley Williams. 2013. Memristive Devices in Computing System: Promises and Challenges. *J. Emerg. Technol. Comput. Syst.* 9, 2 (May 2013).

[59] A.L. Yarin. 2006. DROP IMPACT DYNAMICS: Splashing, Spreading, Receding, Bouncing. *Annual Review of Fluid Mechanics* 38, 1 (2006), 159–192.

[60] X. Zhang, K. Davis, and S. Jiang. 2010. IOrchestrator: Improving the Performance of Multi-node I/O Systems via Inter-Server Coordination. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'10)*.

[61] X. Zhang, K. Davis, and S. Jiang. 2011. QoS Support for End Users of I/O-intensive Applications using Shared Storage Systems. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'11)*.

[62] X. Zhang, K. Davis, and S. Jiang. 2012. iTransformer: Using SSD to Improve Disk Scheduling for High-performance I/O. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. 715–726.

[63] X. Zhang, S. Jiang, and K. Davis. 2009. Making resonance a common case: A high-performance implementation of collective I/O on parallel file systems. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. 1–12.

[64] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*.

26 chen ping

27 chen ping

28 chen ping

Large-scale adaptive mesh simulations through non-volatile byte-addressable memory

Nguyen, Bao; Tan, Hua; Zhang, Xuechen

01 chen ping Page 1
27/8/2019 7:47

02 chen ping Page 1
27/8/2019 7:18

03 chen ping Page 1
27/8/2019 7:42

04 chen ping Page 1
27/8/2019 7:49

05 chen ping Page 1
27/8/2019 7:43

06 chen ping Page 1
27/8/2019 7:44

07 chen ping Page 1
27/8/2019 7:52

08 chen ping Page 1
27/8/2019 7:53

09 chen ping Page 2
27/8/2019 7:53

10 chen ping Page 2

27/8/2019 7:54

11 chen ping Page 2

27/8/2019 8:06

12 chen ping Page 2

27/8/2019 8:08

13 chen ping Page 2

27/8/2019 7:57

14 chen ping Page 2

27/8/2019 7:58

15 chen ping Page 2

27/8/2019 7:59

16 chen ping Page 2

27/8/2019 7:59

17 chen ping Page 2

27/8/2019 8:01

18 chen ping Page 2

27/8/2019 8:01

19 chen ping Page 2

27/8/2019 8:26

20 chen ping Page 2

27/8/2019 8:26

21 chen ping Page 2

27/8/2019 8:03

22 chen ping Page 2

27/8/2019 8:04

23 chen ping Page 2

27/8/2019 8:04

24 chen ping Page 2

27/8/2019 8:05

25 chen ping Page 2

27/8/2019 8:05

26 chen ping Page 12

27/8/2019 7:55

27 chen ping Page 12

27/8/2019 7:55

28 chen ping Page 12

27/8/2019 7:55