

# Metal:

## A Persistent Memory Allocator For Data-Centric Analytics



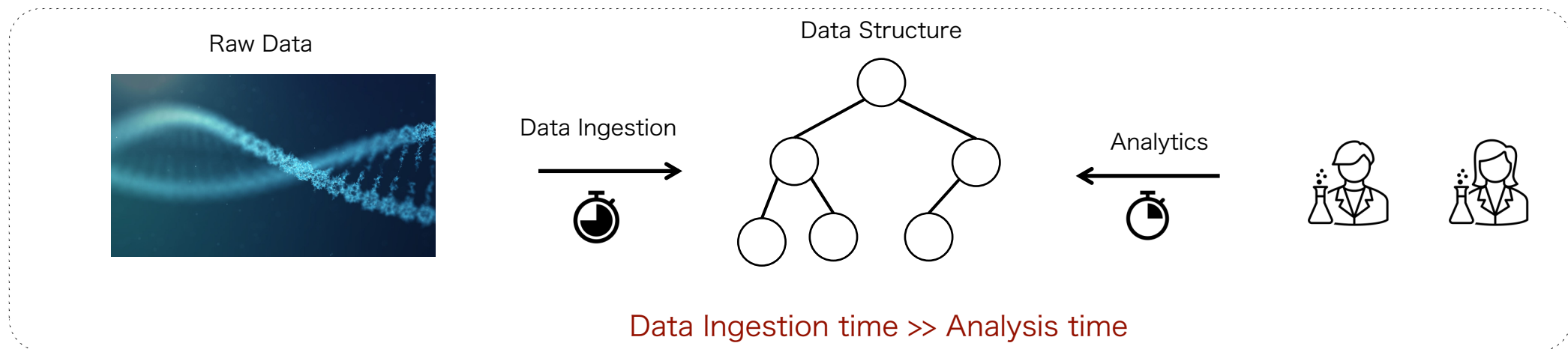
Keita Iwabuchi, Roger Pearce, Maya Gokhale



# Background

## Large-scale Data Analytics

- High volume data analytics is one of the key challenges in exascale
- Data ingestion
  - Indexing and partitioning data with analytics-specific data structures
    - e.g., read raw graph data from text files, and transform into a graph data structure
  - Often more expensive than analytics
  - The same data (or derived data) is re-ingested frequently
    - e.g., run multiple analytics on the same data, changing parameters; develop/debug analytics program

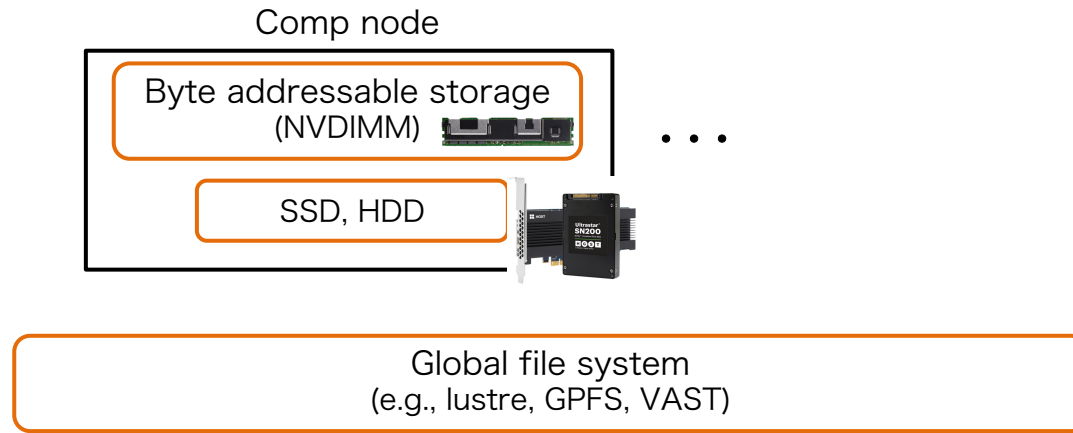




# Background

## Persistent Memory (PM) in HPC

- Substantial performance improvements and cost reductions in non-volatile random-access memory (NVRAM)
- Many HPC systems have various types of NVRAM devices today



### Supercomputers w/ PM

- Sierra
- Summit
- Aurora
- Mammoth
- Fugaku (RIKEN, Japan)
- 
- 
- 

These devices offer cost-effective ways of persistently storing large datasets with efficient means of accessing the data for processing

## Background

# Store Data into PM

- Data serialization is expensive
  - Dismantling and assembling large complex data structures is expensive in terms of performance and programming cost
- Should leverage the file system
  - Tremendous amount of powerful technologies
  - We can support various persistent memory types

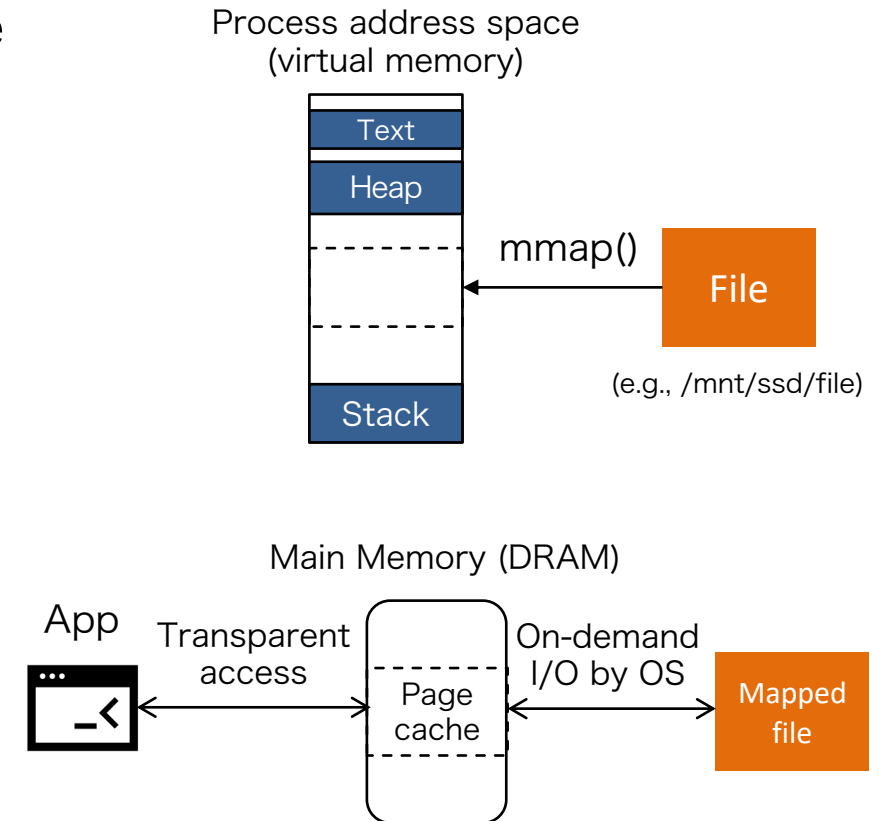
Can we allocate data into file directly and store the data as is while providing transparent access to applications?



# Background

## Memory-mapped File Mechanism (mmap() system call)

- Maps a file into a process's virtual memory (VM) space
- Applications can access mapping area as if it were regular memory
- *Demand paging*
  - OS performs I/O on-demand by *page* granularity (e.g., 4 KB or 64 KB)
  - OS keeps cache in DRAM (*page cache*)
- Can map a file bigger than the DRAM capacity



# Background

## Memory-mapped File Mechanism (mmap() system call)

- Example

```
int fd = open("/mnt/ssd/file", O_RDWR); // Open a file

int size = 1024;
// Maps a file into main memory (1024 bytes)
int* array = (int*)mmap(NULL, size,
                        PROT_READ | PROT_WRITE,
                        MAP_SHARED, fd, 0);

close(fd);

array[0] = 10;

msync(array, size, MS_SYNC); // Flush dirty pages into the file
munmap(array, size); // Close the mapping
```

mmap is powerful; however,  
calling mmap for every memory allocation is expensive

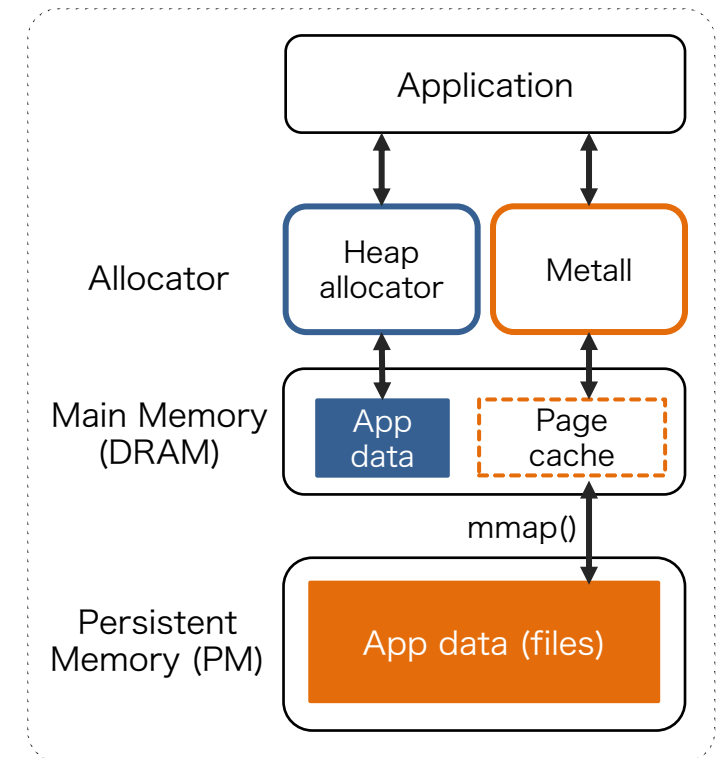


# Metal

## A C++ Allocator for Persistent Memory

[github.com/LLNL/metal]

- A memory allocator built on top of a memory mapping region
  - Designed to work on any devices with file system support (including *tmpfs*)
  - Can leverage file system technologies
- Enables applications to **allocate heap-based objects into PM**, just like main-memory
- Can resume memory allocation work after restarting
- Incorporates state-of-the-art allocation algorithms
  - Some key ideas from SuperMalloc<sup>[Kuszmaul'15]</sup> and jemalloc
- Employs the API developed by Boost.Interprocess (BIP)
  - Useful for **allocating C++ custom data structures in PM**



Fork me on GitHub

# Persistent Memory Allocation using Metall

Create new data (create.cpp)

```
void main () {  
    metall::manager metall_mgr(metall::create_only, "/ssd/test");  
    int* n = metall_mgr.construct<int>("val0")();  
    *n = 10;  
}
```

Annotations for create.cpp:

- Allocate a manager object (points to `metall::manager metall_mgr`)
- Directory to store data (points to `"/ssd/test"`)
- Allocate and construct an object (points to `construct<int>`)
- Store a key to retrieve the data later (points to `"val0"`)
- Metall::manager's destructor synchronizes data with the PM (files) (points to the closing brace `}`)

Terminal

```
$ ./create  
$ ./open  
10
```

Reattach the data (open.cpp)

```
void main () {  
    metall::manager metall_mgr(metall::open_only, "/ssd/test");  
    int* n = metall_mgr.find<int>("val0").first;  
    std::cout << *n << std::endl;  
}
```

Annotation for open.cpp:

- Retrieve data with its key (points to `"val0"`)

Data is directly accessed in PM  
(no serialization overhead)



# Metal with C++ Standard Template Library (STL) Container

A vector type with the STL allocator in Metal  
using `vec_t = vector<int, metall::allocator<int>>`;

Template parameters of the STL vector container

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;
```

Create new data (create.cpp)

```
void main () {
    metall::manager metall_mgr(metall::create_only, "/ssd/test");
    vec_t* pvec = metall_mgr.construct<vec_t>("vec")(metall_mgr.get_allocator());
    pvec->push_back(10); ← Can use it normally, including
                        changing its capacity
}
```

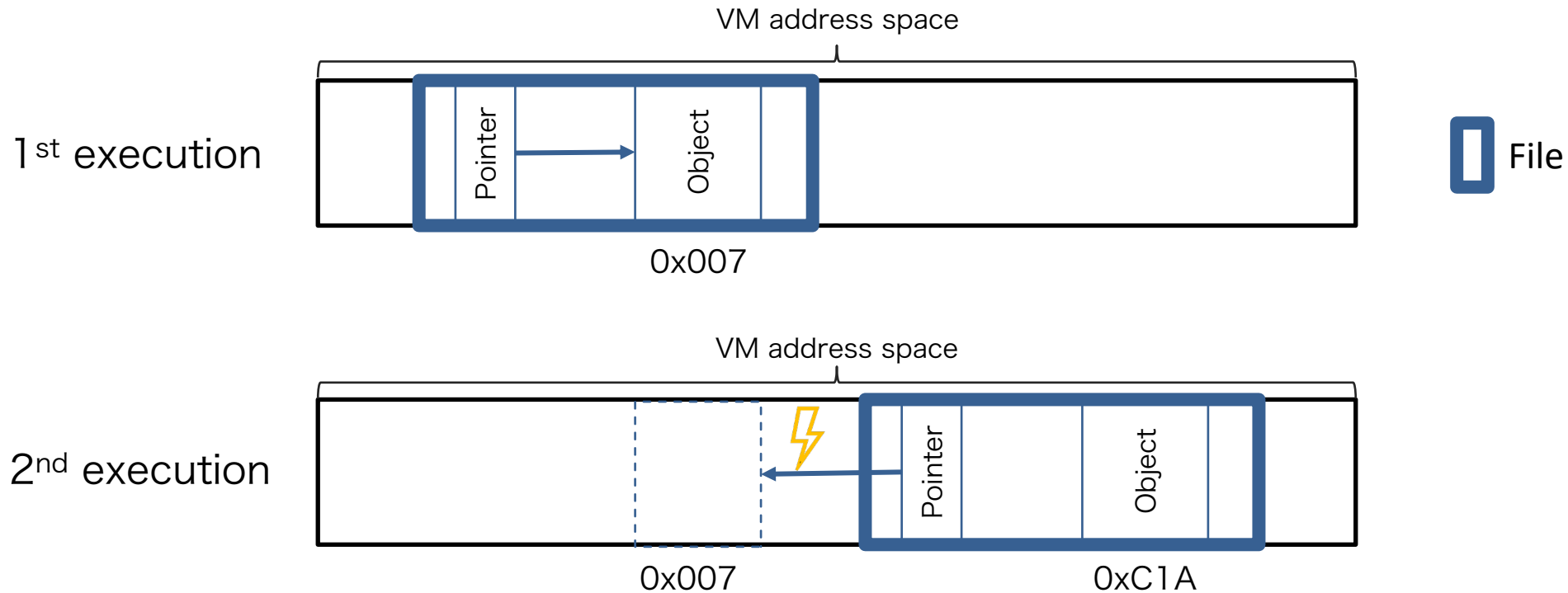
↑  
Arguments to vec\_t's constructor

Reattach the data (open.cpp)

```
void main () {
    metall::manager metall_mgr(metall::open_only, "/ssd/test");
    auto pvec = metall_mgr.find<vec_t>("vec").first;
    pvec->push_back(20); ← Can resume work, including
                        changing its capacity
}
```

Metal follows the C++ standard  
style of using custom allocator  
(no directives, no change to compilers)

# Random Memory Placement



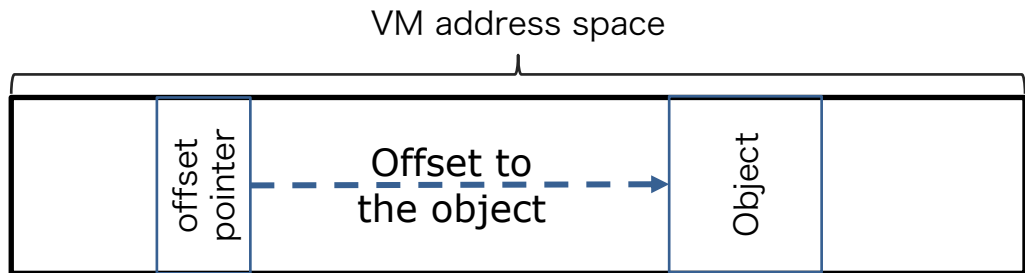
- Metall does not assume that file(s) are mapped to the same VM addresses every time

How to fix the random memory placement issue?



# offset pointer

- An offset pointer holds the offset from itself to the object it points to



Possible implementation

```
template <class T>
struct offset_pointer {
    int64_t offset;
    ... many methods ...
}
```

- Metall inherits offset\_pointer implemented in Boost.Interprocess library (BIP)
- BIP's offset pointer works (almost) transparently with the raw pointer
- The concept of non-raw pointer is being integrated in C++

Usage examples

```
struct data { int n };
data d;
offset_ptr<data> p(&d);
p->n = 10;
p = nullptr;
```

```
int n[2];
offset_ptr<int> p(n);
p[0] = 1;
++p;
--p;
```

# Solutions To Random Memory Placement

- Raw pointer
  - Must be replaced with offset pointers
- Reference, virtual function, and virtual base class
  - Must be removed since raw pointers are used
- STL Container
  - Some implementations do not support offset pointers fully
  - Boost.Container library is compatible with Metall
- Static data members are not supported

# Persistence Policy — fine grained vs coarse grained

- Fine grained persistence policy
  - Synchronizes data with persistent memory after every write operation
  - Ideal for transactional operations with recent byte-addressable PM
  - Can incur an unnecessary overhead for non-transactional apps
- Coarse grained persistence policy
  - Metall employs this policy
  - Synchronizes data only when initiated by application
  - Could cause data inconsistency if there is a crash before synchronizing

# Coarse grained persistence model in Metall

```
metall::manger manager(metall::open, ...); // mmap() → Data is consistent
// Application does some work:
// memory allocations and write operations } Data could be inconsistent
manager.~metall::manager(); // msync() and munmap() → Data is consistent
```

- Consistent mark
  - Metall leaves a mark (file) at the end of its destructor
  - The mark is deleted when Metall opens the data
- Read-only open mode
  - Writing to data allocated in Metall space causes a segmentation fault
    - Metall maps files with the read-only mode in `mmap()`
  - Metall does not delete the consistent mark

# Snapshot/versioning in Metall

- Another way to create a consistent data

```
metall::manger manager(...);  
// Application does some work  
manager.snapshot('/mnt/ssd2/data');  
// Application does some work  
manager.~metall::manager();
```

- 
- calls msync() and copies the mapped files to the '/mnt/ssd2/data'
  - '/mnt/ssd2/data' is **consistent** if snapshot() finishes correctly

# Snapshot/versioning in Metall (cont'd)

- Demonstration of Metall's snapshot

create.cpp

```
metall::manager manager(metall::create_only, "/ssd/dir");

// Allocate an int object and assign 10
auto *n = manager.construct<int>("n");
*n = 10;

manager.snapshot("/ssd/snapshot");

*n = 20;

// Assume that a fatal error happens here
```



open.cpp

```
// 'flag' is false
bool flag = metall::manager::consistent("/ssd/dir");

if (metall::manager::consistent("/ssd/snapshot")) {

    // Open the snapshot
    metall::manager manager(metall::open_only, "/ssd/snapshot");

    int *n = manager.find<int>("n").first; // reattach 'n'
    std::cout << n; // 'n' is 10
}
```

**Lawrence Livermore National Laboratory**  
LLNL-PRES-817002





# Key Design Points

- Focus on relatively large size allocations
- *Virtual memory is cheap in 64-bit machine, physical memory is dear* [Kuszmaul'15]
- Leverage *demand paging* (physical memory is not consumed until accessed)

Simplify implementation & increase speed

# Metall Internal Architecture

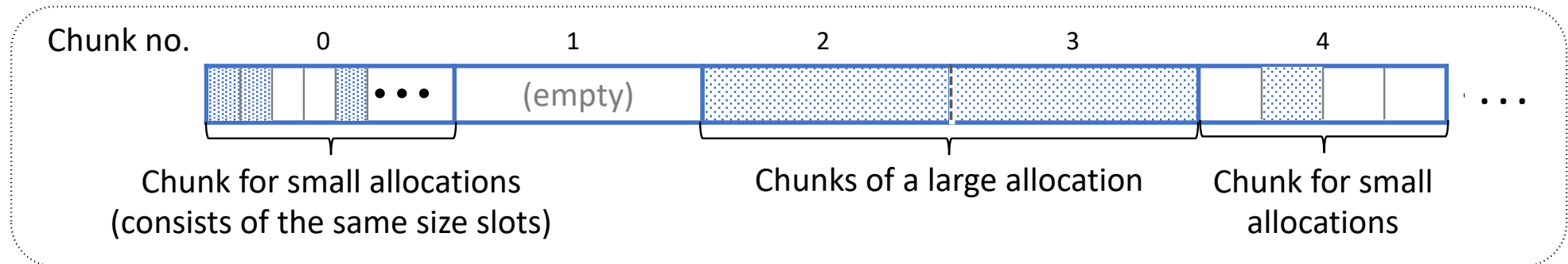
## Internal Allocation Sizes

- Small size category (e.g.,  $\leq 1$  MB)
  - Rounded up to the nearest internal allocation size
  - Internal sizes are **designed to keep internal fragmentations  $< 25\%$** <sup>[Supermalloc][jemalloc]</sup>
- Large size category (e.g.,  $> 1$  MB)
  - Rounded up to the nearest power of 2
  - **Designed not to waste physical memory much**
    - Thanks to demand paging, untouched pages do not consume physical memory
    - Worst case: 1.6% when allocating 1MB + 1 B with 4 KB page

# Metall Internal Architecture

## Application Heap Segment & Allocation Sizes

- Application heap segment
  - All application data (allocated by Metall) are stored in this region
  - Reserves a large (e.g., a few TB) continuous VM region in the process's address space
  - Maps backing files to the VM region on demand



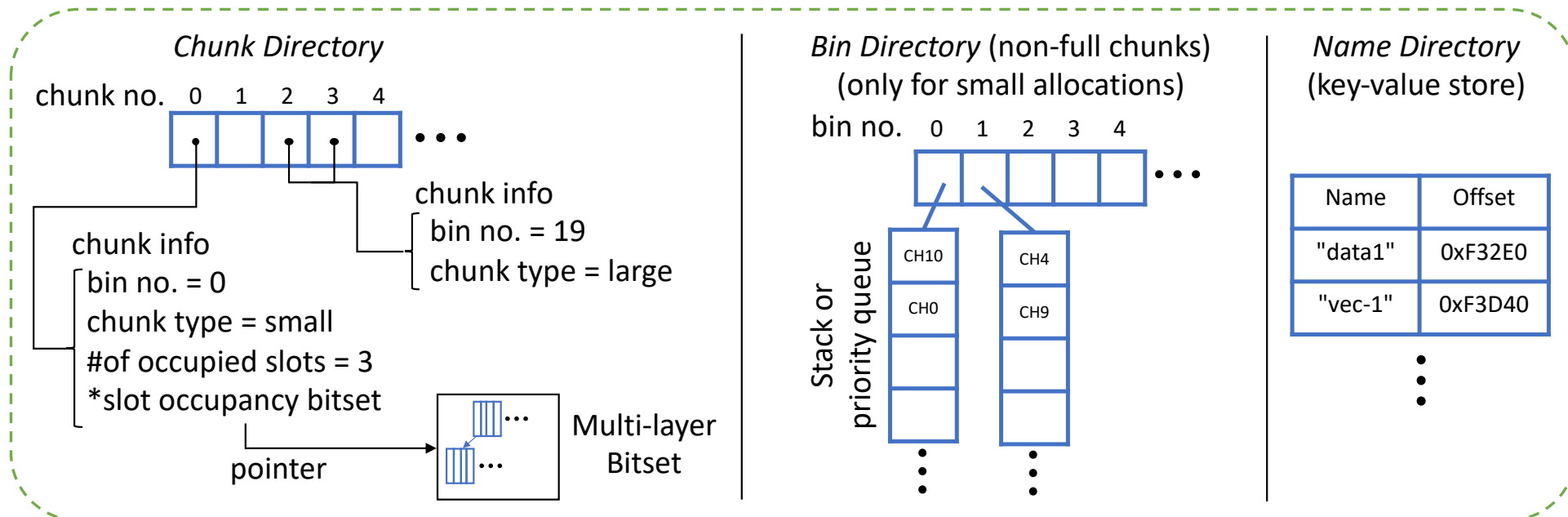
The default chunk size is 2 MB

# Metall Internal Architecture

## Memory Allocation Management Data (SLUB allocator)

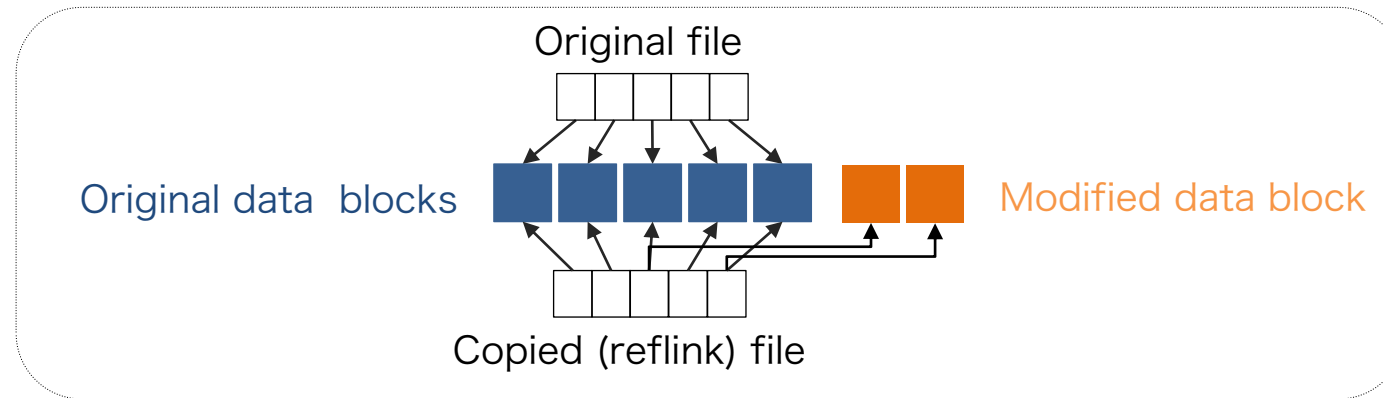
- **Allocated in DRAM**, separating from application heap segment to improve data locality
  - Unserialized/serialized when Metall's constructor/destructor is called
  - Employs state-of-the-art allocation algorithms
  - Free-slot caches
    - CPU core level to improve multi-thread performance

*Management Data (constructed in **DRAM**)*



# Snapshot/Versioning in Metall

- Calls `msync()` and copies backing-files to another location using *reflink*
- reflink
  - copy-on-write file copy mechanism implemented in filesystems (e.g., XFS, ZFS, Btrfs)



- In case reflink is not supported by the filesystem, Metall automatically falls back to a regular copy

# Hands-on Session



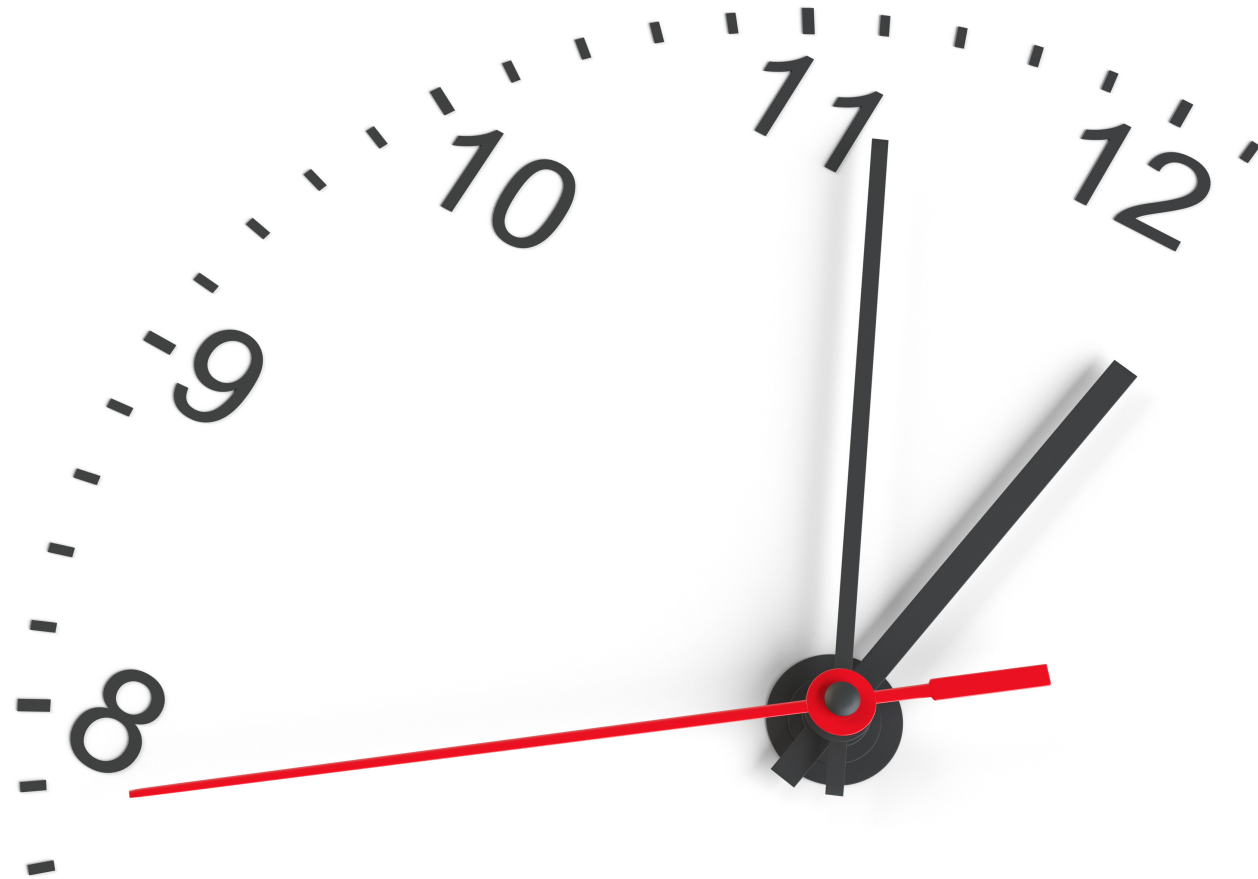
# Hands-on Instruction

---

1. Visit GitHub
2. Search 'LLNL/metall' (two 'l' at the end)
3. Click [tutorial] -> [hands\_on]
  1. Full URL: [https://github.com/LLNL/metall/tree/master/tutorial/hands\\_on](https://github.com/LLNL/metall/tree/master/tutorial/hands_on)



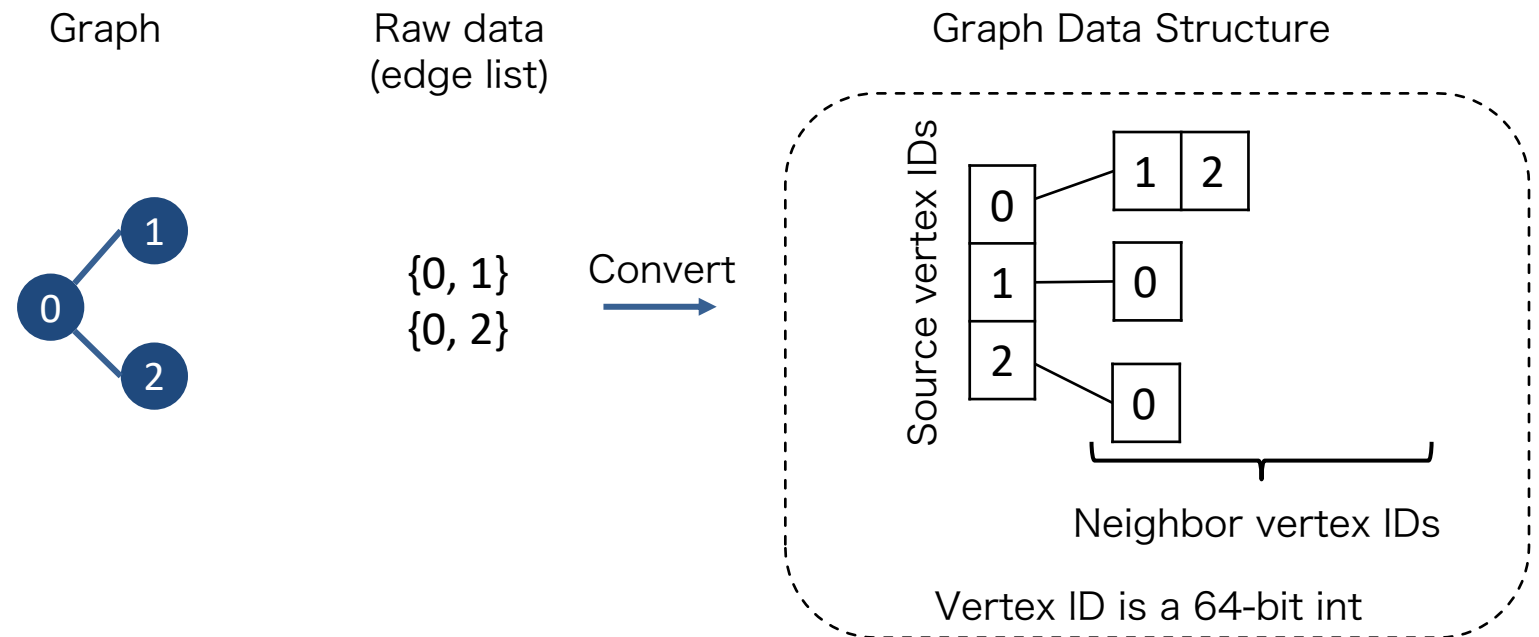
# Evaluation



# Evaluation

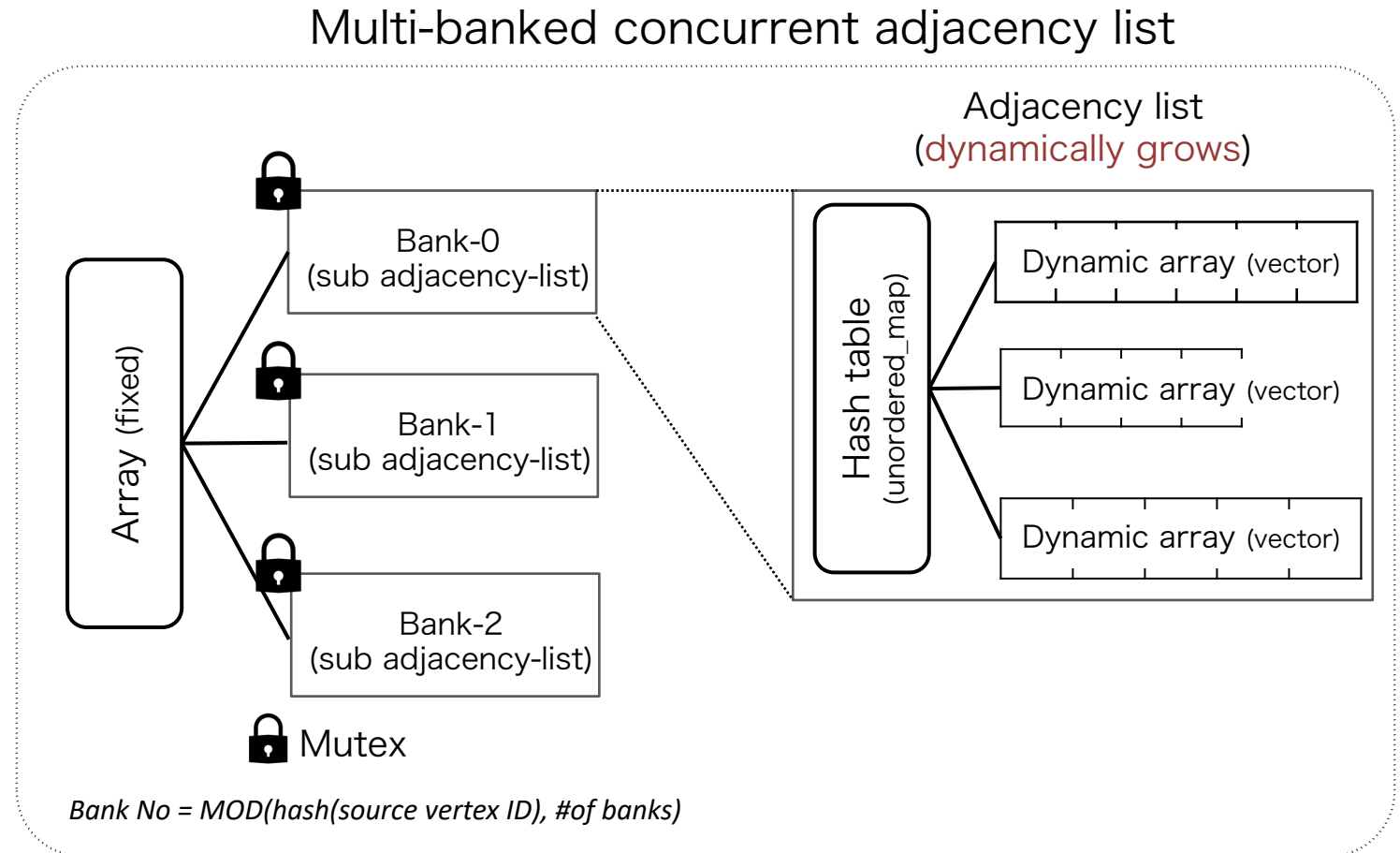
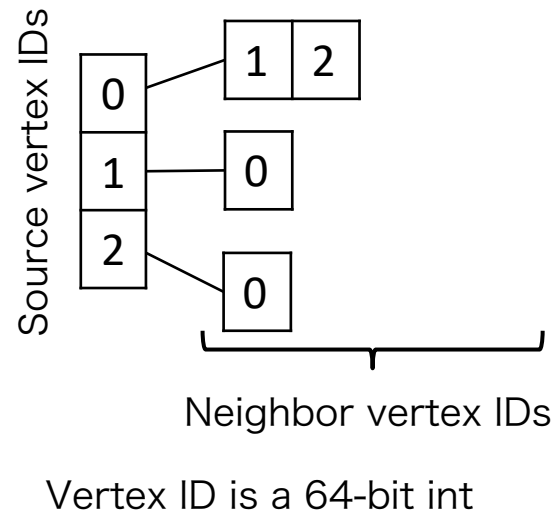
## Graph Construction Benchmark

- A necessary step before performing actual graph analytics
- Partition and ingest raw graph data into a memory access efficient data structure
- Often more expensive graph analytics step



# Evaluation Graph Data Structure

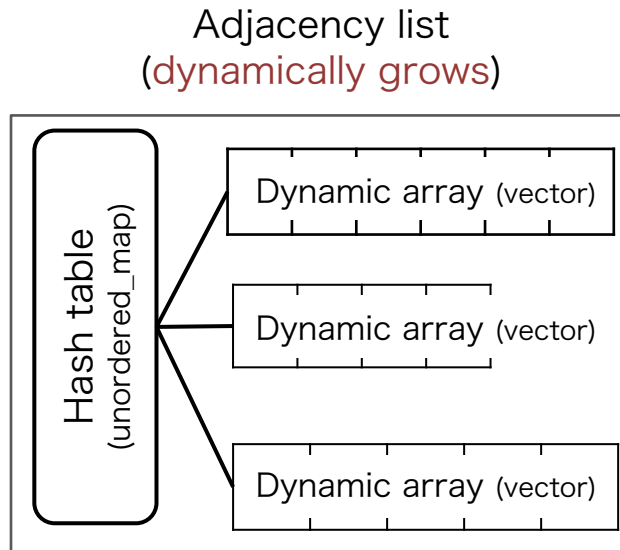
- Adjacency list (one of de-facto standard graph data structures)



# Evaluation

## Adjacency List Implementation

Adjacency-list with the default allocator (std::allocator)



```
class AdjacencyList {  
  
    using edgeVec = vector<int>;  
    using vertexTable = unordered_map<int, edgeVec>;  
    vertexTable table;  
  
    AdjacencyList() : table {}  
  
    void addEdge(int source, int target) {  
        table[source].push_back(target);  
    }  
  
}
```

\*some unimportant details are omitted

# Evaluation

## Allocator-Aware Adjacency List

- Adjacency list with a custom allocator

```
template <class Alloc = std::allocator<std::byte>>
class AdjacencyList {

    using edgeVec = vector<int, Alloc::rebind<int>::other>;

    using tableAlloc = Alloc::rebind<pair<int, edgeVec>>::other;
    using vertexTable = unordered_map<int, edgeVec, /* ... */, tableAlloc>;
    vertexTable table;

    AdjacencyList(Alloc alloc = Alloc()) : table(alloc) {}

    // No changes to this method
    void addEdge(int source, int target) {}
}
```

Changed to an allocator-aware data structure, following the C++ standard style  
(no code depends on Metall)

# Evaluation

## Allocate Adjacency List Using Metall

With Metall

```
using AdjList = AdjacencyList<metall::manager::allocator_type<std::byte>>;  
  
metall::manager metall_mgr(metall::create_only, "/ssd/graph");  
  
auto* adj = metall_mgr.construct<AdjList>("graph")  
                                         (metall_mgr.get_allocator());  
  
adj->addEdge(1, 2);
```

With std::allocator (allocate in DRAM, no persistency)

```
auto* adj = new AdjacencyList();  
adj->addEdge(1, 2);
```

The same data structure can be used with std::allocator

# Evaluation

## Machine Configuration

- Used two **single-node** machines at LLNL

EPYC (conventional PM device)

Storage	NVMe SSD
DRAM	256 GB
CPU	AMD EPYC CPU x 2 (96 threads)



Optane (byte addressable PM device)

Storage	Intel Optane DC Persistent Memory (App Direct Mode + DAX)
DRAM	192 GB
CPU	Intel Skylake x 2 (96 threads)





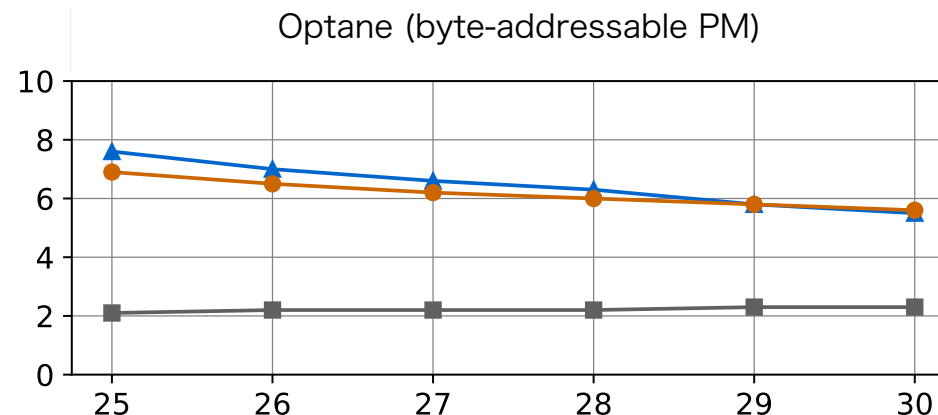
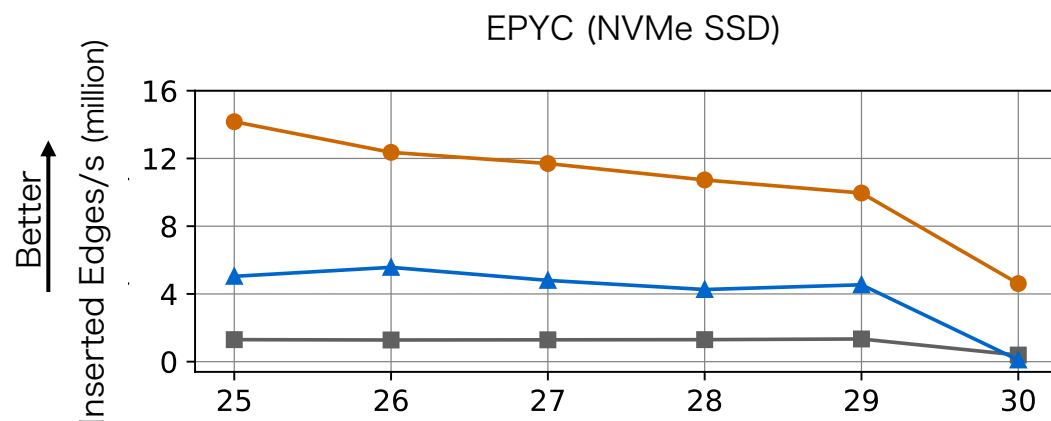
# Evaluation Result

## Dynamic Graph Construction

shared-memory  
multi-thread

- Baselines (memory allocators that use file-backed mmap underneath)
  - Boost.Interprocess
    - Uses a single tree structure for memory allocation management
  - memkind (PMEM kind)
    - Provides an allocator built on top of *jemalloc*
    - **Cannot reattach data** (uses PM as extended volatile memory)

■ Boost.Interprocess (BIP) ▲ memkind (PMEM kind) ● Metall



Graph SCALE (the number of inserted edges =  $2^{\text{SCALE}} \times 32$ )  
(SCALE 30 is larger than DRAM)

Metall provides persistent memory features whereas PMEM kind does not

# Related Work

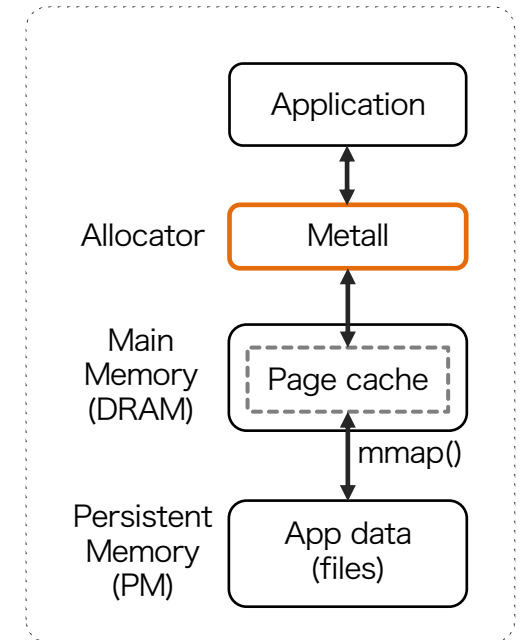
- Heap allocators (e.g., jemalloc, tcmalloc, malloc implementations)
  - Many studies have been conducted and showed notable results
  - Cannot persistently store their internal structures
- Persistent Memory Allocator
  - NVMalloc
    - Allocates memory on a distributed non-volatile memory (NVM) storage system
    - Creates a file per memory allocation request
  - libpmemobj (in PMDK)
    - Employs a fine-grained persistence policy (ideal for transactional operations)
  - Boost.Interprocess
    - Designed for interprocess communication (not designed as a persistent memory)
- Persistent Data Store
  - Hierarchical Data Format (HDF)
    - Allows applications to store data with portable formats
    - Metall is designed as a lightweight tool by limiting data portability

Metall is designed as a lightweight and high-performance persistent memory allocator with the coarse-grained persistence policy

# Summary (1/2)

## Metall

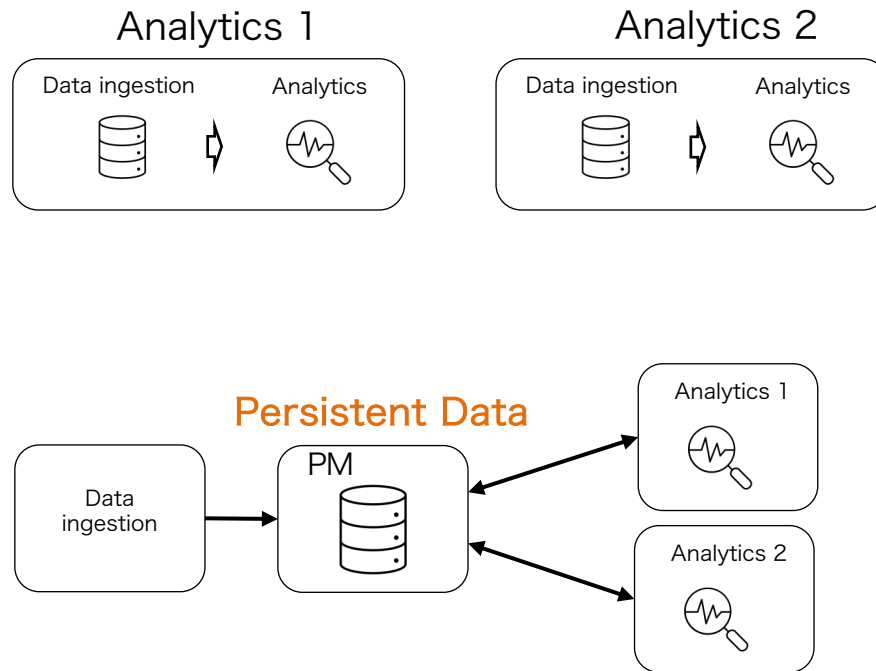
- A memory allocator built on top of a memory mapping region
  - Designed to work on any devices with file system support (including *tmpfs*)
- Enables applications to **allocate heap-based objects into PM**, just like main-memory
- Rich API for custom C++ data structures
- Employs the coarse-grained consistency model
- Provides snapshot/versioning capabilities
- Incorporates state-of-the-art allocation algorithms



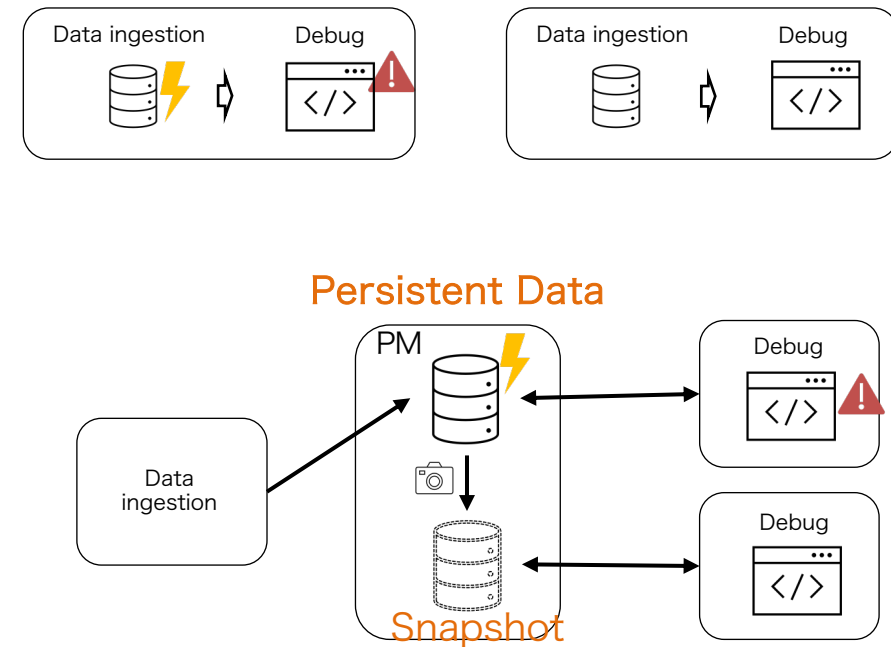
## Summary (2/2)

# Metal: A Persistent Memory Allocator For Data-Centric Analytics

### Data Analytics Workflow



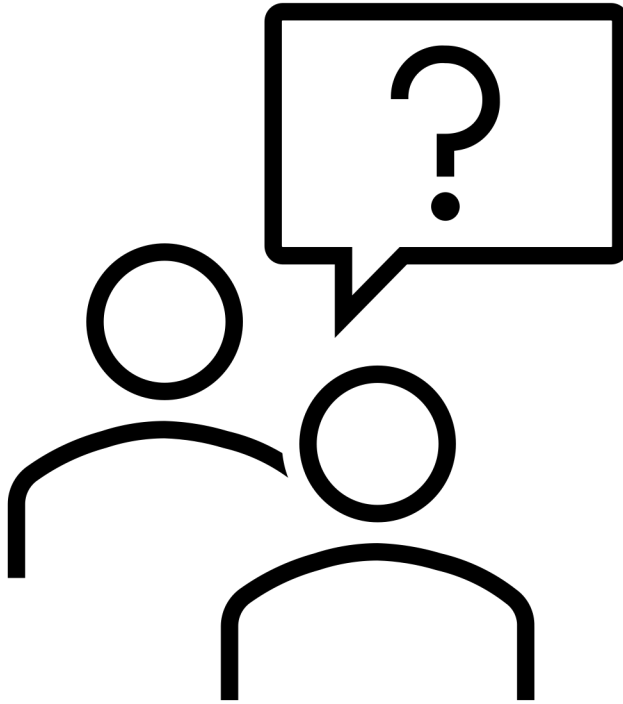
### Development/Debug Workflow



Metal enables applications to efficiently leverage persistent memory for data-centric computing

# Questions?

---





# CASC

Center for Applied  
Scientific Computing



#### Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.