GBTL+Metall – Adding Persistence to GraphBLAS

Kaushik Velusamy
Department of Computer Science
University of Maryland
kaushikvelusamy@umbc.edu

Scott McMillan
Software Engineering Institute
Carnegie Mellon University
smcmillan@sei.cmu.edu

Keita Iwabuchi, Roger Pearce
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
{kiwabuchi, rpearce}@llnl.gov

***Abstract*—It is well known that software-hardware co-design is required for attaining high-performance implementations. System software libraries help us in achieving this goal. Metall persistent memory allocator is one such library. Metall enables large scale data analytics by leveraging emerging memory technologies. Metall is a persistent memory allocator designed to provide developers with rich C++ interfaces to allocate custom C++ data structures in persistent memory, not just from block storage and byte addressable persistent memories (NVMe, Optane) but also in DRAM TempFS. Having a large capacity of persistent memory changes the way we solve problems and leads to algorithmic innovation. In this work, we present GraphBLAS as a real application use case to demonstrate Metall persistent memory allocator benefits. We show an example of how storing and re-attaching graph containers using Metall, eliminates the need for graph reconstruction at a one-time cost of re-attaching to Metall datastore.**

I. Introduction

The exponential growth in data across multiple domains creates challenges in terms of storing and processing data efficiently. The Standard Template Library (STL) data structures are designed for DRAM and do not work well with flash. A design change is needed to support page-oriented, locality-preserving data structures with high thread-level concurrency. Metall library solves this problem by providing a persistent memory allocator to custom C++ data structures. We used graph algorithms from the GraphBLAS Template Library (GBTL) [4] as a real application use case to demonstrate Metall persistent memory allocator benefits. We show an example of how storing and re-attaching graph containers using Metall eliminates the need for graph reconstruction at an one-time cost of re-attaching to Metall datastore.

II. Metall

Metall is a persistent memory allocator designed to provide developers with an API to allocate custom C++ data structures in both block-storage and byte-addressable persistent memories. Metall relies on a file-backed *mmap* mechanism to map a file in a file system into the virtual memory of an application, allowing the application to access the mapped region as if it were regular memory. Such *mmap* mappings can be larger than the physical main-memory of the system, allowing applications to address datasets larger than physical main-memory (often referred to as out-of-core or external memory). Metall incorporates state-of-the-art allocation algorithms in Supermalloc with the rich C++ interface developed by Boost.Interprocess, and provides persistent memory snapshotting (versioning) capabilities. Metall provides the ability to create a persistent object for out-of-core graph algorithms with very large datasets and thereby allow real-time access to large datasets. For more information on the Metall internal architecture, API, persistence policy and snapshotting, refer to [1].

III. Graph Analytics and GraphBLAS

Graph analytics enables us to develop new data processing capabilities. One of the main problems in graph analytics is the need to persist the data beyond the scope of a single execution. Graph algorithms suffer from irregular access patterns that may limit their performance even on DRAM. Graph construction, indexing and regular updates are often more expensive than the analytics itself. With persistent memory, data structures once constructed, can be re-analyzed and updated beyond the lifetime of a single execution. GraphBLAS specifies a set of building blocks for computing on graphs and graph-structured data expressed in the language of linear algebra [2]. This approach represents graphs as sparse matrices, and operations using an extended algebra of semirings. An almost unlimited variety of operators and types are supported for creating a wide range of graph algorithms. The GraphBLAS Template Library (GBTL) is a C++ reference implementation of the GraphBLAS specification [4].

IV. Experiments

A. Machine Configuration

The **Optane** machine has an Intel Xeon Platinum 8260L CPU (48 cores, 96 threads), 192 GB of DRAM, and a 1.5 TB of Intel Optane DC Persistent Memory. In this experiment, we use the App Direct Mode as the operating mode. Here, the device shows up in the system as if it is a conventional block device. We configured the device with EXT4 file system DAX mode to bypass the page cache layer and to enable fine-grained I/O rather than the page granularity.

The **Flash** machine has an Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz (12 cores, 48 threads) 256 GB of DRAM and a 1.5TB of NVMe SSD.

B. Algorithms and Dataset

We used two GBTL algorithms – triangle counting, and breadth-first search (BFS) – as a real application use case to demonstrate Metall persistent memory allocator benefits. Triangle counting is a technique used to discover cliques within a graph. Unlike BFS and PageRank, triangle counting is a non-iterative algorithm that requires each node to count the intersections among the neighbors of its immediate neighbors. BFS is a traversal algorithm that starts at a given source

node and produces a list of nodes that are reachable from the source node by traversing the edges of the graph. In each iteration, only the nodes adjacent to a newly discovered node are processed. For the experiments, we used the adjacency list datasets from SNAP [5].

C. Metall GBTL Data structure

GBTL uses a vector of vector of tuple as its graph container. In such multi-level containers, we use a scoped allocator adaptor in the outermost container, so that the inner containers obtain their allocator arguments from the outer container's scoped allocator adaptor. This is shown in Listing 1 with the use of Boost container vectors here.

```
using ElementType = std::tuple<IndexType, ScalarType>;
using element_allocator_t = typename std::allocator_traits<allocator_t>::template
      rebind_alloc<ElementType>;
using inner_vector_type = bc::vector<ElementType, element_allocator_t>;
using outer_vector_allocator_type = bc::scoped_allocator_adaptor<typename
      std::allocator_traits<allocator_t>::template
      rebind_alloc<inner_vector_type>>;
using outer_vector_type  = bc::vector<inner_vector_type,
      outer_vector_allocator_type>;
outer_vector_type graph_data;
// Here outer_vector_type is List-of-lists storage (LIL) Metall graph container
```

Code 1. Building the GBTL Metall graph container

The Metall utility provides a fallback allocator adaptor, which allows a STL-compatible allocator to fallback to a heap allocator like malloc by passing an empty constructor argument. The purpose of this allocator is to provide a way to quickly integrate Metall into an application that occasionally wants to allocate 'metallized' classes as non-persistent data structures in 'DRAM'. Metall with GBTL allows the user to efficiently mix persistent data storage for the large graph matrices with the non-persistent algorithm temporaries stored in DRAM.

```
using allocator_t = metall::utility::fallback_allocator_adaptor
                             <metall::manager::allocator_type<char>>;
using Metall_MatType = grb::Matrix<T, allocator_t>;
//================= Graph Construction in Metall Scope ========================
{
   metall::manager manager(metall::create_only, "/mnt/nvme/datastore");
   Metall_MatType *A = manager.construct<Metall_MatType>("gbtl_lil_matrix")
                ( NUM_NODES, NUM_NODES, manager.get_allocator());
   A->build(iA.begin(), jA.begin(), v.begin(), iA.size());
}
// Exist the program and reattach the data
//================= Triangle Counting in Metall Scope =========================
{
   metall::manager manager(metall::open_only, "/mnt/nvme/datastore");
   Metall_MatType *A = manager.find<Metall_MatType>("gbtl_lil_matrix").first;
   T count(0);
   count = algorithms::triangle_count_masked_noT(*A);
}
// Exist the program and reattach the data
//================= single BFS in Metall Scope ================================
{
   metall::manager manager(metall::open_only, "/mnt/nvme/datastore");
   Metall_MatType *A = manager.find<Metall_MatType>("gbtl_lil_matrix").first;
   grb::Vector<T> parent_list(NUM_NODES);
   grb::Vector<T> root(NUM_NODES);
   root.setElement(0, 0);
   algorithms::bfs(*A, root, parent_list);
}
```

Code 2. Example of using GBTL containers with graph algorithms

D. Results

Figure 1 and 2 presents the graph construction and algorithm time with Metall on Flash and Optane machine. It is clear that the BFS algorithm time is much shorter than the graph construction time. As shown in the Listing 2, Metall provides a mechanism to exit the program and reattach to the previously

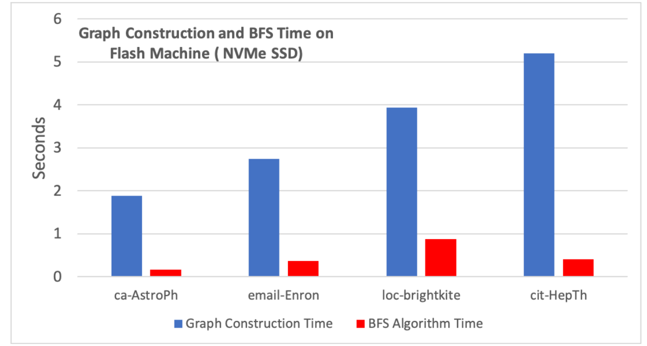

Fig. 1. Graph construction and algorithm time with Metall on Flash machine

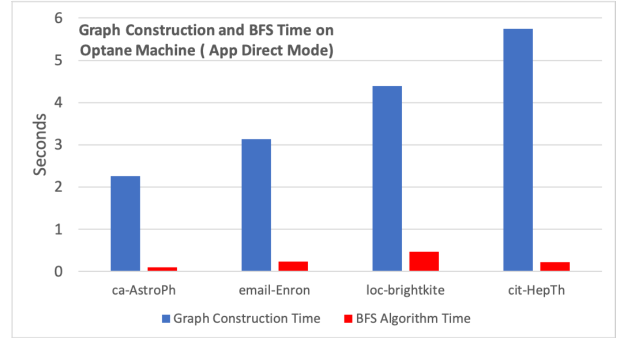

Fig. 2. Graph construction and algorithm time with Metall on Optane machine

created data, avoiding construction time. This would be helpful to many graph analytics applications where the data structure reconstruction can be completely avoided.

V. Conclusion

Memory-mapped persistent pre-built data structures is helpful in enabling interactive real time data science applications with large persistent data structures in unprecedented scales of data, without going through traditional serialization and data structure reconstruction. Application developers can create custom complex persistent and consistent data structures. This ability to attach and detach from previously created datasets in a lightweight manner gives a powerful workflow software productivity benefit.

Acknowledgment

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center [DM21-0180] . This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Lab- oratory under Contract DE-AC52-07NA27344 [LLNL-ABS-819988]. Experiments were performed at the Livermore Computing facility.

References

[1] K. Iwabuchi, L. Lebanoff, R. Pearce, and M. Gokhale, "Metall: A persistent memory allocator enabling graph processing," in Workshop on Irregular Applications: Architectures and Algorithms(IA3-SC2019), https://www.osti.gov/servlets/purl/1576900, 2019.
[2] Jeremy Kepner, Peter Aaltonen, David Bader, Aydın Buluc¸, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Jos´e Moreira, John Owens, Carl Yang, Marcin Zalewski, and Timothy Mattson, "Mathematical foundations of the GraphBLAS", . In IEEE High Performance Extreme Computing (HPEC), 2016.
[3] Benjamin Brock et all., "Considerations for a Distributed GraphBLAS API", in IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW),2020.
[4] Scott McMillan, GraphBLAS Template Library (GBTL) v3.0: Design and Implementation, https://github.com/cmu-sei.gbtl, 2020
[5] Jure Leskovec and Andrej Krevl, SNAP Datasets Stanford Large Network Dataset Collection, http://snap.stanford.edu/data, June 2014