miniVite + Metall: A Case Study of Accelerating Graph Analytics Using Persistent Memory Allocator

Keita Iwabuchi*, Sayan Ghosh†, Roger Pearce*, Mahantesh Halappanavar†, Maya Gokhale*

*Lawrence Livermore National Laboratory, Livermore, CA, USA {kiwabuchi,rpearce,gokhale2}@llnl.gov
†Pacific Northwest National Laboratory, Richland, WA, USA {sg0,hala}@pnnl.gov

I. Introduction

The task of ingesting data — indexing and partitioning data in preparation for running analytics — is often more expensive than the analytics itself. Furthermore, the same data (or derived data) is re-ingested frequently in actual situations, e.g., running different analytics to the same data, evaluating different parameters during analytics, and developing/debugging an analytics program. miniVite [1] is a distributed-memory graph community detection/clustering mini-application, which supports in-memory random geometric graph generation. We observe significant overhead at scale in the parallel graph generation part due to its expensive computation and communication.

Substantial performance improvements and cost reductions have occurred in *persistent memory* (PM) technology (e.g., NVMe SSD and byte-addressable NVM). We use the term of *persistent memory* as a concept, and actual PM hardware could be any non-volatile memory (NVM). The promise of PM is that, once constructed, data structures can be re-analyzed and updated beyond the lifetime of a single execution.

We study the potential impact of persistent memory (PM) in real graph analytics workloads, changing miniVite to support PM as a case study. To optionally store the generated graph and provide options to reload it in a successive run, we integrate Metall [2], a persistent memory allocator built on top of a file-backed memory mapping region, into miniVite. Our experiments show improvements of up to 85× and 65× on NERSC Cori and OLCF Summit supercomputers (using 128-1024 processes), respectively, by reusing a persistently stored graph instead of regenerating it.

A. miniVite

Graph clustering, popularly known as community detection, is an important graph kernel used in a number of scientific and social networking applications for discovering higher order structures within a graph [3]. The Louvain algorithm is a multi-phase multi-iterative heuristic to identify a community-wise partitioning of vertices in a graph that optimizes modularity [4].

`miniVite` [1] is part of the U.S DOE ECP proxy application suite [5] and implements the very first phase of the Louvain method (without rebuilding the graph). `miniVite` can also generate Random Geometric Graphs (RGG) in parallel, thereby making it convenient for users to parameterize synthetic graphs to run the Louvain method in distributed-memory. We specifically chose RGGs because they are known to naturally exhibit consistent community structure with high modularity [6], as opposed to scale-free graphs.

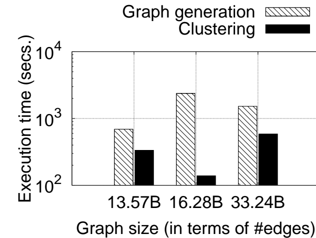

Fig. 1. Performance of graph generation/creation as compared to clustering in miniVite for three different graphs on 8192 processes. For all the cases, the performance gap between graph generation and clustering is significant.

Despite the convenience of an in-memory parallel graph generator, we observed significant overhead at scale in the parallel graph generation part of miniVite: Fig. 1 demonstrates the relative cost of graph generation compared to clustering (up to 10-20×) on 8192 processes. miniVite provides several MPI communication models (two-sided, collective and one-sided) and uses OpenMP for utilizing shared-memory parallelism. Hence, it can be used for repetitive performance measurements by using the same input graph on a platform, under different MPI communication models, varying the number of OpenMP threads and potentially using different runtime systems. Therefore, reducing the overhead of graph generation on successive runs can improve the developer productivity.

B. Metall

Metall [2] is a C++ memory allocator for persistent memory. Metall relies on a file-backed *mmap* mechanism to map files in a filesystem into the virtual memory of an application, allowing the application to access the mapped region as if it were regular memory.

In addition to basic memory allocation interfaces (e.g., *malloc* and *free*), Metall leverages the Boost.Interprocess (BIP) [7] API to allocate C++ objects, including Standard Template Library (STL) containers, into persistent memory. By employing mmap and BIP API, Metall can allocate complex data structures into persistent memory directly — it can avoid serialization, which is an expensive operation.

II. METHODOLOGY

Since clustering is the first-class candidate of miniVite, we set out to optimize the parallel graph generation part by making use of the relatively high bandwidth (multiple orders of GB/s) available between compute nodes and the I/O resources. In order to facilitate the storing and loading of intermediate graph data, we employ Metall to store the generated graph using file-backed memory and provide options to reload it on a successive run. This makes it possible to generate the graph once, and then reuse and also modify it upon successive runs, thereby making it possible to bypass the expensive graph generation.

miniVite uses three STL vector container objects to hold a graph with a Compressed Sparse Row (CSR) format. To store the graph class into persistent memory, we change it to an allocator-aware class like the STL containers. Specifically, the new graph class takes an allocator type in its template and an allocator object in its constructors. The pseudocode of an allocator-aware graph class is described in Code 1.

Code 1. Allocator-aware Graph Class

```
// Allocator-aware CSR graph
template<class vertex_id_type, class allocator>
class graph {
  std::vector<vertex_id_type, allocator_type> vec; // Pass the allocator type
  // Two more vectors here
  graph(allocator alloc) : vec(alloc) {} // Constructor takes an allocator object
  // No code change in the other methods
};
```

We also show how to create and load the graph class using Metall in Code 2. There are only two lines that are necessary to allocate or load a C++ object using Metall. After allocating/loading a graph object, all methods in the vector container can be used normally — including the ones that require dynamic memory allocations, such as *resize* and *push_back*. Metall synchronizes application heap with backing files when the destructor of metall::manager is invoked. Since miniVite is an MPI program, each process allocates an independent Metall manager object.

Code 2. Store and Load a Graph Object Using Metall

```
using graph_t = graph<int, metall::allocator<int>>;
// Allocate and construct a new graph
void create() {
 // Create a new Metall datastore with file path '/ssd/mydata'.
 metall::manager mgr(metall::create_only, "/ssd/mydata");
 // Allocate and construct an object of graph_t with key 'graph'.
 // Pass an STL-style allocator object to graph_t's constructor
 graph_t* g = mgr.construct<graph_t>("graph")(mgr.get_allocator<int>());
 // Construct a graph. No code change here.
}
// Load an existing graph data
void Load() {
 // Open an existing datastore.
 metall::manager mgr(metall::open_only, "/ssd/mydata");
 // Metall returns the address of the object of "graph"
 graph_t* g = mgr.find<graph_t>("graph").first;
 // Run graph analytics. No code change here.
}
```

III. EVALUATION

We perform exhaustive evaluations on multiple platforms to demonstrate the benefits of having the option in miniVite to store the generated graph, and load it on a successive run. We used NERSC Cori and OLCF Summit supercomputers as our evaluation testbeds. Generated graphs are stored to and loaded from the Lustre filesystem (using the default striping options) on NERSC Cori, and the GPFS on OLCF Summit.

Since parallel file systems tend to show high latency and to be prone to variabilities in performance based on the network traffic, we also allow an optional staging mechanism to a temporary data access space such as `tmpfs` or SSD. Metall copies backing files from a staging area to a parallel file system at the end of the graph generation step, and vice versa at the beginning of the graph loading step. For Cori, we show the impact of using `tmpfs` for staging, whereas for Summit we use the 1.6 TB on-node NVMe (uses XFS) for staging the data en route to/from GPFS. We use Cray-MPICH/7.7.10 and Intel v19.0.1 compiler on NERSC, whereas for OLCF we use IBM Spectrum MPI v10.3.0 and GNU C++ v7.4.

TABLE I
EXECUTION TIMES (IN SECONDS) OF GRAPH GENERATION, AND SUBSEQUENT METALL GRAPH LOADING (W/ AND W/O TMPFS FOR STAGING) AND CLUSTERING IN MINIVITE ON NERSC CORI.

Processes (#edges)	Graph Generation			Graph Loading		Clustering time
	Standard	w/ Metall		w/ Metall		
		Luster	Lustre w/ tmpfs	Luster	Lustre w/ tmpfs	
128 (89.7M)	33.42	34.17	32.09	0.40	1.69	0.85
256 (184.3M)	33.75	34.87	32.21	0.50	0.71	1.00
512 (378.1M)	33.97	35.49	32.68	0.56	1.12	1.15
1024 (774.9M)	34.80	38.04	34.07	1.43	1.90	1.36

TABLE II
EXECUTION TIMES (IN SECONDS) OF GRAPH GENERATION, AND SUBSEQUENT METALL GRAPH LOADING (W/ AND W/O ON-NODE SSD FOR STAGING) AND CLUSTERING IN MINIVITE ON OLCF SUMMIT.

Processes (#edges)	Graph Generation			Graph Loading		Clustering time
	Standard	w/ Metall		w/ Metall		
		GPFS	GPFS w/ SSD	GPFS	GPFS w/ SSD	
128 (89.7M)	36.79	31.07	36.76	1.30	0.57	1.19
256 (184.3M)	38.19	31.06	37.15	0.23	0.59	1.39
512 (378.1M)	60.05	57.03	38.25	0.28	1.23	1.75
1024 (774.9M)	48.22	31.13	40.52	0.42	0.98	1.75

The performance characteristics of storing and subsequent loading of the miniVite RGGs (random geometric graphs) on the respective testbed platforms are shown in Table I and Table II. Overall storing and loading the per-process graph in parallel to/from a data store using Metall showed very low overhead. Once a generated graph is stored using Metall, it can be reused as many times as long as the same number of processes that were used to store the original graph are used in the subsequent executions. We achieved about 65×/ 85× performance improvement on NERSC Cori and OLCF Summit systems by bypassing the graph generation stage on subsequent runs.

ACKNOWLEDGMENT

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. We used resources of the NERSC and ALCF facility, supported by U.S. DOE SC under Contract No. DE-AC02-05CH11231 and Contract DE-AC02-06CH11357, respectively. PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830. Prepared by LLNL under Contract DE-AC52-07NA27344 (LLNL-ABS-813827).

References

[1] S. Ghosh, M. Halappanavar, A. Tumeo, A. Kalyanaraman, and A. H. Gebremedhin, “minivite: A graph analytics benchmarking tool for massively parallel systems,” in *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 2018, pp. 51–56.

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

[3] S. Fortunato, “Community detection in graphs,” *Physics Reports*, vol. 486, no. 3, pp. 75 – 174, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0370157309002841

[4] V. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, “Fast unfolding of communities in large networks,” *Journal of Statistical Mechanics: Theory and Experiment*, p. P10008., 2008.

[5] D. F. Richards, O. Aaziz, J. Cook, H. Finkel, B. Homerding, P. McCorquodale, T. Mintz, S. Moore, A. Bhatele, and R. Pavel, “Fy18 proxy app suite release. milestone report for the ecp proxy app project,” Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2018.

[6] E. Davis and S. Sethuraman, “Consistency of modularity clustering on random geometric graphs,” *arXiv preprint arXiv:1604.03993*, 2016.

[7] “Boost Libraries,” https://www.boost.org.