Single-Node Partitioned-Memory for Huge Graph Analytics: Cost and Performance Trade-Offs

Sayan Ghosh
Pacific Northwest National Lab
Richland, Washington, USA
sayan.ghosh@pnnl.gov

Nathan R. Tallent
Pacific Northwest National Lab
Richland, Washington, USA
tallent@pnnl.gov

Marco Minutoli
Pacific Northwest National Lab
Richland, Washington, USA
marco.minutoli@pnnl.gov

Mahantesh Halappanavar
Pacific Northwest National Lab
Richland, Washington, USA
hala@pnnl.gov

Ramesh Peri*
Facebook
Austin, TX, USA
rvperi@fb.com

Ananth Kalyanaraman
Washington State University
Pullman, Washington, USA
ananth@wsu.edu

ABSTRACT

Because of cost, non-volatile memory NVDIMMs such as Intel Optane are attractive in single-node big-memory systems. We evaluate performance and cost trade-offs when using Optane as *volatile* memory for huge-graph analytics. We study two scalable graph applications with different work locality, access patterns, and parallelism. We evaluate single and partitioned address spaces—Memory and AppDirect modes—and compare with distributed executions on GPU-accelerated and CPU-based supercomputers.

We show that AppDirect can perform and scale better than Memory for the largest working sets (12%), even when dominated by irregular access patterns, if most accesses are NUMA-local and Optane accesses are frequently reads. Surprisingly, between Memory and AppDirect, processor-cache performance can change due to line invalidations; updates to the caching policy (via non-temporal hints) can make a 25% improvement. We observe that single-node graph analytics frequently has >4–10× cost/performance advantages over distributed-memory executions on supercomputers.

CCS CONCEPTS

• **Hardware → Non-volatile memory**; • **Computing methodologies → Parallel algorithms**; • **Theory of computation → Graph algorithms analysis**; • **General and reference → Performance**.

KEYWORDS

non-volatile memory, graph analytics, performance evaluation

ACM Reference Format:
Sayan Ghosh, Nathan R. Tallent, Marco Minutoli, Mahantesh Halappanavar, Ramesh Peri, and Ananth Kalyanaraman. 2021. Single-Node Partitioned-Memory for Huge Graph Analytics: Cost and Performance Trade-Offs. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3458817.3476156

*This work was performed while at Intel Corporation.

SC '21, November 14–19, 2021, St. Louis, MO, USA

ACM ISBN 978-1-4503-8442-1/21/11…$15.00
https://doi.org/10.1145/3458817.3476156

1 INTRODUCTION

Graph analytics, using graph algorithms for exploration and discovery, is a rapidly emerging area of data science that has been especially enabled by the availability of massive data. Implementing graph algorithms on distributed-memory platforms is challenging; and their execution is parallel-inefficient due to computation-communication imbalances [28]. Shared-memory systems are a popular alternative, but may not hold graphs with billions to trillions of vertices (entities) and edges (relationships).

An emerging memory option for shared-memory systems is non-volatile memory (NVDIMMs or PMM for persistent memory module or persistent memory), now widely available as Intel Optane. Optane is about half the cost of DRAM [30] and enables large, shared, and byte-addressable memory on single-node systems. Typical Optane-based systems provide 25% more memory for 20–30% cost reduction (cf. Table 5). For example, an Intel Cascade Lake CPU with 6 memory channels (2 DIMMs/channel) supports 1.5 TB DRAM (256 GB DDR4) and 3 TB of PMM (512 GB NVDIMM). An 8-socket system supports up to 24 TB Optane PMM; such systems are now available on Amazon AWS for large in-memory databases. The recent Cooper Lake CPUs support Optane 200, which advertises 25% more memory bandwidth than Optane 100 modules [16, 17].

Systems with PMM form a relatively new offering for "big memory" workloads: single-node systems with coherent shared-memory spaces, *formed from multiple-memory domains as well as multiple memory types* (DRAM and PMM). Although providing the benefit of larger memory capacity, these systems bring a host of challenges ranging from a variety of memory configurations, programming options, and performance differences from DRAM (e.g., asymmetric latency, longer latency, reduced bandwidth, and reduced scaling).

This paper evaluates the suitability of huge-graph analytics on single-node systems with PMM as a *volatile* pool. There are two principal ways to configure PMMs. In Memory mode, DRAM acts as a direct mapped cache for the entire persistent memory space: total volatile capacity is the size of PMM. In AppDirect mode, DRAM and PMM are separate: total volatile capacity is the sum. Prior work using AppDirect mode views PMM as a persistent pool, accessed through a filesystem. The filesystem is used even when using a PMM memory allocator such as `memkind` [43]. We use Linux's emerging support for accessing PMM through the virtual memory system via special NUMA nodes. With either AppDirect interface, the address space is partitioned, forming a *heterogeneous*

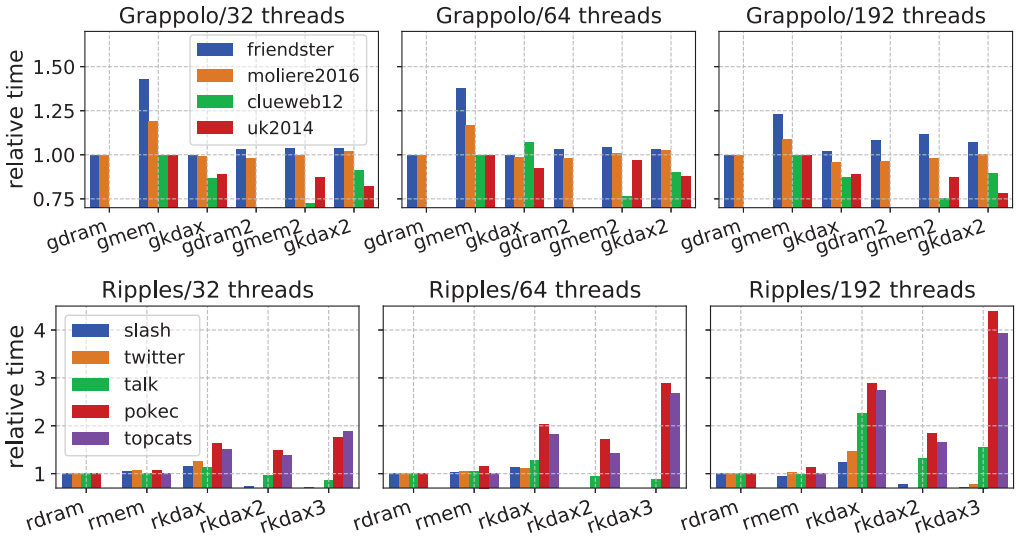


Figure 1: Performance of community detection (Grappolo) and influence maximization (Ripples), at different thread counts, for combinations of application variants (cf. §4.3.2 and §4.3.3) and Optane memory modes. Combinations along horizontal axis formed from application name (prefix of *g* or *r*), memory mode—DRAM (*dram*), Memory (*mem*), and AppDirect (*kdax*)—and variant (integer suffix). Performance is relative to leftmost combination, i.e., base DRAM variant (*gdram* or *rdram*) or Memory variant (*gmem* or *rmem*); values below 1.0 indicate a speedup. AppDirect variants can perform better than Memory: gkdax > gmem always (except clueweb12 at 64 threads); gkdax2 > gmem2 for uk2014; rkdax{2|3} > rmem for slash, twitter, and talk (≤ 64 threads). AppDirect and Memory variants can match base DRAM variant: gkdax for friendster and moliere2016; gmem2 for moliere2016; rmem for slash, talk, topcats at 192 threads; rkdax{2|3} for slash, twitter, and talk (≤ 64 threads).

memory system with separate address spaces. The programmer must then decide where to allocate each data structure and (with PMM) how it is distributed across NUMA domains.

Evaluations on several workloads have shown that Memory mode has about 10% overhead compared to workloads running exclusively on DRAM [13, 40, 51]. Non-graph workloads in AppDirect usually experience a slowdown of 1.5-2× or more [40, 51]. An open question is whether graph applications can exploit AppDirect, enabling maximum memory capacity while achieving better performance. This is challenging due to Optane's longer memory latencies, especially for writes. To exploit AppDirect, the programmer must understand application and Optane characteristics.

We explore how *graph applications with large working sets but different work locality, access patterns, and parallelism can exploit Optane memory*. We study two large-scale graph applications—community detection (Grappolo) [27] and influence maximization (Ripples) [32]—as exemplars of two broader classes of irregular graph applications; namely, iterative graph methods and stochastic diffusion-based graph methods. Both Grappolo and Ripples have parallel implementations for both shared-memory (OpenMP) and distributed-memory (MPI+OpenMP); and both are dominated by irregular memory accesses. However, there are four key differences between these applications—working set, partitioning, access patterns, and synchronization—that are representative of many important graph analytic workloads.

(1) Grappolo's *working set* is dominated by the input graph whereas Ripples' is dominated by a Monte Carlo (MC)-based 'big-data' analysis, with numerous small data structures per MC sample on the input graph.
(2) Grappolo's careful *partitioning* of graph and work distributes tasks such that they have relatively good NUMA locality. In contrast, Ripples' probabilistic graph traversals result in many unpredictable accesses with poor NUMA locality, even if it does have many irregular accesses to local structures.
(3) Grappolo's *accesses* are quite irregular within a task but those to the large graph are fairly regular (and mostly reads). In contrast, both of Ripples' graph and local accesses are irregular.
(4) Finally, Grappolo's iterative algorithm depends on global knowledge and therefore requires atomic and all-to-all *synchronization*. In contrast, Ripples' MC tasks are comparatively asynchronous and can therefore exploit accelerators and scale well on distributed resources.

This combination of shared and unique traits make these two applications ideal for evaluating the efficacy and associated trade-offs of Optane memory for graph analytic workloads.

Our work makes the following contributions.

- We evaluate the suitability of huge-graph analytics—not graph kernels—on single-node systems, compare with distributed executions from CPU-only and GPU-accelerated clusters, and estimate cost/performance. We use large working sets (up to 2.3 TB) on a single system (6 TB Optane + 768 DRAM). Figure 1 summarizes our single-node results.
- We show that AppDirect can perform and scale *better* than Memory (12% for huge graphs), even when dominated by irregular access patterns, as long as most accesses are NUMA-local and Optane accesses are frequently reads. Moreover, careful AppDirect and Memory implementations can *match* DRAM-only.

- Surprisingly, between Memory and AppDirect, processor-cache performance can improve due to fewer line invalidations. Updates to the caching policy (via non-temporal hints) can make a substantial improvement (25%) and remove Memory's performance penalty.
- We demonstrate a performance and programming advantage for AppDirect's new virtual memory interface and NUMA controls. We explain performance through carefully crafted metrics.
- We show frequent price/performance benefits (usually >4–10×) for shared-memory analytics over corresponding executions on GPU-accelerated and CPU-based supercomputers.

This paper is organized as follows. We provide an overview of Optane memory and execution modes (§2). We then discuss our selected graph applications and their characteristics (§3). Our experimental evaluation (§4) discusses microbenchmark results, single-node performance, comparisons with distributed performance, and overall price/performance. Finally, we consider related work (§5) and provide conclusions (§6).

2 INTEL OPTANE PERSISTENT MEMORY

Optane memory modules (PMMs) can be configured into different operating modes that determine the available address space for applications and the overall functionality. There are three Optane memory modes: Memory, AppDirect and Mixed. Figure 2 shows how the three modes differ in how DRAM caches PMM, using the most common provisioning of a socket's DIMM slots with DDR and PMM modules. Table 1 lists the possible Optane configurations from an application view. Mixed mode is now deprecated.

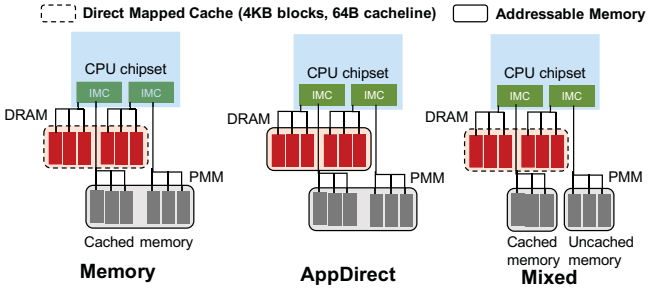


Figure 2: Optane modes: Memory, AppDirect and Mixed.

Configuration	Address space	Description
dram: AppDirect DRAM	DRAM	DRAM is main memory with normal `malloc` interface.
mem: Memory	PMM	PMM is main memory; DRAM is direct-mapped L4 cache
fsdax: AppDirect FS-DAX	DRAM, PMM	DRAM as usual; PMM via file system (classic block device)
kdax: AppDirect KMEM-DAX	DRAM, PMM	Linux >v5.1; PMM via virtual memory (hotplug character device)
Mixed (deprecated)	x, (PMM − x)	x PMM in Memory mode; remaining PMM in AppDirect

Table 1: Possible Optane configurations.

2.1 Memory mode

Memory mode replaces the DRAM volatile capacity with the persistent memory capacity. In this mode, DRAM acts as a direct-mapped write-back cache to the PMM modules connected to the same integrated memory controller (iMC) on a CPU socket; there are two iMC's per socket. Because of this restriction, DRAM cannot cache accesses to PMM on other sockets. If the memory controller is unable to detect the requested data block in the DRAM cache, it brings the data into the DRAM cache from the PMM in specific block sizes (4096 bytes in our case). Thus, a cache hit in the DRAM cache is equivalent to DRAM latency (tens of nanoseconds), whereas a cache miss incurs PMM latency to fetch the requested blocks into the DRAM (hundreds of nanoseconds).

Memory mode is arguably the most productive mode in the context of application development, as it requires no updates to the application code, and is ideal for applications that exhibit high reuse in cache. Persistence is not enabled in this mode, which is only possible through AppDirect mode.

2.2 AppDirect mode

Traditionally, persistent memory systems are accessed using the filesystem like standard block devices. File I/O over disk or flash storage requires maintaining page cache on DRAM, which is redundant for byte-addressable block devices, as reads/writes can be performed directly without any intermediate buffering.

An Optane NVDIMM region is a physical address range that is striped across one or more *namespace* devices. Namespaces are a Linux kernel feature that can partition the system resources (for e.g., DIMMs) among processes such that different sets of processes have their own view of the resources [42]. Hence, it is possible to provision one or more namespaces from an NVDIMM region. The AppDirect mode enables the creation of a Direct-Access (DAX) namespace as a volatile memory address space which enables *direct access* to a DAX character device file (as opposed to a block device) that can send and receive bytes. Two relevant options are:

- **FS-DAX** is the classic AppDirect mode that enables direct access to persistent memory through the file system, by creating a block device. This requires creating namespaces on sockets and mounting one or more file systems. An important limitation in this mode is that namespaces are restricted to the DIMMs mapped to a single socket. To spread allocations across multiple namespaces (sockets) either a striped device must be created (e.g., RAID-0 using `dmsetup`), or a programmer must use distributed-memory data structures.
- **KMEM-DAX** eliminates the need for mounting file systems over namespaces (since it creates a character device). Allocations, *including a single allocation*, can be spread across PMM name spaces without having a backing file passing specific allocation policy to the persistent memory allocator. KMEM-DAX requires a relatively new Linux kernel (>v5.1) to transform the namespaces into hotplugged memory regions seen as special memory-only NUMA nodes that the kernel can use for allocation. The KMEM-DAX option also requires extra space in DRAM for storing page table metadata, which is 64 bytes per 4 KB segment. Optionally, PMM can store the metadata, but for performance we prefer

DRAM. This metadata can easily grow to tens of GBs, for larger namespaces.

We use the `memkind` [43] library to manage allocations on PMM. However, memory allocation is only relevant for KMEM-DAX. In the classic filesystem route (i.e., FS-DAX), since the memory is already available via the file, `memkind` just points to the memory. Using `memkind` in the FS-DAX option can be cumbersome, if multiple filesystems are mounted separately or are logically combined in a RAID-0 configuration through multiple namespaces.

On the other hand, in the KMEM-DAX option, memory is managed by the O/S, and there are policies that can be set via `memkind` to govern the allocation strategies. We use the `MEMKIND_DAX_KMEM_PREFERRED` policy in which each thread allocate its own memory such that the objects each thread allocates are local to it, and it only allocates in the remote (cross-socket/package) PMM node when the memory runs out. Other policies are `MEMKIND_DAX_KMEM` and `MEMKIND_DAX_KMEM_ALL`. For both the cases, memory comes from the closest PMM NUMA node at the time of allocation, but when the current node memory is exhausted, for `MEMKIND_DAX_KMEM` policy, the swap space is used, whereas for the `MEMKIND_DAX_KMEM_ALL` case memory is allocated on another PMM node before spilling to the swap space. We prefer `MEMKIND_DAX_KMEM_PREFERRED` because it enables spanning an allocation across all PMM NUMA nodes.

2.3 Mixed mode

It is possible to configure a system to utilize a percentage of the persistent memory in Memory mode and the rest in AppDirect mode, referred as the Mixed mode. But, since this mode provisions all the DRAM as a direct-mapped last-level cache (as in the Memory mode), PMM capacity is lost. Also, there are some challenges in utilizing this mode, as some systems restricts the ratio of DRAM memory to PMM to be a minimum of 1:4. Also, the performance may suffer as the traffic on the memory bus will increase. As a result, this mode is deprecated and is not investigated.

3 GRAPH APPLICATIONS

In this section, we provide a brief overview of two prototypical graph algorithms used in this work—community detection and influence maximization—focusing on the key computation/algorithmic features that can provide insight on system performance, and providing a contrast between the two algorithms. More details can be found in the respective publications [27, 32].

3.1 Community Detection using Grappolo

Given a graph $G = (V, E, \omega)$, the goal of community detection is to partition the input set of vertices into a set of "communities" (or clusters) such that there is a significantly higher concentration of edges connecting the vertices within a cluster than between vertices across clusters. The problem is typically posed as one of optimizing a quality metric called modularity [35]. Since the underlying optimization problem is NP-Hard [6], efficient heuristics are used in practice [10].

Grappolo [27] is a shared memory multi-threaded implementation of the Louvain heuristic [4]. It is a multi-phase multi-iteration algorithm that begins by assigning each vertex to its own community. Within an iteration, all vertices are visited once. Each vertex scans its local neighborhood to determine if and which of its neighboring community it should join. This locally greedy decision is made by calculating the net modularity gain that results by either staying or migrating to each of the target communities. The algorithm iterates until the net modularity gain achieved between successive iterations falls below a certain threshold. Algorithmically similar to Grappolo are two libraries named **Vite** [12] and **miniVite** [11] that target distributed-memory platforms.

In addition to the graph data structure stored in a compressed sparse row (CSR) format, the algorithm maintains two auxiliary data structures—a vector to maintain the community states of all vertices globally, and a second set of thread-local data structures to store the set of unique communities that are neighbors of each vertex. The purpose of the latter is to help each vertex reach its locally greedy decision by examining its neighboring communities efficiently. It is implemented as a C++ `std::map`, storing the set of unique communities that the neighbors belong to and the number of edges that the vertex has to each of those unique communities. The search combined with allocation and de-allocation of memory makes this operation one of the most expensive operations in Grappolo. Alternative approaches with different memory and performance tradeoffs also exist in Grappolo.

3.2 Influence Maximization using Ripples

Given a directed graph $G = (V, E, \omega)$, a diffusion model, and a budget k, the objective of the *influence maximization* problem is to identify a set of k seed vertices ($S \subseteq V$), which when initially activated, will result in the activation of the maximum expected number of vertices in G at the end of the diffusion process. The edge weight ($\omega(s, t)$) represents the probability of a node s influencing (activating) its neighbor t. A diffusion model is a stochastic process that defines how information originating from a seed set (S) spreads through the rest of the network. However, the classical problem formulation [20] is NP-hard, and various efficient algorithms (including some with approximation guarantees) have been devised [20, 25, 46, 47].

Ripples [32, 34] is a parallel influence maximization implementation that builds on the `IMM` algorithm of Tang et al. [47]. For experiments in this paper, we chose the Independent Cascade (IC) model for diffusion, where there is a one-shot chance for an activated vertex to activate its neighbors. Minutoli et al. [32, 33] show that the IC model is more memory intensive and computationally challenging than the Linear Threshold (LT) model.

Intuitively, the algorithm is a randomized experiment to infer the most impactful source of activation of the vertices in the graph that is based on the concept of *reverse reachablility*. From a randomly chosen vertex v, the algorithm reconstructs the possible "paths of the spread" causing the activation of v by following the rules of the diffusion model going backwards. This process is computed with probabilistic breadth-first searches (BFS) working on the transposed graph. The frontier of one such BFS will constitute a vertex set $\mathbf{R}_v$, a reconstruction of the process resulting in the activation of the given vertex v. The collection of such sets originating from a set of vertices chosen uniformly at random is denoted by $\mathbb{R}$. Building $\mathbb{R}$ consumes the greatest fraction of the computation time. The number of $\mathbf{R}_v$ sets that should be computed, represented by θ, is calculated through

an estimation procedure aimed at finding a lower bound for the influence function. The value of θ is nondeterministic and is nonlinearly proportional to the graph size and some other parameters, which control the submodular approximation guarantee of the computed solution ($1 - 1/e - \epsilon$, for $0 < \epsilon < 1$)[32].

3.3 Characterization & Contrast

Community detection using Grappolo and influence maximization using Ripples are by no means unique, and are representative of two broader classes of graph analytical workloads. Grappolo is a classic example of an iterative graph algorithm that makes multiple sweeps of the graph, each time updating the labels associated with every vertex based on the labels of its neighbors, until a convergence criterion is reached. In the literature, these are also referred to as topology-driven methods [23], and other examples include PageRank, Bellman-Ford shortest path algorithm, and graph coloring. Ripples, on the other hand, is representative of stochastic diffusion-based class of approaches that perform Monte-Carlo simulations on the graph. Numerous complex networks applications involve such simulations [14, 37]. Therefore, characterizing Grappolo and Ripples toward their ability to exploit nonvolatile memory helps us inform and guide future use of the memory technologies on a wider range of irregular applications. Furthermore, the contrasting traits that these two applications embody—in working set, partitioning, access patterns, and synchronization (as elaborated in Section 1)—would help us evaluate and test the various modes and trade-offs associated with single-node partitioned-memory systems.

4 EXPERIMENTAL EVALUATION

This section first discusses our shared-memory platform (§4.1) and microbenchmark evaluations (§4.2). We then evaluate application single-node performance (§4.3) and compare to corresponding distributed performance (§4.4) and cost-performance (§4.5).

4.1 Experimental platform

Our platform has four sockets, each with a SDP (pre-release) Xeon Platinum 8260 (Cascade Lake) CPU (2.3 GHz, 24 cores, 32 KB L1 and 1 MB L2 private caches per core, 33 MB shared L3 cache). Over four sockets, there are 192 hardware threads. Each processor has 6 memory channels with 1 DDR4 and 1 PMM module per channel. The DDR4 modules are 32 GB each, making 768 GB total DRAM capacity. The Optane DIMMs are 256 GB each, making 1.5 TB persistent memory per socket, or 6 TB total. There are 3 links for Intel Ultra Path Interconnect (UPI) per CPU for connecting to other CPU sockets. The platform uses Ubuntu v19.04 O/S, with Linux kernel v5.1.

Grappolo uses OpenMP parallelization and is compiled with Intel compiler v19.0.4.243 with `-O3 -xHost` compilation options. Ripples also uses OpenMP and is compiled using GCC v8.3.0 with the `-O3 -mtune=native` options. The input graphs were obtained from Suitesparse Matrix Collection [22], SNAP datasets [24] and LAW repository [5].

For AppDirect, we focus on KMEM-DAX. Without filesystem overhead, KMEM-DAX provides superior latency and bandwidth over FS-DAX, as will be discussed. Further, we believe KMEM-DAX provides a superior interface for Optane as volatile memory. Memory can be allocated through the virtual memory system, according to memkind's NUMA policies. (Although the policies are currently more constrained than DRAM, there are no analogs in FS-DAX.) Also, FS-DAX mode is cumbersome to use: to create an Optane region that spans multiple sockets, one must create a RAID filesystem cross multiple distinct namespaces (volumes).

We use 1 GB page size in AppDirect KMEM-DAX to minimize potential page faults. We also enable transparent hugepages (THP) [2], using 2 MB pages, and turn off page migration to prevent page movement across NUMA nodes during program execution. We use `memkind` library v1.10.1-rc1 for PMM allocations.

4.2 Microbenchmark evaluation

For both graph analytics application, units of work are typically bound by memory latency of irregular accesses. Therefore, we expect parallel performance to be limited by parallel random access bandwidth. We collect benchmark results showing idle latency and random access bandwidth. For comparison, we also obtain bandwidth of regular accesses (STREAM). We show results across memory modes and (usually) for local and remote accesses.

We assess memory latency and bandwidth of the Optane configurations shown in Table 1 using two microbenchmarks: Intel Memory Latency Checker [49] (MLC) and STREAM [29]. The STREAM benchmark was modified to support allocation on PMM, whereas only the binary version of MLC is available. Particularly, in the AppDirect KMEM-DAX option, the buffer is spread over four namespaces across the NUMA regions (using the `MEMKIND_DAX_KMEM_PREFERRED` `memkind` policy discussed in §2.2).

4.2.1 Sequential and random access bandwidth. The local and remote random access bandwidth is particularly relevant for graph applications. Table 2 shows local and remote (across NUMA nodes) bandwidths for DRAM-only (via AppDirect), Memory mode, and PMM (via AppDirect FS-DAX). We use Intel MLC v3.7 [49]. Since we do not have access to the MLC source code, we were unable to make the requisite changes to use AppDirect KMEM-DAX mode.

Memory (mode)		Read		Write	
	Access	Local	Remote	Local	Remote
DRAM-only (dram)	Seq	103.67	34.3	43.14	31.42
	Random	97.95	34.27	42.16	29.72
Memory (mem)	Seq	100.88	34.29	23.76	18.12
	Random	95.51	34.27	20.7	16.82
PMM (fsdax)	Seq	32.42	32.02	2.79	2.78
	Random	10.44	10.36	3.71	3.63

Table 2: Local/remote sequential/random access B/W (GB/s).

The DRAM-only and Memory modes exhibits at least 3× better bandwidth compared to PMM AppDirect FS-DAX. In general, bandwidth of random access operations is about 10% slower than sequential for the AppDirect DRAM and Memory modes.

4.2.2 STREAM. We updated the STREAM benchmark [29] to allocate the buffers on PMM, in order to compare performances between the different Optane modes we use in the application evaluation. The STREAM benchmark uses three buffers to conduct memory

bandwidth evaluation on four kernels. We set the buffer length to 10^9, and the overall memory usage is about 24GB. We use 192 threads, and each experiment is run for 10 iterations. Figure 3 demonstrates that Memory provides competitive performance to AppDirect DRAM. MLC's TRIAD's results (not shown) agree.

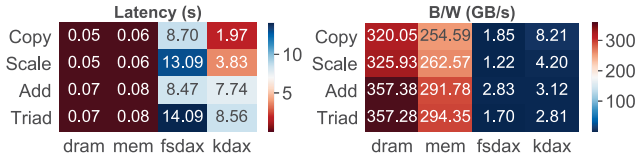


Figure 3: Average latency and bandwidth on updated STREAM for Optane modes (redder is better).

The AppDirect KMEM-DAX and FS-DAX modes exhibits about 40× reduction compared to AppDirect DRAM/Memory. The performance of KMEM-DAX is about 1.1–4.5× better than FS-DAX, as the data is spread across the device namespaces on all NUMA regions. Nevertheless, the next section demonstrates that in full graph applications, KMEM-DAX mode can improve aggregate bandwidth.

4.3 Application Single-node Performance

We analyze several variants of Grappolo and Ripples, over different Optane modes, using a variety of real-world graphs, with working sets (PMM + DRAM) up to 2.3 TB. Our key findings are:

- AppDirect mode can perform and scale better than Memory, even when dominated by irregular access patterns with many writes, as long as most accesses are NUMA-local and Optane accesses are frequently reads.
- Processor-cache latency can be a major contributor to performance degradation, even for executions dominated by random accesses to remote DRAM or PMM.
- Surprisingly, between Memory and AppDirect, processor-cache performance can improve due to fewer line invalidations. Updates to the caching policy, via non-temporal hints for accesses with low temporal locality, can make a substantial improvement and remove Memory's performance penalty.
- Prefer Memory when dominated by random and remote reads and writes with complicated temporal reuse.

The AppDirect variants were not difficult to implement, assuming detailed knowledge of the application's implementation and data accesses.

4.3.1 Performance metrics. The evaluations employ the following metrics, crafted to highlight memory performance and data access locality. They are combinations of raw metrics from the CPU's performance monitoring unit, collected with Intel VTune profiler v2021.1.0 beta10. Our analysis script aggregates and collates the raw data to compare across application variants.

CPU Time (s): Average time (derived from cycles) the CPU is working (per thread). In contrast to wall clock time, this excludes time spent idling, and thus can be significantly smaller due to load imbalance and memory bound operations.

Mem Stalls, Cache Stalls: Cycles that CPU cores were stalled waiting on data from memory and processor-cache (L1...L3), respectively. Stalls represent waste, so the fewer stalls the better. Mem Stalls captures effects of accessing DRAM vs. PMM. Cache Stalls shows differences in cache performance between memory modes.

RFO Cycles: Cycles with at least one read-for-ownership (RFO) transaction (exclusive access to a cache line on a store miss), i.e., cache coherence activity. This metric helps explain why cache performance differs between memory modes.

***L*DRAM, *L*PMM, *R*DRAM+PMM**: Loads that resolve in local (L) or remote (R) DRAM or PMM, or their sum. These enable evaluation of NUMA-related problems such as Memory's DRAM-cache.

4.3.2 Grappolo: Graph Community Detection. We use four graphs of different characteristics for Grappolo evaluations, shown in Table 3. Community detection using Louvain method is multi-phase and there are multiple iterations within a phase (until convergence), but we only execute the very first phase as it is the most expensive and typically represents the full application run.

Graphs	$\|V\|$	$\|E\|$	MAX degree	STDEV degree	#Iters.	Working set
friendster	65.6M	1.8B	5.2K	138	158	48.5 GB
moliere2016	30.2M	3.3B	2.1M	1496	21	128 GB
clueweb12	978.4M	42.5B	75.6M	13575	10	2.16 TB
uk2014	787.8M	47.6B	8.6M	1682	10	2.30 TB

Table 3: Graphs used in Grappolo evaluation.

Variants. The default variants for the different Optane configurations are denoted **gdram** (DRAM-only), **gmem** (Memory) and **gkdax** (AppDirect), respectively.

AppDirect mode requires determining which structures are allocated in DRAM and PMM. Our strategy is designed to obtain *high NUMA locality* by using a vertex-based distribution of parallel work matching the CSR graph format. The target graph is allocated in PMM because it is the dominant contributor to Grappolo's working set and read-only within a phase. We ensure the graph's CSR structure is spread evenly across all sockets. This layout works well when vertices in one NUMA domain rarely access edges in another NUMA domain. Because of its size, we also allocate in PMM a structure of size $|V|$, even though it requires $|E|$ writes. All other auxiliary structures are allocated in DRAM. These are thread local, updated frequently, and should not overwhelm DRAM.

To efficiently allocate a huge input graph in PMM with the desired NUMA distribution, we had to restructure the graph parser and allocation strategy. The original graph parser assumed the entire graph could be stored within DRAM in an interchange format and then converted into Grappolo's Compressed Sparse Row (CSR) from within DRAM. When graphs exceeded available DRAM, the OS's virtual memory system used costly page-coalescing [19] within Linux's default buddy allocator [21]; cf. [7]. We changed Grappolo to read an input CSR graph incrementally, using a fixed-size DRAM buffer.

We developed an optimized variant, denoted as **gdram2**, **gmem2** and **gkdax2**, to improve processor cache performance. The variant adjusts the processor caching policy in two locations involving access patterns with zero temporal locality over the graph's CSR structure. To adjust the policy, we applied non-temporal hints using the pragma `prefetch var:3`. This generates `prefetchnta`, which prioritizes the new line for eviction, avoiding pollution.

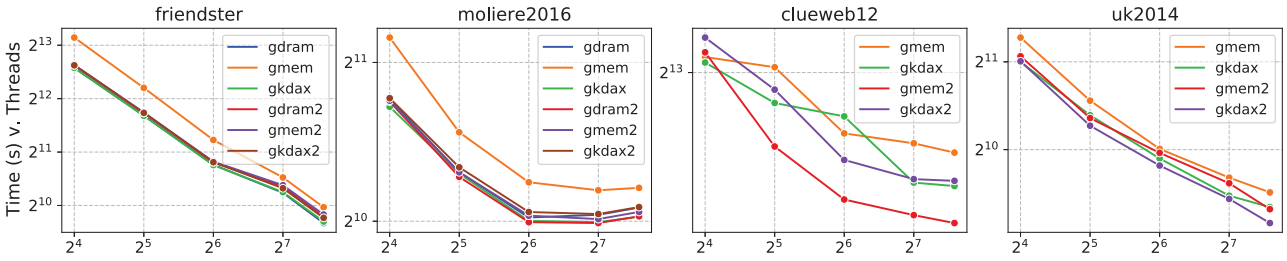


Figure 4: Grappolo: Strong scaling (OpenMP threads) across application variants and memory modes.

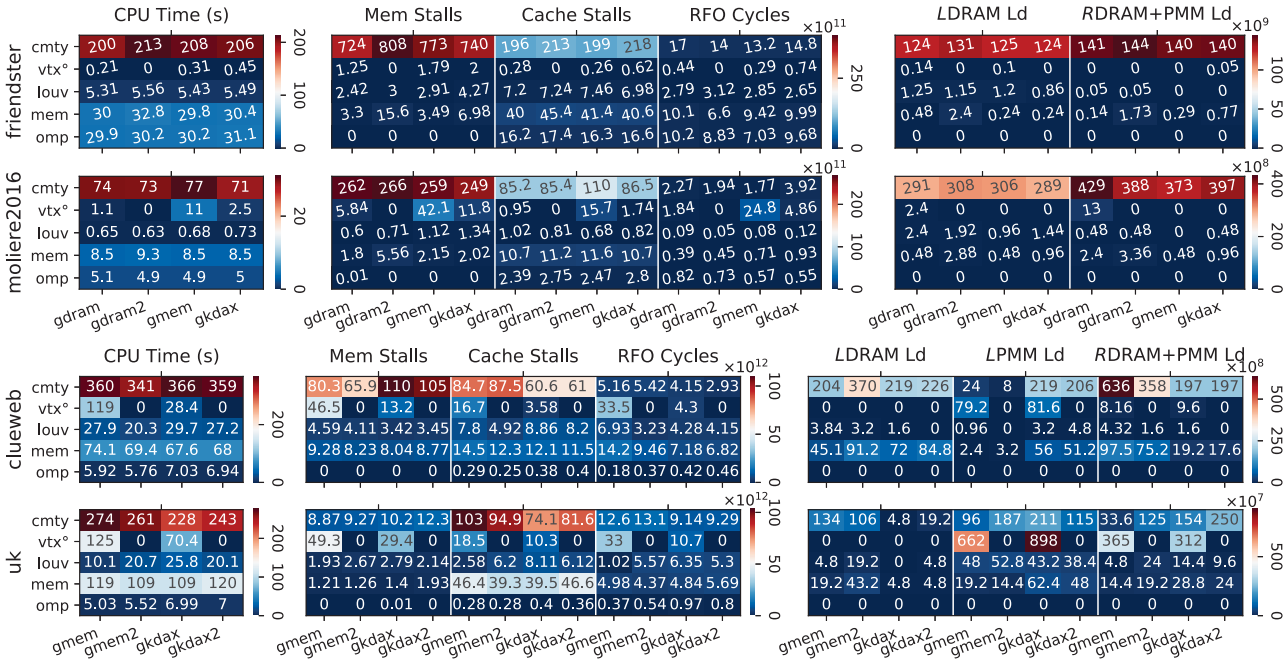


Figure 5: Grappolo: Memory metrics (columns) at 192 threads, per graph (rows), attributed to application variants and function group. Each heat map's values are scaled by the exponent over the color bar.

Execution configuration. We size AppDirect memory to ensure PMM allocations are spread over the NUMA regions. We ensure work and allocations are consistent with NUMA's first-touch policy and use `memkind`'s `MEMKIND_DAX_KMEM_PREFERRED`. We execute with OpenMP's `PLACES=cores`, `BIND=spread`.

Results overview. Figure 4 shows Grappolo's scaling behavior, for both variants, across graphs and memory modes; cf. Fig. 1 for relative performance. (Times *exclude I/O* because it is not parallel, bottlenecked by a single SSD, and consumes more than 50% of total time for very large input graphs.) Figure 5 shows memory metrics, attributed to function group, for 192 threads. Function groups are as follows. Each Louvain phase calculates vertex degrees (`vtx°`) and invokes a parallel loop over vertices (`louv`). The main hotspot is `cmty`, which inspects neighbors of a vertex to determine its own best community. It is dominated by a thread-local, DRAM-allocated C++ `std::map` to identify unique neighbor communities. OpenMP overhead (locks, barriers) is represented by `omp`. Memory management (malloc, free) is `mem`.

Our expectations are that scaling trends are affected first by graph, somewhat by mode, and only modestly by variant. The input graph is significant because load balance is input-dependent: since each vertex scans its neighborhood communities, the graph's vertex-degree distribution determines overall work distribution. Mode should have an effect due to memory performance characteristics and allocation strategy. However, it is not clear that AppDirect can improve over Memory in the context of a full application with all cores active. We do expect the optimized variant to improve over the default variant, but only expect a modest effect.

Scaling. The strong-scaling plots (Fig. 4; cf. Fig. 1) show the following expected behavior. For the default variant and a given graph, scaling trends are qualitatively similar, possibly excepting clueweb12. For the default variant, Memory is good default option but is always more costly than DRAM (when applicable).

For clueweb12, with the default variant and Memory (gmem), scaling suffers at higher thread counts. clueweb12's graph characteristics — a much higher standard deviation for vertex degree

(Table 3) — suggests that remote access can occur far more frequently than with other graphs. Comparing remote accesses (same memory mode) between uk2014 and clueweb12 (*R*DRAM+PMM, Fig. 4), validates this hypothesis. *Thus, graph-structure statistics can predict the likelihood of remote acceses.*

The scaling plots also show unexpected behavior. AppDirect mode *improves over memory mode by 12%* for clueweb12 and uk2014. The optimized variant can be surprisingly good (for clueweb12, gmem2 is 25% faster than gmem) and can eliminate the gap between Memory and DRAM-only (gmem2 on friendster, moliere-2016). For the same graph, scaling trends can qualitatively *change* across variants (gmem2 for clueweb12). We discuss these results below.

Memory. As expected, memory mode is a good default by enabling very large working sets and reasonable scaling without any application changes. However, absolute performance is sacrificed compared to DRAM only (Fig. 4, friendster and moliere2016). The gap over DRAM is strongly connected to the effectiveness of the DRAM cache. First, a load's memory latency can increase due to the additional step of caching PMM. This is probably why `cmty`'s Mem Stalls increase (Fig. 4) between DRAM-only (friendster/moliere2016) and Memory (same graph). Second as NUMA-local, the cache only benefits accesses to *local* PMM. Finally, as direct-mapped, cache effectiveness can decrease with scale: with more threads, more distinct data is accessed concurently, increasing the probability that there will be a cache conflict.

The effects of DRAM cache policies are seen in Fig. 5. Cache effectiveness is indicated by comparing *L*DRAM and *L*PMM. For medium graphs (friendster, moliere), the entire PMM working set can be cached in DRAM, so *L*PMM is negligable. Huge graphs exercise the DRAM cache. For clueweb12, `cmty`'s ratio of accesses to local PMM is much smaller than with uk2014, corresponding to the actual (total) DRAM cache hit ratios of 47% and 18%, respectively. clueweb12's poorer DRAM-cache performance is rooted in a much larger ratio of remote accesses (*R*DRAM+PMM), none of which DRAM can cache. Thus, *Memory mode is particularly sensitive to input graph and layout* when memory accesses become NUMA-remote.

Memory mode also shows unexpected behaviors, such as significant improvement (25%) for the optimized variant (gmem2) for clueweb12. Observe that that *processor-cache stalls* (Cache Stalls) — not just memory stalls (Mem Stalls) — can be a significant contributor to performance, especially for large graphs. Depending on the graph, the *importance of bottlenecks due to memory stall vs. processor-cache stalls changes.* In uk2014, Cache Stalls dominates Mem Stalls, whereas in clueweb12 Mem Stalls just edges Cache Stalls. This effect is discussed below.

AppDirect (KMEM-DAX). We now disuss the two most surprising results. The first is that AppDirect mode can outperform other modes. For medium graphs (friendster, moliere), AppDirect variants match or exceed performance of DRAM and Memory modes. Meeting DRAM is unexpected because AppDirect has more memory resources than in DRAM mode: as our experiments are conducted on one platform, AppDirect mode uses 2 modules per channel (DRAM + PMM) vs. 1 module (DRAM + no PMM). It is particularly notable that, for the default variant, *AppDirect usually meets the best performer across all modes and all graphs.* For huge graphs (clueweb12 and uk2014), gkdax outperforms gmem by up to 12% (even though NUMA accesses are relatively poor). AppDirect's performance can be explained by a surprising *improvement in processor-cache performance* (Cache Stalls). These metrics show that `cmty`'s and `vtx°`'s AppDirect processor-cache performance is always better than in Memory mode, though its importance relative to memory stalls changes between uk2014 (higher) and clueweb12 (lower).

The second surprise is the contrasting effect of our optimized variant, which we created while probing the differences in processor-cache performance. The differences were perplexing because access patterns remained the same across modes even if allocation location differed. Further, processor-cache policies do not change with mode. In particular, in Memory, a DRAM conflict miss *should not* automatically invalidate a line representing local or remote PMM.

We found that Memory can cause more coherence activity. The metric RFO Cycles shows cycles in which read-for-ownership (RFO) requests are present, which correlates extremely well to the relative change in processor-cache performance. More RFOs means more lines are in a 'shared' state, resulting in more invalidation broadcasts, wasting cache bandwidth. An analysis of access patterns identified two locations where graph structure is read — a very large memory footprint — with zero temporal locality. We adjusted the caching policy using non-temporal prefetch hints. This change resulted in a significant reduction in RFO Cycles for the optmized variant (gkdax2 and gmem2), as well as a reduction in the number of loads that resolve in local PMM (*L*PMM) or any remote socket (*R*DRAM+PMM). The relative benefits for `vtx°` and `cmty` also make sense. `vtx°` streams through the CSR graph ($|E|$) and writes to a $|V|$-sized structure. In contrast, `cmty` is dominated by random-access reads/writes as it slowly streams through the CSR graph. Thus, `vtx°` benefits significantly but `cmty` only modestly. The slight gdram2 slowdown on friendster may be due to prefetches stealing bandwidth from demand loads (cf. Mem Stalls and *L*DRAM).

4.3.3 Ripples: Graph Influence Maximization. Our evaluation uses five graphs of various sizes and characteristics (Table 4).

Graphs	$\|V\|$	$\|E\|$	MAX degree	STDDEV degree	Working set
soc-*Slash*Dot0902	82.1K	948.46K	2.5K	37.4	17.5 GB
soc-*twitter*-combined	456.62K	14.85M	51.2K	346.5	21.6 GB
wiki-*talk*	2.39M	5.02M	3.3K	11.8	41.0 GB
soc-*pokec*	1.63M	30.62M	13.6K	31	590 GB
wiki-*topcats*	1.79M	28.51M	202.2K	203.9	785 GB

Table 4: Graphs used in Ripples evaluation.

The algorithm consists of a large number of randomized BFSs that build a collection of *random reverse reachable sets* (RRR) that are then used during the seed selection process (see §3.2). Table 4 shows the number of sets (θ) in the RRR set collection and the final working set size when $\epsilon = 0.13$ and $k = 100$. Although θ is a good proxy for the work performed in Ripples, its scalability is not a simple function of θ: performance is greatly influenced by the

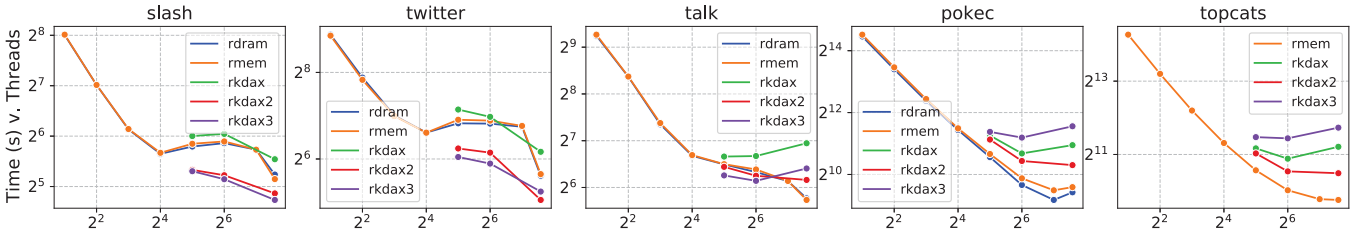


Figure 6: Ripples: Strong scaling (OpenMP threads) across application variants and memory modes.

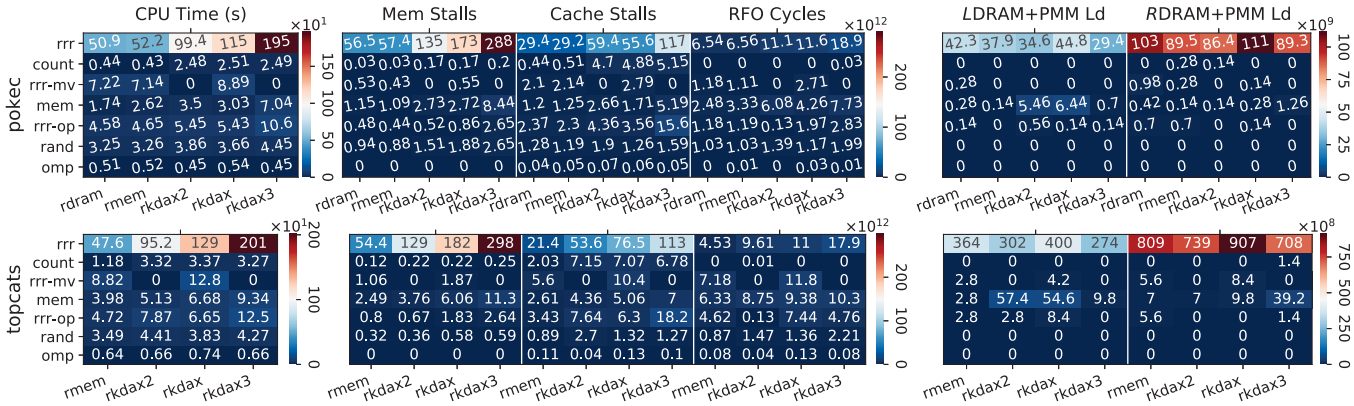


Figure 7: Ripples: Memory metrics (columns) at 64 threads, per graph (rows), attributed to memory mode and function group. Each heat map's values are scaled by the exponent over the color bar.

irregular and non-local memory accesses over the input graph (e.g., randomized BFSs). We denote as `rrr` the functions associated with building RRRs, the hotspot.

Variants. The default DRAM-only and Memory variants are denoted **rdram** and **rmem**, respectively.

The characteristics of Ripples suggest a partitioning of data structures between DRAM and PMM. Recall that the dominant contributors to working set (Table 4) are the RRR sets and therefore candidates for PMM. In contrast to RRR sets, the input graph has a small memory footprint and is frequently accessed from random starting points, which makes it a good candidate for DRAM.

The three AppDirect variants change the allocation strategy used by `rrr`'s two steps. `rrr` first performs randomized BSFs on the input graph starting from a random source to generate an RRR set. The second step sorts RRR sets by vertex IDs [32]. The three variants are as follows: i) **rkdax** allocates the memory for the RRR sets diretly in PMM and performs the BFS on the graph stored on DRAM. The sorting algorithm uses a temporary buffer allocated in DRAM; ii) **rkdax2** is as rkdax but the final sorting step leverages an in-place sort and, therefore, sorting occurs directly in PMM; iii) **rkdax3** is as rkdax2 but RRR sets are staged and sorted in DRAM and then written to PMM in a single move.

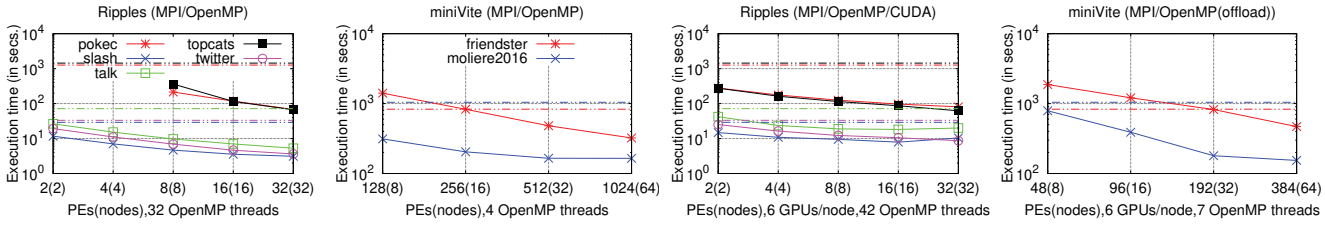
Execution configuration. The later stages of the algorithm that use RRR sets exhibit random data-dependent non-local accesses. We minimize average NUMA distance using an interleaved memory-allocation policy. For DRAM and Memory we use NUMA "interleave" (via Linux `numactl`). For AppDirect, we use memkind's new `KMEM_DAX_INTERLEAVE`. We execute with `PLACES=sockets`, `BIND=close`.

Results overview. Figure 6 shows Ripples' scaling behavior across graphs and memory modes; cf. Fig. 1 for relative performance. (Times *exclude I/O.*) Figure 7 shows memory metrics, attributed to function group, for 64 threads. The primary function group is `rrr` (building RRRs). Other function groups represent auxiliary operations and data movement. As with Grappolo, `omp` and `mem` represent OpenMP and memory management overhead, respectively.

Due to many random NUMA-remote accesses, we expect DRAM to have the best performance and Memory to have modest degradation over DRAM. However our scaling plots show that Memory is surprisingly close to DRAM. We expect AppDirect variants to be considerably slower than the other two modes. Interestingly, AppDirect can significantly outperform Memory and DRAM (50%) for smaller working sets.

We expect rkdax3 to provide the best AppDirect performance because of minimizing random writes to PMM (by aggregating in DRAM). Further with rkdax3, memory intensive operations (randomized BFSs, parallel independent graph traversals, sorting) all use DRAM's random-access bandwidth, which is better than PMM (see §4.2). Surprisingly, rkdax2 is the best variant.

Scaling trends and Memory mode. Figure 6 shows our scaling results. Trends show that Ripples' scaling depends on input. For the smaller inputs (slash and twitter), there is insufficient work at larger thread counts. In contrast, larger inputs offer more parallel work and show better scaling, but they become eventually *limited by remote random access memory bandwidth* (e.g., pokec at 128 threads); cf.



(a) NERSC Cori performance compared to single-node Optane. **(b) OLCF Summit performance compared to single-node Optane.**

Figure 8: Comparing distributed-memory performance for Ripples and Grappolo (as miniVite) with AppDirect (DRAM + PMM). Horizontal lines are based on best AppDirect configuration.

Table 2. Scaling trends and execution times between DRAM-only and Memory are very close. Again, the reason is that Memory and DRAM have very similar (local and remote) random access read bandwidth (Table 2). (Note similar values for Mem Stalls and Cache Stalls.) Interestingly, AppDirect scaling is often qualitatively different from the other modes, ranging from substantially better to worse, depending on input size and variant, as we discuss below.

AppDirect (KMEM-DAX). As expected, for all memory modes, NUMA-remote loads (*R*DRAM+PMM) dominate NUMA-local loads. We expected rkdax3 to be the best AppDirect variant because memory-intensive irregular operations (randomized BFSs, parallel independent graph traversals, sorting) all use DRAM, which has better random access bandwidth than PMM (Table 2). However, this causes an extra memory copy (to PMM) after staging RRR in DRAM. The result in an *overall increase in (write) memory activity when remote access bandwidth is already saturated.* Figure 7 shows that rkdax3 has the highest number of Mem Stalls for `rrr` on both pokec and topcats. By this measure, rkdax3 is up to 1.66× worse than rkdax and up to 2.31× worse than rkdax2 on pokec and topcats.

For large working sets (pokec and topcats), Memory provides better performance than AppDirect. Overall, AppDirect shows substantial increase in memory stalls (Mem Stalls) and processor-cache performance (Cache Stalls). The increases in Cache Stalls are closely correlated with requests for ownerships (RFOs, RFO Cycles) generated by the cache coherence protocol. Among the AppDirect variants, rkdax2 is consistency better than rkdax1 and rkdax3.

An open question is whether there is an effective technique for improving processor-cache performance, such as increasing work-data locality or adjusting cache policies, similar the changes for Grappolo's **gkdax2**. A simple solution is not obvious. A hardware solution would be for Memory's DRAM to cache non-local data, but that would increase complexity of memory controllers.

For smaller input graphs, AppDirect's rkdax2 and rkdax3 variants are most beneficial. Although an interesting counterpoint, we do not comment further because the larger working sets are more interesting and the metrics are not of sufficiently high quality. Metrics are gathered by time-based multiplexing of many event sets, and the run times at larger thread counts are too short.

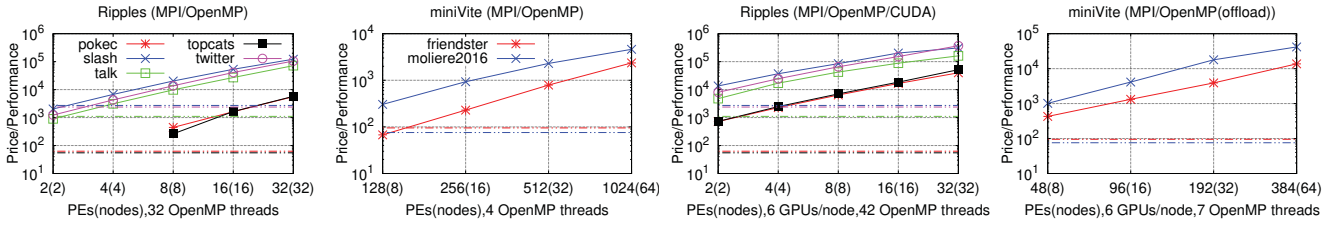
4.4 Distributed Performance

To quantify tradeoffs of *partitioned shared-memory* vs. distributed-memory machines for our graph applications, we compare our single-node AppDirect executions with high quality MPI-based distributed implementations. Ripples has a distributed-memory implementation [32] with support for multi-GPU systems [33]. For Grappolo, we use miniVite [11] from the ECP Proxy Application suite [41], which is algorithmically and structurally equivalent to Grappolo. For distributed-memory machines, we use the NERSC Cori and OLCF Summit supercomputers, which are CPU-based and GPU-accelerated, respectively.

Recall that the two applications were selected because they exhibit several opposing characteristics. Grappolo/miniVite has fairly good NUMA locality, whereas Ripples illustrates poor locality. Grappolo/miniVite suffers from synchronization overheads, whereas Ripples is highly asynchronous and can effectively utilize accelerators due to its embarassingly parallel nature.

Figure 8 shows strong scaling for each application and graph on each supercomputer. Each plot shows per-graph comparisons (horizontal lines) representing the partitioned configuration (i.e., AppDirect in FS-DAX mode using DRAM and PMM) with the best average times. Specifically, for Grappolo, we use gkdax at 192 threads; and for Ripples we use rkdax2 at 192 threads. Observe that these configuations are not necessarily the best times with respect to a particular graph or thread count. The cross-over point between the single-node and distributed curves indicates the number of distributed nodes needed to equal single-node performance. Overall, it usually takes several distributed nodes to match big-memory single-node performance.

Ripples greatly benefits from distributed execution because of its asynchrony and the higher aggregated memory bandwidth available. In distributed settings, the input graph is replicated on the allocated nodes. This strategy is possible because the memory requirements of the application are dominated by the RRR sets and the input graph is just a minor fraction of it. However in these settings, Ripples' seed selection process becomes critical for scaling. The seed selection algorithm is a greedy sequential decision process that on distributed systems requires all-reduce communication. The work of Minutoli et al. [33] has enabled the application to improve scaling and leverage GPUs when available on the system. Ripples benefits over a corresponding single-node execution *as soon as the RRR working set fits* for both CPU-only and GPU-accelerated systems (see Fig. 8). In our settings, two nodes are always sufficient on smaller graphs and on the bigger graphs (Topcats and Pokec) on a system like the OLCF Summit (see Table 5). However, Ripples requires at least 8 nodes for Topcats and Pokec on a more traditional HPC cluster configuration like the NERSC Cori.



(a) NERSC Cori Price/Performance. **(b) OLCF Summit Price/Performance.**

Figure 9: Comparing price/performance of distributed-memory vs.. AppDirect (DRAM + PMM) for Ripples and Grappolo (as miniVite). Horizontal lines are based on best AppDirect configuration. Distributed costs use invoice and exclude network.

In contrast, Grappolo/miniVite can require 16–32 nodes (cf. friendster) to show improvement over a single node with Optane memory. Further, miniVite's distributed-memory performance is more dependent on the underlying process graph topology and graph distribution. miniVite uses ghost vertices to represent remote edges. *Because of the memory required for ghost vertices, miniVite requires at least 8 nodes to execute what are merly medium-sized graphs for our single-node system.* At scale, severe load imbalances occur due to irregularities in process topology. In contrast, Ripples replicates a graph across processes, without incurring the overhead of graph distribution (graph partitioning is an NP-hard problem).

4.5 Overall Price and Performance

We now compare price/performance for single-node AppDirect (DRAM + PMM) and distributed executions. As before, performance is execution time. Table 5 shows pricing ($) for DDR-only and DDR+Optane systems as well as nodes similar to the NERSC Cori and OLCF Summit supercomputers. The $-price estimates are obtained from multiple system vendors. For a single system, DDR+Optane configurations provide the best price-per-memory.

Memory Configuration	Total Memory	Price	Price/ Memory-GB
4x Intel Xeon Platinum 8260L (Cascade Lake) platform [ca. 2021]			
48 x 64 GB DDR4	3 TB DDR4	42,115 msrp	14
48 x 128 GB DDR4	6 TB DDR4	87,787 msrp	14.6
24 x 128 GB Optane, 24 x 32 GB DDR4	3 TB Optane + 768 GB DDR4	43,225 msrp	11.5
24 x 256 GB Optane, 24 x 64 GB DDR4	6 TB Optane + 1.5 TB DDR4	79,075 msrp	10.5
Distributed-Memory configurations			**Platform details**
12 x 32 GB DDR4	192 GB DDR4	11,872 invoice	2x Intel Xeon 6126 (Sky Lake) [ca. 2017]
16 x 32 GB DDR4	512 GB DDR4	13,115 msrp	2x Intel Xeon Gold 6258R (Cascade Lake) [ca. 2021]
16 x 32 GB DDR4, 6 x 16 GB HBM2, 1.6 TB NVMe Flash	512 GB DDR4 + 96 GB HBM2 + 1.6 TB NVMe	143,239 msrp 100,267 invoice	2x IBM Power9 AC922 with 6 Nvidia V100 GPUs [ca. 2021]

Table 5: Price estimates in USD (from multiple system vendors). Top: Shared-memory with near-iso memory capacity; highlight is close to our configuration (6 TB Optane, 768 GB DDR4). Bottom: Distributed memory platforms.

We use the estimates in Table 5 to plot price/performance in Fig. 9. We bias our prices in favor of the distributed-memory platforms by excluding network switching and I/O subsystem costs and using *invoice* (not MSRP) price. Since NERSC Cori uses a discontinued CPU platform (i.e., Intel Haswell), we use the invoice price of a similar Intel SkyLake platform node for calculating price/performance. Recall that Fig. 8 demonstrates better performance of Ripples and miniVite/Grappolo for distributed-memory configurations. *When considering price/performance, the assessment radically alters.*

Figure 9 demonstrates substantial price/performance benefits for single-node Optane systems. For Ripples' larger working sets (i.e., pokec and topcats), an Optane-based node demonstrates 4–13× improvement at the minimum sufficient nodes (8). For the smaller working sets, a single-node always benefits over the GPU cluster by at least 3×; for a CPU cluster, it benefits at 4 nodes (about 2×). For Grappolo/miniVite, a single-node benefits by 4–10× at 8 nodes, with the exception of friendster on Cori, which has a 1.4× degradation. After 8 nodes, a single-node always benefits by *at least* 2.5×. For GPU-based nodes, a single-node always benefits by *at least* 4.5×, due to those nodes' significantly higher cost. We note that these benefits may not be true for other kinds of applications.

5 RELATED WORK

The closest areas of related work are explorations of Optane on graph analytics (below) and management of heterogeneous memories. As heterogeneous memories become more common, systems (e.g., [1, 31]) will incorporate support for managing multiple address spaces, but may well expose placement hints for performance. Our work is the first to beneficially exploit AppDirect's partitioned address space for graph applications. Similar to our use of non-temporal hints to avoid cache pollution, the runtime of Alvarez et al. [1] tracks single-use data objects.

Optane and graph analytics. Peng et al. [40] discuss the performance of Optane on a dual-socket node with 96 threads in the context of five graph workloads (of sizes 35–270G) taken from the GAP benchmark [3] and Ligra framework [45]. The authors observe that Memory mode performs 2–18× better than AppDirect modes, but the performance diminishes with larger problem sizes. We perform extensive performance analysis using two graph applications of different computational characteristics, and problem sizes of up to 3 TB on a four-socket system, demonstrating that the AppDirect KMEM-DAX mode can provide better or competitive performance to DRAM and Memory modes for most of the cases (see Fig. 4 and Fig. 6). Gill et al. [13] investigates the effect of NUMA-aware memory allocation for a number of graph kernels from Galois [36] and GraphIt [56] frameworks, and GBBS [8] and GAP [3] benchmarks

on a two-socket Optane system (96 threads), using large graphs. They report competitive performance of Optane Memory mode compared to DRAM for a variety of graph benchmark kernels. In contrast, we also employ the AppDirect modes, notably the recent KMEM-DAX option that uses O/S virtual memory interface, and show competitive performance to rest of the modes. The AppDirect modes maximizes the memory capacity, since both DRAM and the PMM are available to the application as addressable memory.

Optane and HPC workloads. There are a number of existing works on the application of Optane persistent memory in HPC workloads. An early evaluation of Optane DC persistent memory module and its impact on high-performance scientific applications is discussed in [51], in the context of accelerating I/O through the AppDirect mode, as writing to PMM is faster than file I/O on parallel file system. Using Optane as a block device for performing I/O in the context of MPI I/O and POSIX I/O is explored in [52]. In [39] and [38], the authors evaluate multiple HPC applications with relatively small footprints, exhibiting similar performance in AppDirect DRAM and Memory modes. Christgau et al [7] are able to improve portions of a multigrid solver in AppDirect mode, but not the overall time. Recent work has explored dynamic data migration between DRAM and PMM both at the runtime [26] and OS levels [9].

Optane and database analytics. In database analytics, Optane can be beneficial when favorable access patterns can be exploited. Designing a key-value store for read-dominated accesses can accelerate algorithms using memoization or dynamic programming [53]. However, when using persistent memory to perform database joins with random accesses, AppDirect mode yields slowdowns [44, 50, 55]. Interestingly, Shanbag et al. [44] shows that if the hash table fits in the private L2 or shared L3 cache, there is no performance difference between DRAM and AppDirect mode for random accesses, as the memory boundedness is relatively low. This is similar to our observation that improved cache locality helps to bridge the gap between DRAM and PMM performance, as shown in our application analysis (§4.3).

Optane benchmarking. There is a variety of research discussing basic performance of Optane systems through empirical analysis of benchmarks, data structures and database management systems [15, 18, 48, 54]. The worse-case PMM bandwidth and latencies relative to DRAM as reported in these papers match with our benchmarking analysis results (§4.2).

6 CONCLUSIONS AND FUTURE WORK

We evaluate the suitability of huge-graph analytics (working sets up to 2.3 TB) on single-node systems with Optane as a volatile pool (6TB Optane + 768 DRAM). We study two large-scale graph applications — Grappolo [27] for community detection and Ripples [34] for influence maximization — with different locality, access patterns, and parallelism. We compare application variants for single- and partitioned-address spaces or Memory and AppDirect modes. Finally, we compare against distributed executions from CPU-only (NERSC Cori) and GPU-accelerated (OLCF Summit) supercomputers and estimate cost/performance.

Single-node huge-memory systems can provide compelling price-performance ratios for graph analytics. Although distributed executions eventually show better performance, the price-performance of a single-node Optane system is usually at least 4–10× better. Further a single-node Optane demonstrates reasonable performance and scaling for huge graphs; whereas it can take 8 nodes simply to execute *medium* graphs.

We demonstrate that huge-memory partitioned-address graph analytics are feasible and can achieve good performance. We show that Optane's AppDirect mode can perform and scale *better* than its Memory mode (12% for huge graphs), even when dominated by irregular access patterns, as long as most accesses are NUMA-local and Optane accesses are frequently reads. We also show that AppDirect's new virtual memory interface, which enables much more flexible PMM allocations compared to the traditional file-system interface, also has a performance advantage.

We identify surprising performance phenomena. Between Memory and AppDirect, processor-cache performance can improve due to fewer line invalidations. Updates to the caching policy (via non-temporal hints) can make a substantial improvement (25%) and even remove Memory's typical performance penalty.

There are broader lessons to our work. Many graph applications use algorithms that are similar to Grappolo's or Ripple's work locality, access patterns, and parallelism. For each application class, our study shows expectations for cost and performance trade-offs. Assuming detailed knowledge of application characteristics and implementations, our AppDirect variants were not difficult to implement. Our work also relates to the emergence of heterogeneous memories through HBM and scratchpads. We are particularly interested in how applications can exploit the different access and performance properties of these memory regions.

ACKNOWLEDGMENTS

We thank Joe Oster, Eduardo Berrocal, Vineet Singh, Pallavi Mehrotra, Steve Scargall, and Kelly Lyon, all of Intel, for helpful discussion.

This research is supported by the U.S. Department of Energy (DOE) through the Office of Advanced Scientific Computing Research's "Integrated End-to-end Performance Prediction and Diagnosis", the Exascale Computing Project (17-SC-20-SC) (ExaGraph), and the Data-Model Convergence (DMC) initiative at Pacific Northwest National Laboratory.

We are grateful for access to computational resources at NERSC and OLCF, which are supported by DOE under Contract Nos. DE-AC02-05CH11231 and DE-AC05-00OR22725.

Pacific Northwest National Laboratory is operated by Battelle for the DOE under Contract DE-AC05-76RL01830.

REFERENCES

[1] Lluc Alvarez, Marc Casas, Jesus Labarta, Eduard Ayguade, Mateo Valero, and Miquel Moreto. Runtime-guided management of stacked dram memories in task parallel programs. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, pages 218–228, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357838. doi: 10.1145/3205289.3205312. URL https://doi.org/10.1145/3205289.3205312.

[2] Andrea Arcangeli. Transparent hugepage support. In *KVM forum*, volume 9, 2010.

[3] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[4] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008. URL http://stacks.iop.org/1742-5468/2008/i=10/a=P10008.

[5] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[6] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Gorke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. On modularity clustering. *IEEE transactions on knowledge and data engineering*, 20(2):172–188, 2007.

[7] Steffen Christgau and Thomas Steinke. Leveraging a heterogenous memory system for a legacy fortran code: The interplay of storage class memory, dram and os. In *Proc. of the 2020 IEEE/ACM Workshop on Memory Centric High Performance Computing*, 2020. doi: 10.1109/MCHPC51950.2020.00008.

[8] Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E Blelloch, and Julian Shun. The graph based benchmark suite (gbbs). In *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, pages 1–8, 2020.

[9] Thaleia Dimitra Doudali, Daniel Zahka, and Ada Gavrilovska. Cori: Dancing to the right beat of periodic data movements over hybrid memory systems. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 350–359, 2021. doi: 10.1109/IPDPS49936.2021.00043.

[10] Santo Fortunato. Community detection in graphs. *Physics reports*, 486(3-5): 75–174, 2010.

[11] Sayan Ghosh, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, and Assefaw 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)*, pages 51–56. IEEE, 2018.

[12] Sayan Ghosh, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, Hao Lu, Daniel Chavarrià-Miranda, Arif Khan, and Assefaw Gebremedhin. Distributed louvain algorithm for graph community detection. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 885–895, 2018. doi: 10.1109/IPDPS.2018.00098.

[13] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. Single machine graph analytics on massive datasets using intel optane dc persistent memory. *Proc. VLDB Endow.*, 13(8):1304–1318, April 2020. ISSN 2150-8097. doi: 10.14778/3389133.3389145. URL https://doi.org/10.14778/3389133.3389145.

[14] Adrien Guille, Hakim Hacid, Cecile Favre, and Djamel A Zighed. Information diffusion in online social networks: A survey. *ACM Sigmod Record*, 42(2):17–28, 2013.

[15] Satoshi Imamura and Eiji Yoshida. The analysis of inter-process interference on a hybrid memory system. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region Workshops*, pages 1–4, New York, NY, USA, 2020. Association for Computing Machinery. doi: 10.1145/3373271.3373272.

[16] Intel. Intel Debuts Cooper Lake Xeons for 4- and 8-Socket Platforms. https://www.hpcwire.com/2020/06/18/intel-debuts-cooper-lake-xeons-for-4-8-socket-platforms/, June 2020.

[17] Intel. Taking a deep dive into Cooper Lake xeon sp processors. https://www.nextplatform.com/2020/06/18/taking-a-deep-dive-into-cooper-lake-xeon-sp-processors/, June 2020.

[18] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[19] Mark S Johnstone and Paul R Wilson. The memory fragmentation problem: Solved? *ACM Sigplan Notices*, 34(3):26–36, 1998.

[20] David Kempe, Jon M. Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 24 - 27, 2003*, pages 137–146. ACM, 2003. doi: 10.1145/956750.956769.

[21] Kenneth C Knowlton. A fast storage allocator. *Communications of the ACM*, 8 (10):623–624, 1965.

[22] Scott P Kolodziej, Mohsen Aznaveh, Matthew Bullock, Jarrett David, Timothy A Davis, Matthew Henderson, Yifan Hu, and Read Sandstrom. The suitesparse matrix collection website interface. *Journal of Open Source Software*, 4(35):1244, 2019.

[23] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Parallel graph analytics. *Communications of the ACM*, 59(5):78–87, 2016.

[24] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[25] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne M. VanBriesen, and Natalie S. Glance. Cost-effective outbreak detection in networks. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Jose, California, USA, August 12-15, 2007*, pages 420–429. ACM, 2007. doi: 10.1145/1281192.1281239.

[26] Jiawen Liu, Dong Li, Roberto Gioiosa, and Jiajia Li. Athena: High-performance sparse tensor contraction sequence on heterogeneous memory. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '21, pages 190–202, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383356. doi: 10.1145/3447818.3460355. URL https://doi.org/10.1145/3447818.3460355.

[27] Hao Lu, Mahantesh Halappanavar, and Ananth Kalyanaraman. Parallel heuristics for scalable community detection. *Parallel Computing*, 47:19–37, 2015.

[28] Andrew Lumsdaine, Douglas P. Gregor, Bruce Hendrickson, and Jonathan W. Berry. Challenges in parallel graph processing. *Parallel Process. Lett.*, 17(1): 5–20, 2007. doi: 10.1142/S0129626407002843. URL https://doi.org/10.1142/S0129626407002843.

[29] John D McCalpin. Stream benchmark. *Link: www. cs. virginia. edu/stream/ref. html# what*, 22, 1995.

[30] Chris Mellor. Why Micron fears Optane will eat its server DRAM lunch. https://blocksandfiles.com/2019/09/29/optane-pricing-micron-dram-headache/, September 2019.

[31] MemVerge. Memory Machine. https://memverge.com/wp-content/uploads/2020/10/Data-Sheet_Memory-Machine.pdf, may 2021.

[32] Marco Minutoli, Mahantesh Halappanavar, Ananth Kalyanaraman, Arun Sathanur, Ryan Mcclure, and Jason McDermott. Fast and scalable implementations of influence maximization algorithms. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12. IEEE, 2019.

[33] Marco Minutoli, Maurizio Drocco, Mahantesh Halappanavar, Antonino Tumeo, and Ananth Kalyanaraman. curipples: influence maximization on multi-gpu systems. In *Proceedings of the 34th ACM International Conference on Supercomputing*, pages 1–11, 2020.

[34] Marco Minutoli, Mahantesh Halappanavar, and Ananth Kalyanaraman. pnnl/ripples, 2021. URL https://github.com/pnnl/ripples.

[35] Mark EJ Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

[36] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471, 2013.

[37] Romualdo Pastor-Satorras, Claudio Castellano, Piet Van Mieghem, and Alessandro Vespignani. Epidemic processes in complex networks. *Reviews of modern physics*, 87(3):925, 2015.

[38] Onkar Patil, Latchesar Ionkov, Jason Lee, Frank Mueller, and Michael Lang. Performance characterization of a dram-nvm hybrid memory architecture for hpc applications using intel optane dc persistent memory modules. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, pages 288–303, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450372060. doi: 10.1145/3357526.3357541. URL https://doi.org/10.1145/3357526.3357541.

[39] I. Peng, K. Wu, J. Ren, D. Li, and M. Gokhale. Demystifying the performance of hpc scientific applications on nvm-based memory systems. In *2020 IEEE International Parallel and Distributed Processing Symposium*, pages 916–925, 2020.

[40] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. System evaluation of the intel optane byte-addressable nvm. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, pages 304–315, New York, NY, USA, 2019. Association for Computing Machinery. doi: 10.1145/3357526.3357568.

[41] David F Richards, Omar Aaziz, Jeanine Cook, Hal Finkel, Brian Homerding, Peter McCorquodale, Tiffany Mintz, Shirley Moore, Abhinacv Bhatele, and Robert Pavel. Fy18 proxy app suite release. milestone report for the ecp proxy app project. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2018.

[42] Rami Rosen. Resource management: Linux kernel namespaces and cgroups. *Haifux, May*, 186:70, 2013.

[43] Steve Scargall. Volatile use of persistent memory. In *Programming Persistent Memory*, pages 155–186. Springer, 2020.

[44] Anil Shanbhag, Nesime Tatbul, David Cohen, and Samuel Madden. Large-scale in-memory analytics on intel optanetm dc persistent memory. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, DaMoN '20, New York, NY, USA, 2020. Association for Computing Machinery. doi: 10.1145/3399666.3399933.

[45] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 135–146, 2013.

[46] Youze Tang, Xiaokui Xiao, and Yanchen Shi. Influence maximization: near-optimal time complexity meets practical efficiency. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 75–86. ACM, 2014. doi: 10.1145/2588555.2593670.

[47] Youze Tang, Yanchen Shi, and Xiaokui Xiao. Influence maximization in near-linear time: A martingale approach. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1539–1554. ACM, 2015. doi: 10.1145/2723372.2723734.

[48] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. Persistent memory i/o primitives. In *Proceedings of the 15th International*

Workshop on Data Management on New Hardware, DaMoN'19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368018. doi: 10.1145/3329785.3329930. URL https://doi.org/10.1145/3329785.3329930.

[49] Vish Viswanathan, Karthik Kumar, T Willhalm, P Lu, B Filipiak, and S Sakthivelu. Intel memory latency checker. *Intel Corporation*, 2013.

[50] Daniel Waddington, Mark Kunitomi, Clem Dickey, Samyukta Rao, Amir Abboud, and Jantz Tran. Evaluation of intel 3d-xpoint nvdimm technology for memory-intensive genomic workloads. In *Proceedings of the International Symposium on Memory Systems*, pages 277–287, 2019.

[51] Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. An early evaluation of intel's optane dc persistent memory module and its impact on high-performance scientific applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362290. doi: 10.1145/3295500.3356159. URL https://doi.org/10.1145/3295500.3356159.

[52] Kai Wu, Frank Ober, Shari Hamlin, and Dong Li. Early evaluation of intel optane non-volatile memory with hpc i/o workloads. *arXiv preprint arXiv:1708.02199*, 2017.

[53] Zhen Xie, Wenqian Dong, Jie Liu, Ivy Peng, Yanbao Ma, and Dong Li. Md-hm: Memoization-based molecular dynamics simulations on big memory system. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '21, pages 215–226, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383356. doi: 10.1145/3447818.3460365. URL https://doi.org/10.1145/3447818.3460365.

[54] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-12-0. URL https://www.usenix.org/conference/fast20/presentation/yang.

[55] P. Zardoshti, M. Spear, A. Vosoughi, and G. Swart. Understanding and improving persistent transactions on optane dc memory. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 348–357, 2020.

[56] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance dsl for graph analytics. *arXiv preprint arXiv:1805.00923*, 2018.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We study the performance of a shared-memory platform equipped with Intel Optane memory. The study starts by setting expectations on the platform through targeted micro-benchmarks for the three memory modes that the Intel Optane technology supports. We then evaluate the performance of two real world application from the data mining/graph analytics domain: community detection and influence maximization in social networks.

The microbenchmarking study reports idle latency, sequential and random access bandwidth for both sequential and parallel accesses. Measurements were taken using the Intel Memory Latency Checker (MLC v3.7) and a modified version of the STREAM benchmark.

The study then evaluates the system performance under two real world applications:

- Community Detection: Grappolo (shared-memory version) and miniVite (distributed-memory version);
- Influence Maximization: Ripples (shared-memory version) and cuRipples (distributed-memory version);

All the application codes are available open source on GitHub.

We conclude the study with a cost/performance analysis comparing an Intel Optane equipped shared-memory machine with state-of-the-art HPC systems. The MSRP for all the systems considered in the study were obtained directly contacting leading HPC system vendors.

0.0.1 Data Sets. All the data sets used in the study are publicly available for download at:

- **SNAP**: https://snap.stanford.edu/data/
- **LAW Dataset:** http://law.di.unimi.it/datasets.php
- **Suitesparse Matrix Collection:** https://sparse.tamu.edu/

The input graphs are converted to a special binary format to optimize I/O operations (the binary version allows loading the graph into the CSR data structures, without any intermediate steps). Instructions to convert to the binary version are provided in the respective `README` documents of the community detection and influence maximization codebases. The "Experimental evaluation" section of the paper mentions the build options.

0.0.2 Codes. Following are the links to the codebases used in this paper:

- Ripples (shared-memory): https://github.com/pnnl/ripples
- cuRipples (distributed-memory, CPU/GPU): https://github.com/pnnl/ripples
- Grappolo (shared-memory): https://github.com/Exa-Graph/grappolo
- miniVite (distributed-memory, CPU/GPU): https://github.com/Exa-Graph/miniVite

0.1 Platform-specific settings

0.1.1 Intel Optane platform. The shared memory system used in the study is a 24-core 4-way SDP (pre-release) version of the Intel Xeon Platinum 8260, with 2.3GHz Intel Xeon Platinum processor from the Cascade Lake micro-architecture (192 total threads) with 32KB L1 cache, 1MB L2 private cache, and 33MB shared L3 cache. Each processor has 6 memory channels, and 32GB DDR4 DIMMs making a total of 768GB of DRAM. The Optane DIMMs are 256GB each, making the total persistent memory capacity as 1.5TB per socket, or 6TB total. There are 3 links for Intel Ultra Path Interconnect (UPI) per CPU for connecting to other CPU sockets. We use Ubuntu 19.04 with v5.1 of Linux kernel. We use Intel VTune profiler v2021.1.0 beta10.

Ripples was compiled with GCC 8.3.0 and was configured to use $epsilon = 0.13$ and $k = 100$ for all the datasets.

Grappolo was compiled with the Intel Compiler v19.0.4.243.

0.1.2 OLCF Summit. We execute our implementations on OLCF Summit, a US Department of Energy's leadership class machine hosted at Oak Ridge National Laboratory. The system consists of 4,608 nodes, each equipped with two POWER9 CPUs, 6 NVIDIA Tesla V100 GPUs, 512 GB of DDR4 RAM, and two Infiniband EDR Network interfaces. A POWER9 CPU hosts 22 cores (1 is reserved for system software) with 4 threads each, has a frequency of 3.07 GHz and hosts 110 MB of L3 cache. Each Volta GPU hosts 80 Streaming Multiprocessors (64 FP32, 64 INT, 32 FP64 and 8 Tensor Cores running at 1.333 GHz) with a 16 GB of HBM2 memory (1750 MHz on a 4096-bit bus, providing 900 GB/s of bandwidth). Group of 3 GPUs are directly connected with a POWER9 CPU and among each other with NVLINK2 connections. Each connection uses 2 of the 6 NVLINK2 channels of a Volta GPU, offering up to 100 GB/s of bidirectional bandwidth. A group composed of 3 GPUs and a CPU communicates with the other one through the X-Bus between the two CPUs, providing a maximum bandwidth of 64 GB/s.

Ripples was compiled with GCC 10.2.0 and cuRipples was compiled with gcc 8.1.1 and CUDA 10.2.89. Both used the spectrum MPI library 10.3.1.2-20200121. We have performed scalability studies on 2–32 nodes of the machine as detailed in the paper. Ripples and cuRipples were configured to use $epsilon = 0.13$ and $k = 100$ on all datasets.

miniVite was compiled with GCC 8.1 and used Spectrum MPI library 10.3.1.2-20200121. We have performed scalability studies on 8–64 nodes. The "omptarget" branch of miniVite, which uses OpenMP offload model to move the computations on the GPUs was used in this evaluation. Additionally, to maintain an equitable graph distribution across process configurations, miniVite uses the "-b" option which is documented in the README.

0.1.3 NERSC Cori. We executed our implementation on the Haswell partition of Cori at the National Energy Research Scientific Computing (NERSC) Center. The system consists of 2,388 nodes each equipped with two Intel Xeon Processors E5-2698 v3 at 2.3GHz, 128GB of RAM and the Cray Aries with Dragonfly topology interconnect. A single CPU on the system hosts 16 cores with 2 threads per core.

Ripples was compiled with GCC 8.3.0 and we used the Cray MPICH 7.7.10. We have performed scalability studies from 2–32 nodes of the Cori Haswell partition as detailed in the paper. Ripples was configured to use *epsilon* = 0.13 and $k = 100$ on all datasets.

miniVite was compiled with Intel compiler (Intel v19.0.3) and we used Cray MPICH 7.7.10 as the MPI implementation. We performed scalability studies on 8–64 nodes of the Cori Haswell partition as detailed in the paper. Similar to the OLCF Summit platform, we use the balanced partitioning option for miniVite.

Author-Created or Modified Artifacts:

```
Persistent ID: 10.5281/zenodo.4673587,
↪ https://github.com/pnnl/ripples
Artifact name: Ripples (includes cuRipples)

Persistent ID: 10.5281/zenodo.4673626,
↪ https://github.com/Exa-Graph/grappolo
Artifact name: Grappolo

Persistent ID: 10.5281/zenodo.4677693,
↪ https://github.com/Exa-Graph/miniVite
Artifact name: miniVite
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Operating systems and versions: Ubuntu 19.04 with Linux kernel 5.1