Single Machine Graph Analytics on Massive Datasets Using Intel Optane DC Persistent Memory

Gurbinder Gill[1], Roshan Dathathri[1], Loc Hoang[1], Ramesh Peri[2], Keshav Pingali[1]
University of Texas at Austin[1], Intel Corporation[2]
{gill,roshan,loc,pingali}@cs.utexas.edu, {ramesh.v.peri}@intel.com

ABSTRACT

Intel Optane DC Persistent Memory (Optane PMM) is a new kind of byte-addressable memory with higher density and lower cost than DRAM. This enables the design of affordable systems that support up to 6TB of randomly accessible memory. In this paper, we present key runtime and algorithmic principles to consider when performing graph analytics on extreme-scale graphs on Optane PMM and highlight principles that can apply to graph analytics on all large-memory platforms.

To demonstrate the importance of these principles, we evaluate four existing shared-memory graph frameworks and one out-of-core graph framework on large real-world graphs using a machine with 6TB of Optane PMM. Our results show that frameworks using the runtime and algorithmic principles advocated in this paper (i) perform significantly better than the others and (ii) are competitive with graph analytics frameworks running on production clusters.

PVLDB Reference Format:
Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. Single Machine Graph Analytics on Massive Datasets Using Intel Optane Persistent Memory. *PVLDB*, 13(8): 1304 - 1318, 2020.
DOI: https://doi.org/10.14778/3389133.3389145

1. INTRODUCTION

Graph analytics systems must process graphs with billions of nodes and trillions of edges. Since the main memory of most single machines is limited to a few hundred GBs, shared-memory graph analytics systems like Galois [44], Ligra [52], and GraphIt [64] cannot perform in-memory processing of these large graphs. Two approaches have been used to circumvent this problem: (i) *out-of-core* processing and (ii) *distributed-memory* processing.

In out-of-core systems, the graph is stored in secondary storage (SSD/disk), and portions of the graph are read into DRAM under software control for in-memory processing. State-of-the-art systems in this space include GridGraph [66], X-Stream [49], Mosaic [38], and BigSparse [31]. Secondary storage devices do not support random accesses efficiently: data must be fetched and written in blocks. As a consequence, algorithms that perform well on shared-memory machines often perform poorly in an out-of-core setting, and it is necessary to rethink algorithms and implementations when transitioning from in-memory graph processing to out-of-core processing. In addition, the graph may need to be preprocessed to organize the data into an out-of-core friendly layout.

Large graphs can also be processed using distributed-memory clusters. The graph is partitioned among the machines in a cluster using one of many partitioning policies in the literature [24]. Communication is required during the computation to synchronize node updates. State-of-the-art systems in this space include D-Galois [20] and Gemini [65]. Distributed-memory graph analytics systems can scale out by adding new machines to provide additional memory and compute power. The overhead of communication can be reduced by choosing good partitioning policies, avoiding small messages, and optimizing metadata, but communication remains the bottleneck in these systems [20]. Obtaining access to large clusters may also be too expensive for many users.

Intel® Optane™ DC Persistent Memory (Optane PMM) is new byte-addressable memory technology with the same form factor as DDR4 DRAM modules but with higher memory density and lower cost. It has longer access times compared to DRAM, but it is much faster than SSD. It allows a single machine to have up to 6TB of storage at relatively low cost, and in principle, it can run memory-hungry applications without requiring the substantial reworking of algorithms and implementations needed by out-of-core or distributed-memory processing.

We explore the use and viability of Optane PMM for analytics of very large graphs such as web-crawls up to 1TB in size. We design and present studies conducted to determine *how* to run graph analytics applications on Optane PMM and large-memory systems in general. Our studies make the following points:

1. Non-uniform memory access (NUMA)-aware memory allocation of graph data structures that maximizes near-memory (DRAM treated as cache) usage is important on Optane PMM as cache misses on the platform are significantly slower than cache misses on DRAM. (Section 4)

2. Avoiding page management overhead while using Optane PMM is key to performance as kernel overhead on Optane PMM is higher due to higher access latency. (Section 4)

3. Algorithms must avoid high amounts of memory accesses on large memory systems: graph frameworks should give users the flexibility to write non-vertex, asynchronous programs with efficient parallel data structures. (Section 5)

We evaluate four shared-memory graph analytics frameworks – Galois [44], GAP [6], GraphIt [64], and GBBS (Ligra) [22] – on Optane PMM to show the importance of these practices for graph analytics on large-memory systems. We compare the performance

Proceedings of the VLDB Endowment, Vol. 13, No. 8
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3389133.3389145

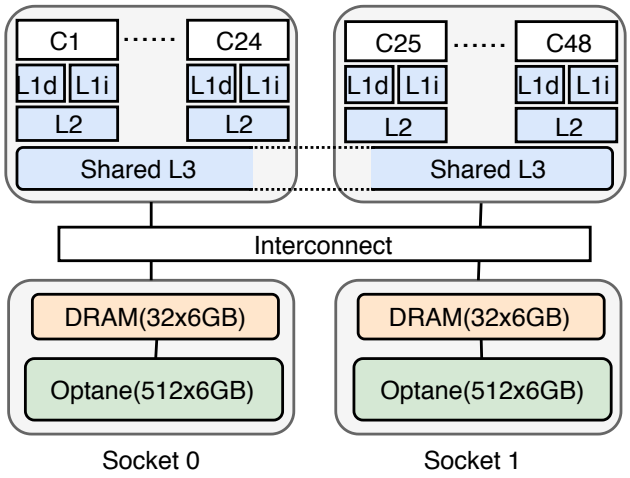


Figure 1: Memory hierarchy of our 2 socket machine with 384GB of DRAM and 6TB of Intel Optane PMM.

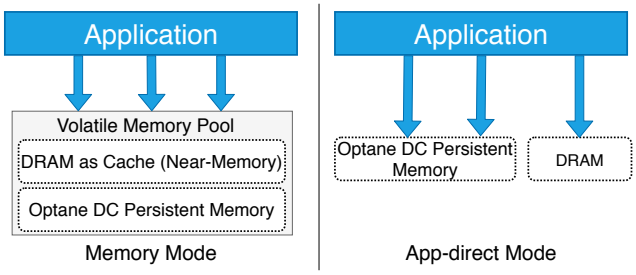


Figure 2: Modes in Optane PMM.

Table 1: Bandwidth (GB/s) of Intel Optane PMM (Rand: Random, Seq: Sequential).

Mode		Read		Write	
		Local	Remote	Local	Remote
Memory	**Rand**	90.0	34.0	50.0	29.5
	Seq	106.0	100.0	54.0	29.5
App-direct	**Rand**	8.2	5.5	3.6	2.3
	Seq	31.0	21.0	10.5	7.5

of the best of these frameworks, Galois (which uses our proposed practices), with the state-of-the-art distributed graph analytics system, D-Galois, and show that Galois running on Optane PMM is competitive with D-Galois on 256 machines for many benchmarks and inputs: as the Optane PMM system supports more efficient shared-memory algorithms which are difficult to implement on distributed-memory machines, applications using the more efficient algorithms can outperform distributed-memory execution. We also evaluate GridGraph [66], an out-of-core graph analytics system, using Optane PMM as external storage in app-direct mode and show that using Optane PMM as main memory in memory mode (modes are explained in Section 2) is orders of magnitude faster than app-direct as it allows for more sophisticated algorithms from shared-memory graph analytics systems that out-of-core systems currently do not support (in particular, non-vertex programs and asynchronous data-driven algorithms detailed in Section 5).

The paper is organized as follows. Section 2 introduces Optane PMM. Section 3 describes the experimental setup. Section 4 describes how to efficiently use the memory hierarchy on large-memory systems and Optane PMM for graph analytics. Section 5 discusses graph algorithm design for large-memory systems. Section 6 presents our evaluation. Section 7 surveys related work.

2. Optane PMM

Optane PMM delivers a combination of affordable large capacity and persistence (non-volatility). As shown in Figure 1, Optane PMM adds a new level to the memory hierarchy. It comes in the same form factor as a DDR4 memory module and has the same electrical and physical interfaces. However, it uses a different protocol than DDR4 which means that the CPU must have Optane PMM support in its memory controller. Similar to the DRAM distribution in non-uniform memory systems in which memory is divided into sockets, the Optane PMM modules are distributed among sockets. Figure 1 shows an example of a two socket machine with 6TB of Optane PMM split between sockets. Optane PMM can be configured as volatile main memory (memory mode), persistent memory (app-direct mode), or a combination of both (Figure 2).

Memory Mode: In memory mode, Optane PMM is treated as as main memory, and DRAM acts as a direct-mapped (physically indexed and physically tagged) cache called *near-memory*. The granularity of caching from Optane PMM to DRAM is 4KB. This enables the system to deliver DRAM-like performance at substantially lower cost and power with no modifications to the application. Although the memory media is persistent, the software sees it as volatile memory. This enables systems to provide up to 6TB of randomly accessible storage, which is expensive to do with DRAM.

Traditional code optimization techniques can be used to tune applications to run well in this configuration. In addition, software must consider certain asymmetries in machines with Optane PMM. Optane PMM modules on a socket can use only the DRAM present in its local NUMA node (i.e., socket) as near-memory. Therefore, in addition to NUMA allocation considerations, software using Optane PMM has to account for near-memory hit rate as the cost of a local near-memory miss is higher than the remote near-memory hit (discussed in Section 4). Therefore, it should allocate memory so that the system can utilize more DRAM as near-memory even if it means more remote NUMA accesses.

App-direct Mode: In app-direct mode, Optane PMM modules are treated as byte-addressable persistent memory. One compelling case for app-direct mode is in large memory databases where indices can be stored in persistent memory to avoid rebuilding them on reboot, achieving a significant reduction in restart time. Optane PMM modules can be managed using an API or a command line interface provided by the `ipmctl` [18] OS utility in Linux. `ipmctl` can be used to configure the machine to use $x\%$ of Optane PMM modules capacity in the memory mode and the rest in the app-direct mode [1]; for $x > 0$, all DRAM on the machine is used as the cache (`near-memory`). When all the Optane PMM modules are in app-direct mode, DRAM is the main volatile memory.

Specifications for the Optane PMM machine used in our study are in Section 3. Tables 1 and 2 show the bandwidth and latency of PMM observed on our machine. Although Optane PMM is slower than DDR4, its large capacity enables us to analyze much larger datasets on a single machine than previously possible. In this paper, we focus on memory mode; we use app-direct mode only for running the out-of-core graph analytics system, GridGraph [66].

[1] `ipmctl create -goal MemoryMode=x PersistentMemoryType=AppDirect`

Table 2: Latency (ns) of Intel Optane PMM.

Mode	Local	Remote
Memory	95.0	150.0
App-direct	164.0	232.0

Table 3: Inputs and their key properties.

	kron30	clueweb12	uk14	iso_m100	rmat32	wdc12
$\|V\|$	1,073M	978M	788M	76M	4295M	3,563M
$\|E\|$	10,791M	42,574M	47,615M	68,211M	68,719M	128,736M
$\|E\|/\|V\|$	16	44	60	896	16	36
max D_{out}	3.2M	7,447	16,365	16,107	10.4M	55,931
max D_{in}	3.2M	75M	8.6M	31,687	10.4M	95M
Est. diameter	6	498	2498	83	7	5274
Size (GB)	136	325	361	509	544	986

3. PLATFORMS AND GRAPH ANALYTICS SYSTEMS

Optane PMM experiments were conducted on a 2 socket machine with Intel's second generation Xeon scalable processor ("Cascade Lake") with 48 cores (96 threads with hyperthreading) with a clock rate of 2.2 Ghz. The machine has 6TB of Optane PMM, 384GB of DDR4 RAM, and 32KB L1, 1MB L2, and 33MB L3 data caches (Figure 1). The system has a 4-way associative data TLB with 64 entries for 2KB pages (small pages) and 32 entries for 2MB pages (huge pages). Code is compiled with g++ 7.3. We used the same machine for DRAM experiments by configuring it to run entirely in app-direct mode, leaving DRAM as the main volatile memory (equivalent to removing the Optane PMM modules). We also use the machine in app-direct mode to run GridGraph, an out-of-core graph analytics system: Optane PMM modules are used as the external storage. *Transparent Huge Pages (THP)* [37], which tries to allocate huge pages for an application without explicitly reserving memory for huge pages, are enabled (default in Linux). When we evaluated explicit *Huge Pages* allocation, we reserved 2TB and 360GB for Optane PMM and DRAM experiments respectively. To collect hardware counters and analyze performance, we used Intel's Vtune Amplifier [17] and Platform Profiler [16].

To show that our study of algorithms for massive graphs (Section 5) is independent of machine architecture, we also conducted experiments on a large 4 socket DRAM machine we call Entropy. Entropy uses Intel Xeon Platinum 8176 ("Skylake") processors with a total of 112 cores with a clock rate of 2.2 Ghz, 1.5TB of DDR4 DRAM, and 32KB L1, 1MB L2, and 38MB L3 data caches. Code is compiled with g++ 5.4. For our experiments on Entropy, we use 56 threads, restricting our experiments to 2 sockets.

Distributed-memory experiments were conducted on the Stampede [53] cluster at the Texas Advanced Computing Center using up to 256 Intel Xeon Platinum 8160 ("Skylake") 2 socket machines with 48 cores with a clock rate of 2.1 Ghz, 192GB DDR4 RAM, and 32KB L1, 1MB L2, and 33MB L3 data caches. The machines are connected with a 100Gb/s Intel Omni-Path interconnect. Code is compiled with g++ 7.1 and MPI MVAPICH2-X 2.3.

Table 3 specifies the input graphs: clueweb12 [48], uk14 [8, 7], and wdc12 [42] are web-crawls (wdc12 is the largest publicly available one), and iso_m100 [4] is a protein-similarity network (which is the largest dataset in the IMG isolate genomes publicly available as part of the HipMCL software [4]). kron30 and rmat32 are randomized scale-free graphs generated using kron [34] and rmat [10] generators (using weights of 0.57, 0.19, 0.19, and 0.05, as suggested by graph500 [1]). Table 3 also lists graph sizes on disk in Compressed Sparse Row (CSR) binary format. kron30 and clueweb12 fit into DRAM, so we use them to illustrate differences in workloads that fit into DRAM and those that do not. The other graphs – uk14, iso_m100, rmat32, and wdc12 – do not fit in DRAM on our Optane PMM machine. We observe that uk14 and wdc12 have non-trivial diameters, whereas rmat32 has a very small diameter. We believe that rmat32 does not represent real-world datasets, so we exclude it in all our experiments except to show the impact of diameter in our study of algorithms (Section 5). iso_m100 has edge weights; for the other graphs, we generate random weights.

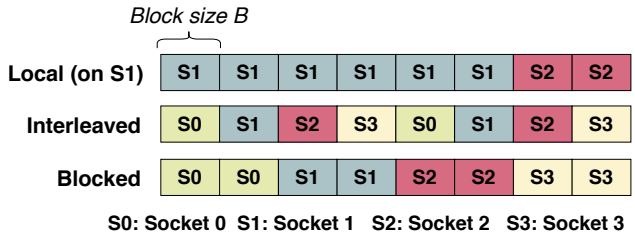


Figure 3: Illustration of 3 different NUMA allocation policies on a 4-socket system: each policy distributes blocks (size B, which is the size of a page) of allocated memory among sockets differently.

Our evaluation uses 7 benchmarks: single-source betweenness centrality (bc) [29], breadth-first search (bfs) [15], connected components (cc) [50, 51], k-core decomposition (kcore) [19], pagerank (pr) [45], single-source shortest path (sssp) [43], and triangle counting (tc) [28]. Only sssp uses edge weights. The source node for bc, bfs, and sssp is the maximum out-degree node. The tolerance for pr is 10^{-6}. The k in kcore is 100. *All benchmarks run until convergence except pr, which is run for up to 100 rounds*. We present the mean of 3 runs for the main experiments.

The shared-memory graph analytics frameworks we use are Galois [44], GAP [6], GraphIt [64], and GBBS (Ligra) [22] (all described in more detail in Section 6). D-Galois [20] is a distributed-memory framework. GridGraph [66] is an out-of-core framework that streams graph topology and data into memory from external storage (in this case, Optane PMM in app-direct mode).

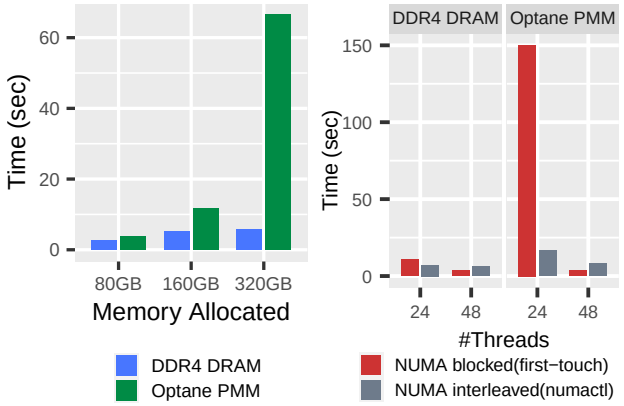
4. MEMORY HIERARCHY ISSUES

This section shows that on Optane PMM machines, the overhead of memory operations such as non-uniform memory accesses (NUMA) across memory sockets, cache miss handling, and page table maintenance are higher than DRAM machines. These overheads are reduced by intelligent memory allocation and by reducing the time spent in the kernel for page-table maintenance. We address three issues: NUMA-aware allocation (Section 4.1), NUMA-aware migration (Section 4.2), and page size selection (Section 4.3).

4.1 NUMA-aware Allocation

NUMA-aware allocation increases bandwidth and reduces latency of accesses to memory by allocating memory on the same NUMA nodes as the cores likely to access it. Allocation falls into three main categories: (a) *NUMA local*, which allocates memory on a node specified at allocation time (if there is not enough memory available on the preferred node, other nodes are used), (b) *NUMA interleaved*, which interleaves pages across nodes in a round-robin fashion, and (c) *NUMA blocked*, which blocks the pages and distributes the blocks among nodes (illustrated in Figure 3).

There are several ways for application programs to specify the allocation policy. The policy can be set globally by using OS utilities such as `numactl` [36] on Linux. To allow different policies to be used in different allocations, applications can use the OS-provided



(a) NUMA local on 96 threads (b) 320GB memory allocated

Figure 4: Time to write memory allocated on Optane PMM and DDR4 DRAM using a micro-benchmark.

NUMA allocation library (`numa.h` in Linux), which contains a variety of `numa_alloc` functions. OS-based approaches, however, can only use the NUMA local or interleaved policies. Another way to get fine-grained NUMA-aware allocation is to manually allocate memory using `anonymous mmap` [35] and have threads on different sockets inside the application touch the pages (called `first-touch`) to allocate them on the desired NUMA nodes. This method, unlike OS-provided methods, allows applications to implement application-specific NUMA-aware allocation policies.

To understand the differences in local, interleaved, and blocked NUMA allocation policies on our Optane PMM setup, we use a micro-benchmark that allocates different amounts of memory m using different NUMA allocation policies and writes to it using t threads such that thread i sequentially writes to the i^{th} contiguous block of size $\frac{m}{t}$. To explore the effects of NUMA on different platforms, we run this microbenchmark with two setups: one using only DDR4 DRAM (by setting Optane PMM into app-direct mode) and one using Optane PMM. The following micro-benchmark results show that on the Optane PMM machine, applications must not only maximize local NUMA accesses, but *must also use a NUMA policy that maximizes the near-memory used in order to reduce DRAM conflict misses* (recall that DRAM is direct-mapped cache called near-memory for Optane PMM in this mode, and the limited cache size increases the probability of conflict misses due to physical addresses mapping to the same cache line).

NUMA Local. Figure 4(a) shows the execution time of the microbenchmark on DDR4 DRAM and Optane PMM for the NUMA local allocation policy using $t = 96$ and different amounts of allocations. Using NUMA local, all the memory of socket 0 is used before memory from socket 1 is allocated. We observe that going from 80GB to 160GB increases the execution time by $2\times$ for both DRAM and Optane PMM: this is expected since we increase the work by $2\times$. Going from 160GB to 320GB also increases the work by $2\times$. For DRAM, a 320GB allocation spills to the other socket (each socket has only 192GB), increasing the effective bandwidth by $2\times$, so the execution time does not change much. In Optane PMM, however, the 320GB is allocated entirely on socket 0 as our machine has 3TB per socket. Since there is no change in bandwidth, one would expect the performance to degrade by $2\times$, but it degrades by $5.6\times$. This is because the machine can only use 192GB of DRAM as near-memory; this cannot fit 320GB, so the conflict miss rate of the DRAM accesses increase by roughly $1.8\times$. This illustrates that (i) near-memory conflict misses are detrimental to the performance for Optane PMM and (ii) NUMA local is not suitable for allocations larger than 192GB on our setup.

NUMA Interleaved and Blocked. The execution times of the microbenchmark on DDR4 DRAM and Optane PMM for the NUMA interleaved and blocked allocation policies using an allocation of 320GB and different thread counts are shown in Figure 4(b). For DRAM, both policies are similar for different t. When $t \leq 24$ on Optane PMM, NUMA blocked only allocates memory on socket 0 (because it uses `first-touch`), so performance degrades $39\times$ compared to 48 thread execution as 320GB does not fit in the near-memory of a single socket. This illustrates that the cost of local near-memory misses is much higher than the cost of remote near-memory hits. In contrast, the NUMA interleaved policy for 24 threads uses both sockets and improves performance by $9\times$ over NUMA blocked even though 50% of accesses are remote when $t \leq 24$. When $t = 48$, both allocation policies are able to fit 320GB in the near-memory of 2 sockets (384GB). However, NUMA interleaved is $\sim 2\times$ slower than NUMA blocked because interleaved results in more remote accesses as compared to blocked.

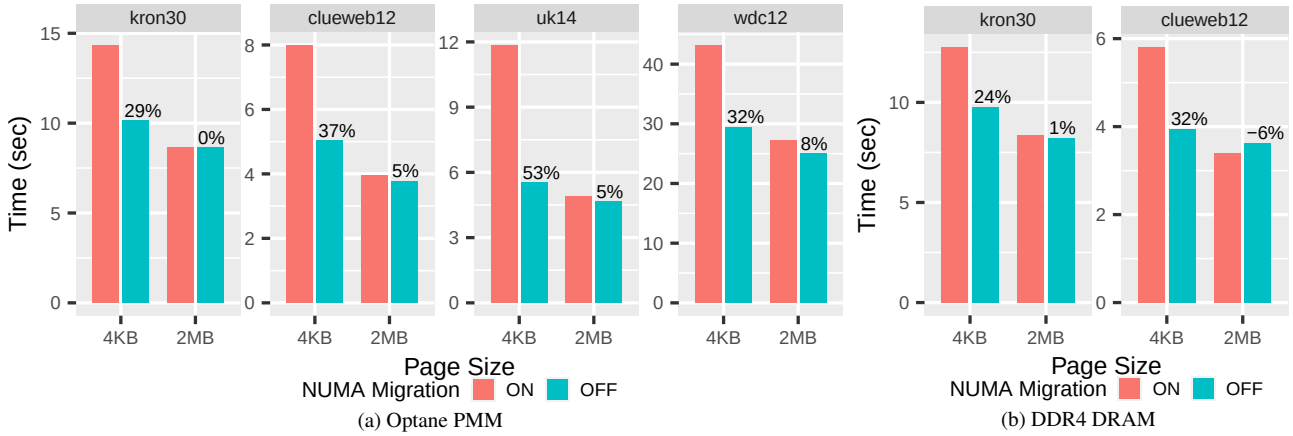
4.2 NUMA-aware Migration

When an OS-level NUMA allocation policy is not specified, the OS can dynamically migrate data among NUMA nodes to increase local NUMA accesses. NUMA page migrations are helpful for multiple applications sharing a single system as they try to move pages closer to the cores assigned to each application. However, for a single application, this policy may not always be useful, especially when application has specified its own allocation policy.

NUMA migration has overheads: (a) it requires book-keeping to track pages accesses for selecting migrated pages, and (b) it changes the virtual-to-physical address mapping, making the Page Table Entries (PTEs) cached in the CPU's Translation Lookaside Buffers (TLBs) stale which causes TLB shootdown to invalidate stale entries. TLB shootdown increases TLB misses and involves slow operations such as inter-processor interrupts (IPIs).

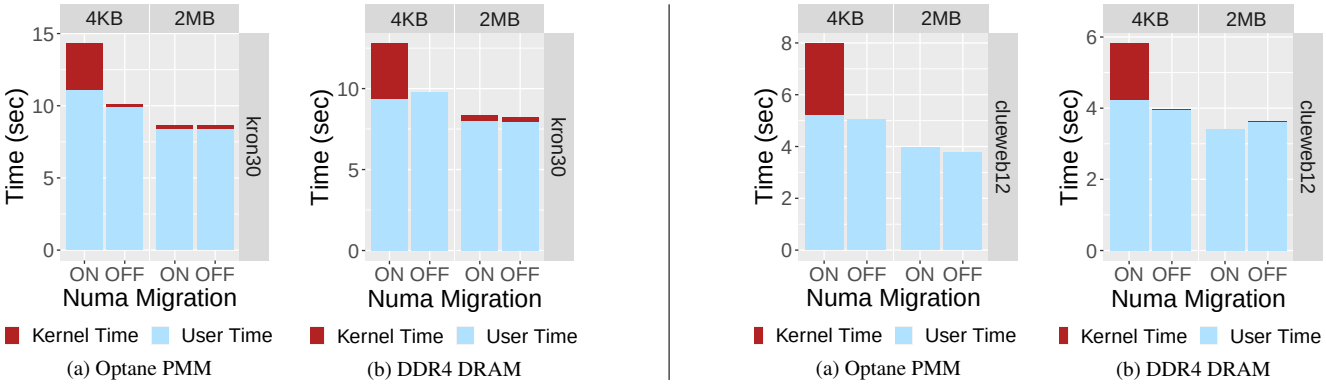
Graph analytics applications tend to have irregular access patterns: accesses are arbitrary, so there may be many shared accesses across NUMA nodes. To examine the effects of page migration on graph analytics applications, we run breadth-first search (bfs) (similar trends observed for other benchmarks) with Galois [44] using NUMA interleaved allocation on both Optane PMM and DDR4 DRAM with NUMA migration on and off. We also examine the effects of page migration for different page sizes (which affects the number of pages migrated): (a) 4KB small pages and (b) 2MB huge pages. The results suggest that *NUMA migration should be turned off for graph analytics applications on Optane PMM*.

Figure 5 shows the effect of NUMA migration where the number on each bar presents the % change in the execution time when NUMA migration is turned off. A positive number means turning migration off improves performance. Performance improves in most cases if migration is turned off. Figure 6 shows that the time spent in user code is not affected by the migrations, which shows that they add kernel time overhead without giving significant benefits. Another way to measure the efficacy of the migrations is to measure the % of local near-memory (DRAM) accesses in Optane PMM: if migration is beneficial, then it should increase. However, this does not change by more than 1%. Figure 6 shows that migrations hurt performance more on Optane PMM compared to DRAM as kernel time spent is higher. This is due to (a) higher bookkeeping cost as accesses to kernel data structures are more expensive on Optane PMM and (b) higher cost of TLB shootdown as it increases the access latency to the near-memory (DRAM) being used as a



(a) Optane PMM (b) DDR4 DRAM

Figure 5: Execution time of bfs in Galois using small (4KB) and huge (2MB) page sizes with and without NUMA migration.



(a) Optane PMM (b) DDR4 DRAM (a) Optane PMM (b) DDR4 DRAM

Figure 6: Breakdown of execution time of bfs in Galois using different page sizes for kron30 (left) and clueweb12 (right).

direct-mapped cache since TLB translation is on the critical path[2]. Larger graphs exacerbate this effect as they use more pages.

4KB small page size shows more performance improvement than 2MB huge pages when turning page migrations off. We observe that the number of migrations is in the millions for small pages and in the hundreds for huge pages. The finer granularity of small pages makes them more prone to migrations, leading to more TLB shootdowns and data TLB misses. Therefore, for small pages, the number of data TLB misses reduces by $\sim 2\times$ by turning off the migrations for all the graphs. The number of small pages being $512\times$ the number of huge pages also increases the bookkeeping overhead in the OS. This is reflected in time spent in the OS kernel seen in Figure 6: the time spent in the kernel is more for the smaller page size than for the larger page size if migration is turned on.

4.3 Page Size Selection

When memory sizes and workload sizes grow, the time spent handling TLB misses can become a performance bottleneck since large working sets need many virtual-to-physical address translations that may not be cached in the TLB. This bottleneck can be tackled (a) by increasing the TLB size or (b) by increasing the page size. The TLB size is determined by the micro-architecture and cannot easily be changed by a user. On the other hand, processors allow users to customize page sizes as different page sizes may work best for different workloads. For example, x86 supports traditional 4KB small pages as well as 2MB and 1GB huge pages.

We studied the impact of page size on graph analytics using a 4KB small page size and a 2MB huge page size. We did not include the 1GB page size as it requires special setup; moreover, we do not expect to gain from 1GB pages as our machine supports only 4 TLB entries for 1GB page size. We run bfs (similar behavior observed for other benchmarks) using Galois [44] with no NUMA migration and NUMA interleaved allocation for various large graphs on (a) Optane PMM and (b) DDR4 DRAM. The results suggest that *a page size of 2MB is good for graph analytics on Optane PMM*.

Figure 5 shows bfs runtimes with various page sizes, and we observe that using huge pages is always beneficial on large graphs as huge pages reduce the number of pages by $512\times$, reducing the number of TLB misses ($3.2\times$ for clueweb12, $11.2\times$ for uk14 and $1.9\times$ for wdc12) and CPU cycles spent on page walking on TLB misses ($7.3\times$ for clueweb12, $12.5\times$ for uk14 and $8.8\times$ for wdc12). We also observe that the benefits of huge pages are higher on Optane PMM than on DRAM because TLB misses increase the near-memory access latency. Huge pages increase the TLB reach (TLB size $\times$ page size), thereby reducing the TLB misses.

4.4 Summary

For high-performance graph analytics on Optane PMM, we recommend (i) NUMA interleaved or blocked allocation rather than NUMA local, particularly for large allocations ($>$ 192GB), (ii) turning off NUMA page migration, and (iii) using 2MB huge pages.

[2]Since near-memory is physically indexed and physically tagged direct-mapped cache, virtual addresses are translated to physical addresses before cache (near-memory) can be accessed.

5. EFFICIENT ALGORITHMS FOR MASSIVE GRAPHS

Generally, there are many algorithms that can solve a given graph problem; for example, the single-source shortest-path (sssp) problem can be solved using Dijkstra's algorithm [15], the Bellman-Ford algorithm [15], chaotic relaxation [11], or delta-stepping [43]. These algorithms may have different asymptotic complexities and different amounts of parallelism. For a graph $G = (V, E)$, the asymptotic complexity of Dijkstra's algorithm is $O(|E|*log(|V|))$ while Bellman-Ford is $O(|E|*|V|)$, but for most graphs, Dijkstra's algorithm has little parallelism compared to Bellman-Ford. In addition, a given algorithm can usually be implemented in different ways that can affect parallel performance dramatically; implementations with fine-grain locking, for example, usually perform better than those with coarse-grain locking.

This section presents a classification of graph analytics algorithms that is useful for understanding parallel performance [47]. It also presents experimental results that provide insights into which classes of algorithms perform well on large input graphs.

5.1 Classification of Graph Analytics Algorithms

Operators. In graph analytics algorithms, each vertex has one or more labels that are initialized at the start of the computation and updated repeatedly during computation until a quiescence condition is reached. Label updates are performed by applying an *operator* to *active vertices* in the graph. In some systems, such as Galois [44], an operator may read and update an arbitrary portion of the graph surrounding the active vertex; this portion is called its *neighborhood*. Most shared-memory systems such as Ligra [52, 22] and GraphIt [64] only support *vertex programs*: operator neighborhoods are only the immediate neighbors of the active vertex. Non-vertex programs, conversely, have no restriction on an operator's neighborhood. A *push-style* operator updates the labels of the neighbors of the active vertex, while a *pull-style* operator updates the label of only the active vertex. *Direction-optimizing* implementations [5] can switch between push and pull style operators dynamically but require a reverse edge for every forward edge in the graph, doubling the memory footprint of the graph.

Schedule. To find active vertices in the graph, algorithms take one of two approaches. A *topology-driven* algorithm executes in rounds. In each round, it applies the operator to all vertices; an example is Bellman-Ford algorithm for sssp. These algorithms are simple to implement, but they may not be work-efficient if there are few active vertices in a lot of rounds. To address this, *data-driven* algorithms track active vertices explicitly and only apply the operator to these vertices. At the start of the algorithm, some vertices are active; applying the operator to an active vertex may activate other vertices, and operator application continues until there are no active vertices in the graph. Dijkstra and delta-stepping sssp algorithms are examples. Active vertices can be tracked using a bit-vector of size V if there are V vertices in the graph: we call this a *dense worklist* [52, 22, 64]. Other implementations keep an explicit worklist of active vertices [44]: we call this a *sparse worklist*.

Some implementations of data-driven algorithms execute in *bulk-synchronous* rounds: they keep a *current* and a *next* (dense or sparse) worklist, and in each round, they process only vertices in the current worklist and add activated vertices to the next worklist. In contrast, *asynchronous* data-driven implementations have no notion of rounds; they maintain a single sparse worklist, pushing and popping active vertices from this worklist until it is empty.

5.2 Algorithms for Very Large Graphs

At present, very large graphs are analyzed using clusters or out-of-core systems, but these systems are restricted to vertex programs and round-based execution. This is not considered a serious limitation for power-law graphs since they have a small diameter and information does not have to propagate many hops in these graphs. In fact, no graph analytics framework other than Galois provides sparse worklists, so they do not support asynchronous data-driven algorithms, and most of them are restricted to vertex programs.

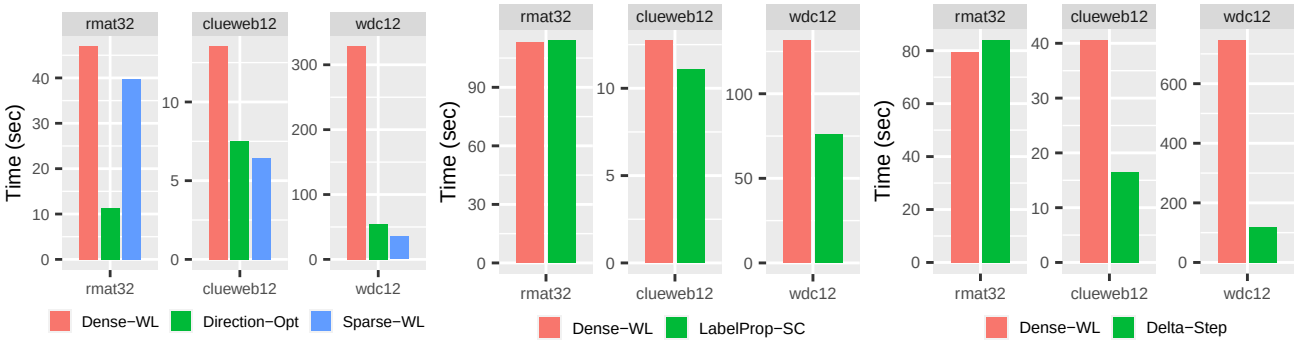
Using Optane PMM, we use a single machine to perform analytics on very large graphs, and our results suggest that conventional wisdom in this area needs to be revised. *The key issue is highlighted by Table 3: clueweb12, uk14, and wdc12, which are real-world web-crawls, have a high diameter (shown in Table 3) compared to kron30 and rmat32, the synthetic power-law graphs.* We show that for standard graph analytics problems, the best-performing algorithms for these graphs may be (a) non-vertex programs and (b) asynchronous data-driven algorithms, which require sparse worklists. These algorithms have better work-efficiency and make fewer memory accesses, which is good for performance especially on Optane PMM where memory accesses are more expensive.

Figure 7 shows the execution time of different data-driven algorithms for bfs, cc, and sssp on Optane PMM using the rmat32, clueweb12, and wdc12 graphs on the Galois system. For bfs, all algorithms are bulk-synchronous. A vertex program with direction optimization (that uses dense worklists) performs well for rmat32 since it has a low-diameter, but for the real-world web-crawls which have higher diameter, it is outperformed by an implementation with a push-style operator and sparse worklists since this algorithm has a lower memory footprint and makes fewer memory accesses. For cc, we evaluate bulk-synchronous label propagation [50] combined with short-cutting (LabelProp-SC) [54], which uses a non-vertex operator. It is a variant of the Shiloach-Vishkin (SV) algorithm [51] where after a round of label propagation it jumps one level unlike SV algorithm where it goes to the common ancestor. LabelProp-SC exhibits better locality compared to SV algorithm and significantly outperforms the bulk-synchronous algorithm that uses a simple label propagation vertex operator for the real-world web-crawls. For sssp, the asynchronous delta-stepping algorithm, which maintains a sparse worklist, significantly outperforms the bulk-synchronous data-driven algorithm with dense worklists. These findings do not apply only to Optane PMM: Figure 8 shows the same experiments for bfs, sssp, and cc conducted on Entropy (DDR4 DRAM machine). The trends are similar to those on Optane PMM machine.

Summary. Large real-world web-crawls, which are the largest graphs available today, have a high diameter unlike synthetically generated rmat and kron graphs. Therefore, conclusions drawn from experiments with rmat and kron graphs can be misleading. On current distributed-memory and out-of-core platforms, one is forced to use vertex programs, but on machines with Optane PMM, *it is advantageous to use algorithms with non-vertex operators and sparse worklists of active vertices that allow for asynchronous execution*. Frameworks with only vertex operators or no sparse worklists are at a disadvantage on this platform when processing large real-world web-crawls, as we show next.

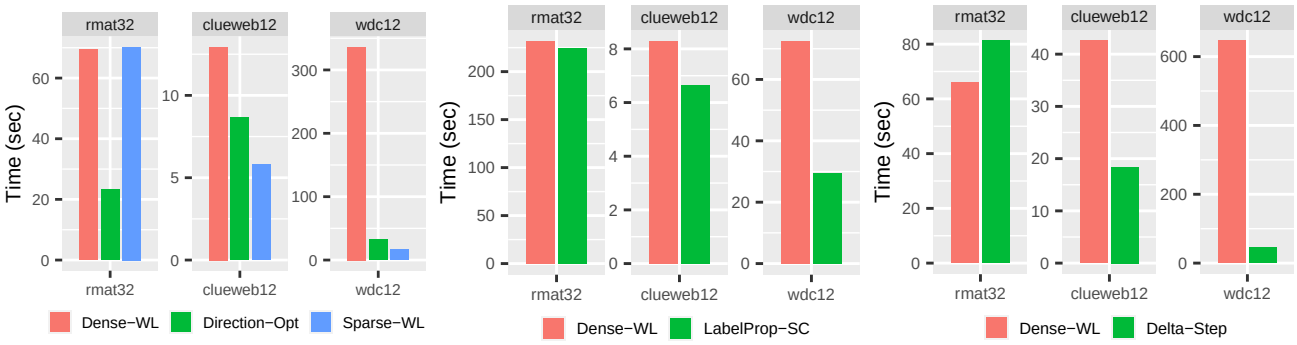
6. EVALUATION OF GRAPH FRAMEWORKS

In this section, we evaluate several graph frameworks on Optane PMM in the context of the performance guidelines presented in Sections 4 and 5. In Section 6.1, four shared-memory graph analytics systems - Galois [44], GAP [6], GraphIt [64], and GBBS [22]



(a) bfs (b) cc (c) sssp

Figure 7: Execution time of different data-driven algorithms in Galois on Optane PMM using 96 threads.



(a) bfs (b) cc (c) sssp

Figure 8: Execution time of different data-driven algorithms in Galois on Entropy (1.5TB DDR4 DRAM) using 56 threads.

- are evaluated on the Optane PMM machine using several graph analytics applications. Section 6.2 describes the experiments with medium sized graphs stored either in Optane PMM or DRAM. These experiments provide end-to-end estimates of the overhead of executing applications with data in Optane PMM rather than in DRAM. Section 6.3 describes experiments with large graphs that fit only in Optane PMM, and performance is compared with distributed-memory performance on a production cluster with up to 128 machines. Section 6.4 presents our experiments with GridGraph [66], an out-of-core graph analytics framework, using Optane PMM's app-direct mode to use it as external memory.

6.1 Galois, GAP and GraphIt on Optane PMM

Setup. To choose a shared-memory graph analytics system for our experiments, we evaluate (1) Galois [44], which is a library and runtime for graph processing, (2) GAP [6], which is a benchmark suite of expert-written graph applications, (3) GraphIt [64], which is a domain-specific language (DSL) and optimizing compiler for graph computations, and (4) GBBS [22], which is a benchmark suite of graph algorithms written in the Ligra [52] framework.

The choice of these frameworks was made as they exemplify different approaches to shared-memory graph analytics. GraphIt is a DSL that only supports vertex programs, and it has a compiler that uses auto-tuning to generate optimized code; the optimizations are controlled by the programmer. Galois is a C++-based general-purpose programming system based on a runtime that permits optimizations to be specified in the program at compile-time or at runtime, giving the application programmer a large design space of implementations that can be explored. GBBS programs are expressed in a graph processing library and runtime, Ligra. Therefore, Galois and GBBS require more programming effort than GraphIt. GBBS includes theoretically efficient algorithms written by experts, while Galois includes algorithms written using expert-provided concurrent data structures and operator schedulers. GAP is a benchmark suite of graph analytics applications written by expert programmers; it does not provide a runtime or data structures like Galois.

The kcore application is not implemented in GAP and GraphIt, so we omit it in the comparisons reported in this section. We omit the largest graph wdc12 for GAP and GraphIt because neither can handle graphs that have more than $2^{31}-1$ nodes (they use a `signed 32-bit int` for storing node IDs). GAP, GraphIt, and GBBS do not use NUMA allocation policies within their applications, so we use the OS utility numactl to use NUMA interleaved. For Galois, we use the best-performing algorithm with a runtime option, and we did not try different worklists or chunk-sizes. Galois allows programmers to choose NUMA interleaved or blocked allocation for each application by modifying a template argument in the program; we choose interleaved for bfs, cc, and sssp and blocked for bc, and pr. For GraphIt, we used the optimizations recommended by the authors in the GraphIt artifact [64].

Results. Figures 9 shows the execution times on Optane PMM (GraphIt does not have bc). Galois is generally much faster than GraphIt, GAP, and GBBS: on average, Galois is 3.8×, 1.9×, and 1.6× faster than GraphIt, GAP, and GBBS, respectively. There are many reasons for these performance differences.

Algorithms and implementation choices affect runtime (discussed in Section 5.2). For all algorithms, GAP, GBBS and GraphIt use a dense worklist to store the *frontier* while Galois uses a sparse work-

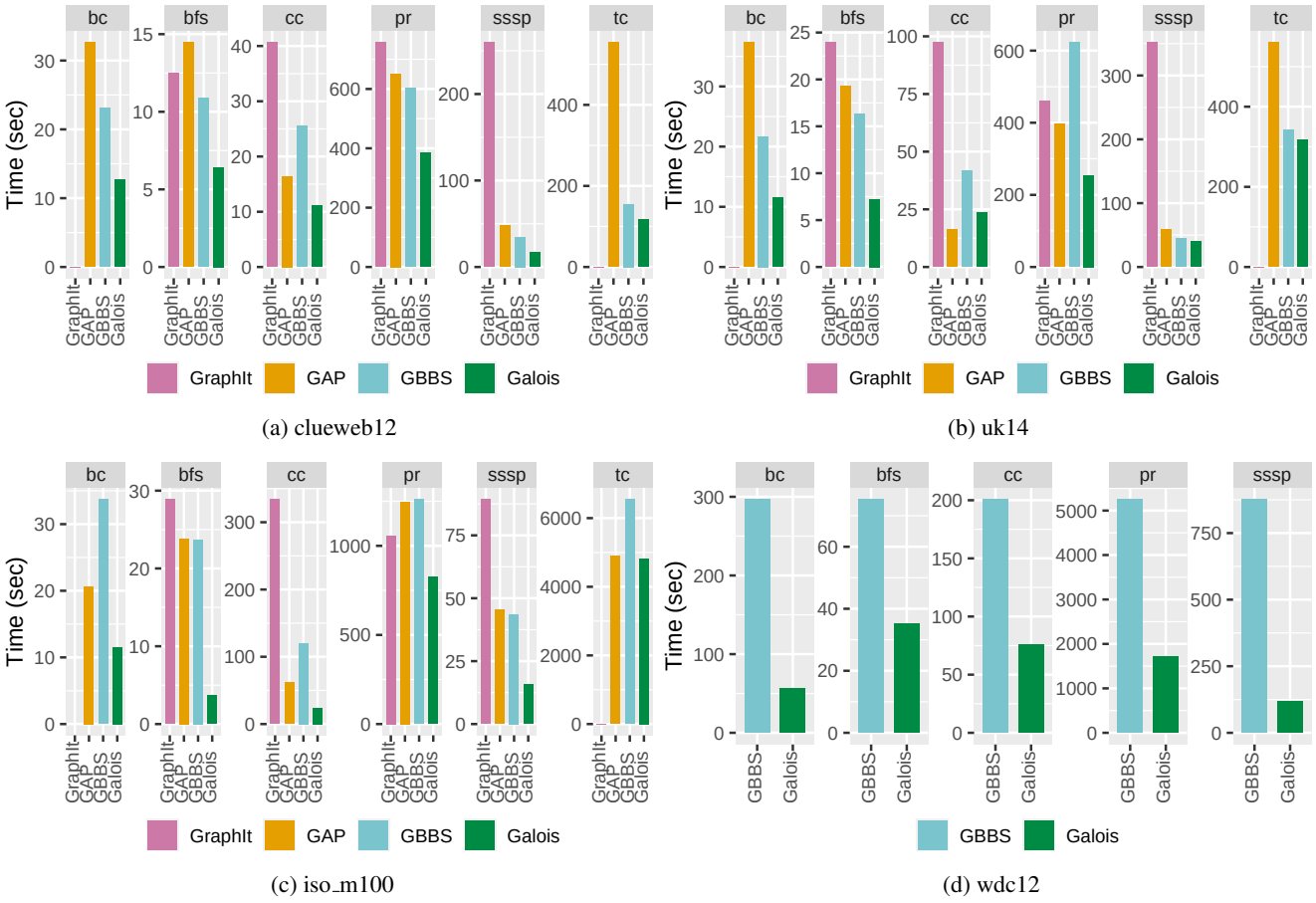


Figure 9: Execution time of benchmarks in GraphIt, GAP, GBBS, and Galois on Optane PMM using 96 threads.

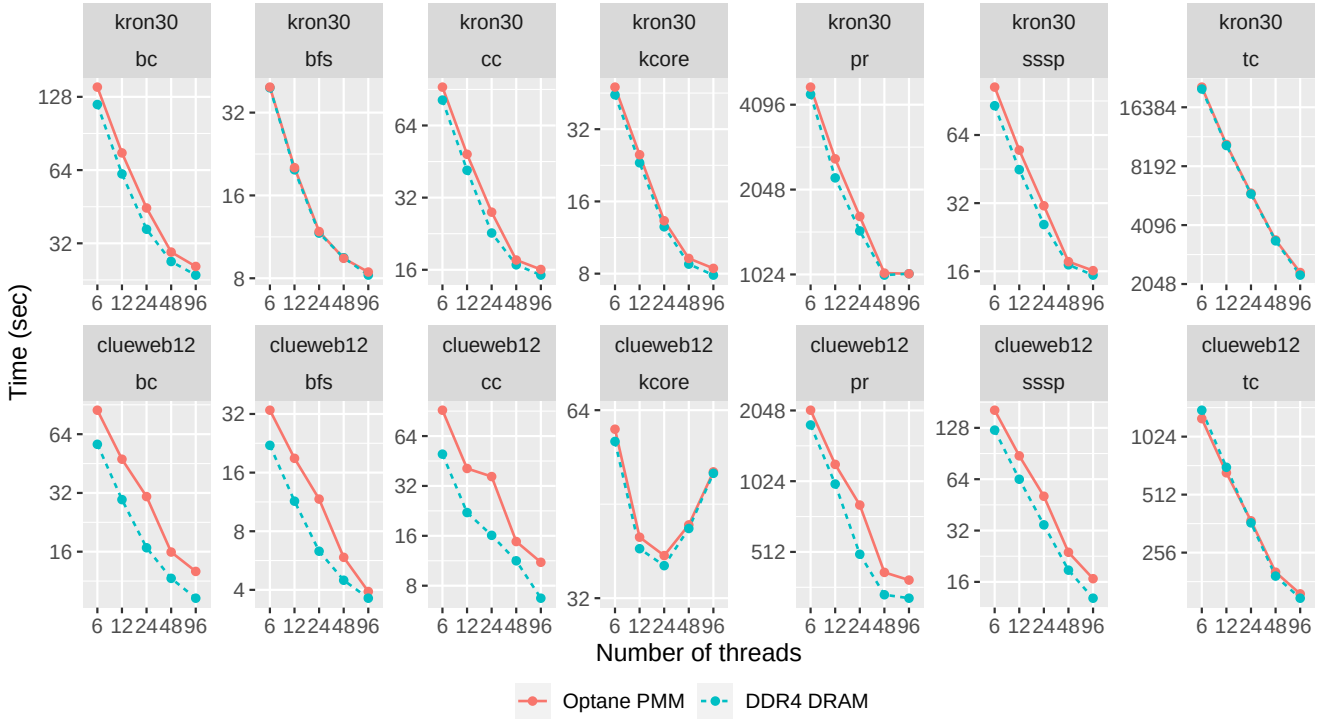


Figure 10: Strong scaling in execution time of benchmarks in Galois using DDR4 DRAM and Optane PMM.

list except for pr (large diameter graphs tend to have sparse frontiers). All systems use the same algorithm for pr. For bfs, all systems except Galois use direction-optimization which accesses both in-edges and out-edges (increasing memory accesses). For sssp, GAP, GBBS, and Galois use delta-stepping [43]; GraphIt does not support such algorithms. For cc, GAP and GBBS use the Shiloach-Vishkin [51] algorithm, Galois uses label propagation with short-cutting [54] algorithm, and GraphIt uses label propagation [50] because it supports only vertex programs. Furthermore, Galois uses asynchronous execution for sssp and cc unlike the others.

Another key difference is how the three systems perform memory allocations. Galois is the only framework that explicitly uses huge pages of size 2MB whereas the others use small pages of size 4KB and rely on the OS to use *Transparent Huge Pages (THP)*. As discussed in Section 4, huge pages can significantly reduce the cost of memory accesses over small pages even when THP is enabled. Galois is also the only one to provide NUMA blocked allocation, and we chose that policy because it performed observably better than the interleaved policy for some benchmarks such as bc and pr (the performance difference was within 18%). In general, we observe that NUMA blocked performs better for topology-driven algorithms, while NUMA interleaved performs better for data-driven algorithms. In addition, GAP, GBBS, and GraphIt allocate memory for both in- and out-edges of the graph while Galois only allocates memory only for the direction(s) needed by the algorithm. Allocating both increases the memory footprint and leads to conflict misses in near-memory when both in-edges and out-edges are accessed.

To conclude, our experiments show that in order to acheive performance for large graphs, a framework must support asynchronous, non-vertex programs as well as allow users explicit control over memory allocation. As Galois supports these features out-of-the-box, we use it for the rest of our experiments.

6.2 Medium size graphs: Using Optane PMM vs. DDR4 DRAM

Setup. In this subsection, we determine the overhead of using Optane PMM over DRAM by examining the runtimes for graphs that are small enough to fit in DRAM (384 GB). We use with kron30 and clueweb12 (Table 3) which both fit in DRAM. We use algorithms in Galois that perform best on 96 threads.

Results. Figure 10 shows strong scaling on DRAM and on Optane PMM with DRAM as cache. kron30 requires $\sim$ 136GB, which is a third of the near-memory available, so Optane PMM is almost identical to DRAM as it can cache the graph in near-memory effectively. On the other hand, clueweb12 requires $\sim$ 365GB, which is close to the near-memory available, so there are significantly more conflict-misses ($\approx$ 26%) in near-memory. On 96 threads, Optane PMM can take up to 65% more execution time than DRAM, but on average, it takes only 7.3% more time than DRAM.

Another trend is that if the number of threads is less than 24, Optane PMM can be slower than DRAM because of the way Galois allocates memory. Interleaved and blocked allocation policies in Galois interleave and block among *threads*, not sockets. If threads used is less than 24, all threads run on one socket, and all memory is allocated there, leading to under-utilization of the DRAM in the entire system: this results in more conflict-misses in near-memory.

The strong scaling for all the applications is similar whether DRAM or Optane is used as main memory. We also measured the performance of all applications for larger graphs on 8 and 96 threads of Optane. The geo-mean speedup of all applications on 96 threads over 8 threads is 4.3×, 4.2×, 4.7×, and 3.5× for kron30, clueweb12, uk14, and wdc12 respectively. Thus, the strong scaling speedup does not vary by much as the size of the graph grows.

Table 4: Execution time (sec) of benchmarks in Galois on Optane PMM (OB) machine using efficient algorithms (non-vertex, asynchronous) and D-Galois on Stampede cluster (DM) using vertex programs with minimum # of hosts that hold the graph. Speedup of Optane PMM over Stampede. Best times highlighted in green.

Graph	App	Stampede (DM)	Optane PMM (OB)	Speedup (DM/OB)
clueweb12	bc	51.63	12.68	4.07×
	bfs	10.71	6.43	1.67×
	cc	13.70	11.08	1.24×
	kcore	186.03	51.05	3.64×
	pr	155.00	385.64	0.40×
	sssp	33.87	16.58	2.04×
uk14	bc	172.23	11.53	14.9×
	bfs	28.38	7.22	3.93×
	cc	14.56	21.30	0.68×
	kcore	56.08	7.94	7.06×
	pr	82.77	254.95	0.32×
	sssp	52.49	39.99	1.31×
iso_m100	bc	6.97	11.57	0.60×
	bfs	7.94	3.69	2.15×
	cc	16.32	23.69	0.69×
	kcore	1.21	0.48	2.52×
	pr	191.21	824.54	0.23×
	sssp	61.90	15.66	3.95×
wdc12	bc	775.84	56.48	13.7×
	bfs	71.50	35.25	2.03×
	cc	69.21	76.00	0.91×
	kcore	105.42	49.22	2.14×
	pr	118.01	1706.35	0.07×
	sssp	136.47	118.81	1.15×

6.3 Very large graphs: Using Optane PMM vs. a Cluster

Setup. For very large graphs that do not fit in DRAM, the conventional choices are to use a distributed or an out-of-core system. In this subsection, we compare execution of Galois on Optane PMM to execution of the state-of-the-art distributed graph analytics system D-Galois [20] on the Stampede [53] cluster to determine the competitiveness of Optane PMM compared to a distributed cluster. We chose D-Galois because it is faster than existing distributed graph systems (like Gemini [65], PowerGraph [26], and GraphX [60]) and it uses the same computation runtime as Galois, which makes the comparison fairer. To partition [27] graphs among machines, we follow the recommendations of a previous study [24] and use Outgoing Edge Cut (OEC) for 5 and 20 hosts and Cartesian Vertex Cut (CVC) [9, 20] for 256 hosts. D-Galois supports bulk-synchronous vertex programs[3] with dense worklists as they simplify communication. Therefore, it cannot support some of the more efficient non-vertex programs in Galois. We exclude graph loading, partitioning, and construction time in the reported numbers. For logistical reasons, it is difficult to ensure that both platforms use the exact same resources (threads and memory). For a fair comparison, we limit the resources used on both platforms.

[3]D-Galois also supports bulk-asynchronous execution [21] (execution on each host is still round-based), but we did not use it here.

Results. Table 4 compares the performance of Optane PMM (on a single machine) running non-vertex, asynchronous Galois programs (referred to as **OB**) with the Stampede distributed cluster running D-Galois vertex programs using the minimum number of hosts required to hold the graph in memory (5 hosts for clueweb12, and uk14, and 20 hosts for wdc12; 48 threads per host and referred to as **DM**). Optane PMM outperforms D-Galois in most cases (best times highlighted), except for pr on clueweb12, uk14, and wdc12 and cc on uk14, and wdc12. Optane PMM gives a geomean speedup of 1.7× over D-Galois even though D-Galois has more cores (240 cores for clueweb12 and uk14; 960 cores for wdc12) and memory bandwidth. In pr, almost all nodes are updated in every round (similar to the topology driven algorithms); therefore, on the distributed cluster, it benefits from the better spatial locality in D-Galois resulting from the partitioning of the graph into smaller local graphs and more memory bandwidth.

Table 4 also shows the performance of many applications on Optane for input graphs of different sizes (smallest to largest). Most applications take more time for larger graphs. Although wdc12 is $\sim 3\times$ larger (in terms of edges) than clueweb12, kcore and pr take less time on wdc12 than clueweb12, but the other applications take on average $\sim 7\times$ more time on wdc12 than clueweb12. This provides an estimate for the weak scaling of these applications.

Further Analysis. The bars labeled **O**_ in Figure 11 show times on the Optane PMM system with the following configurations:- **OB**: Performance using the best algorithm in Galois for that problem and all 96 threads (same as shown in Table 4); **OA**: Performance using the best vertex programs in Galois for that problem and all 96 threads; **OS**: Same as **OA** but using only 80 threads. The bars labeled **D**_ show times on the Stampede system with the following configurations:- **DB**: Performance using D-Galois vertex programs on 256 machines (12,288 threads); **DM**: Performance using D-Galois vertex programs using the minimum number of hosts required to hold graph in memory (same as shown in Table 4). **DS**: Same as **DM** but using a total of 80 threads across all machines.

Results. Figure 11 shows the experimental results. For bars **DS** and **OS**, the algorithm and resources are roughly the same, so in most cases, **OS** is similar or better than **DS**. The notable exception to this is pr (reason is explained above). On average, **OS** is 1.9× faster than **DS** for all inputs and benchmarks. Bars **OB** and **OA** show the advantages of using non-vertex, asynchronous programs on the Optane PMM system. Bars **DB** and **OB** show that with the more complex algorithms that can be implemented on the Optane PMM system, performance on this system matches the performance of vertex programs on a cluster with vastly more cores and memory for bc, bfs, kcore, and sssp. The main takeaway is that *Optane PMM enables analytics on massive graphs using shared-memory frameworks out-of-the-box while yielding performance comparable or better than that of a cluster with the same resources as the framework may support more efficient algorithms*.

6.4 Out-of-core GridGraph in App-direct Mode vs. Galois in Memory Mode

In addition to our main shared-memory experiments, we use an out-of-core graph analytics system with app-direct mode on Optane PMM to determine if an out-of-core system that uses Optane PMM as external storage is competitive with a shared-memory framework that uses it as main memory.

Setup. We used state-of-the-art out-of-core graph analytics framework, GridGraph [66] (some out-of-core graph analytics frameworks [63, 57] are faster than GridGraph but they optimize for a subset of algorithms that GridGraph handles). We compare GridGraph in Optane PMM's app-direct (AD) with Galois in memory mode (MM). The Optane PMM machine was configured in AD mode as described in Section 2. In AD, GridGraph manages the available DRAM (memory budget given as 384GB) unlike in MM where DRAM is managed by the OS as another cache level. The input graphs (preprocessed by GridGraph) are stored on the Optane PMM modules which are used by GridGraph during execution. We used a 512 by 512 grid as the partitioning grid for GridGraph (the GridGraph paper used larger grid partitions for larger graphs to better fit blocks into cache). [4] GridGraph uses a `signed 32-bit int` for storing the node IDs, making it impractical for large graphs with $> 2^{31} - 1$ nodes such as wdc12. We conduct a run of bfs and cc: it does not have bc, kcore, or sssp, and we have observed pr failing due to assertion errors in the code.

Table 5: Execution time (sec) of benchmarks in Galois on Optane PMM in Memory Mode (MM) and the out-of-core framework GridGraph on Optane PMM in App-direct Mode (AD). Best times highlighted in green. "—" indicates it did not finish in 2 hours.

Graph	App	GridGraph (AD)	Galois (MM)	Speedup (AD/MM)
clueweb12	bfs	5722.75	6.43	890.0×
	cc	5411.23	11.08	488.4×
uk14	bfs	—	7.22	NA
	cc	5700.48	21.30	267.6×

Results. Table 5 compares the performance of Optane PMM in memory mode (MM) running shared-memory Galois and app-direct mode (AD) running out-of-core GridGraph. We observe that Galois using MM is orders of magnitude faster than GridGraph (GridGraph bfs on uk14 failed to finish in 2 hours). This can be attributed to the more sophisticated algorithms (in particular, non-vertex programs and asynchronous data-driven algorithms are supported in Galois using MM unlike out-of-core frameworks that only support vertex-programs) and the additional IO overhead required by out-of-core frameworks, especially for real-world web-crawls with very high diameter such as clueweb12 (diameter ≈ 500). We note that after a few rounds of computation on bfs for clueweb12, very few nodes get updated: however, the blocks containing its corresponding edges still must be loaded from the storage to be processed. Note that other out-of-core systems [57, 63] that are optimized for a subset of graph algorithms are no more than an order of magnitude faster than GridGraph; therefore, we expect Galois to outperform those systems in Optane PMM as well.

In summary, although out-of-core graph analytics systems can use Optane PMM via app-direct mode, the lack of expressibility and their IO requirement hurt runtime compared to a shared-memory framework that does not have these limitations.

6.5 Summary and Discussion

Our experiments show that graph analytics on Optane PMM is competitive with analytics on both DRAM and distributed clusters. In addition, the monetary cost of Optane PMM technology is much less than that of DRAM[5]. Distributed clusters also have significantly more operating cost. Furthermore, unlike distributed execution which requires a system designed for distributed analytics that

[4] We have tried larger grids, but preprocessing fails as GridGraph opens more file descriptors than the machine supports.

[5] Based on the current pricing, Optane PMM is $\sim 4\times$ cheaper than DRAM for the same memory capacity due to much higher density and cheaper manufacturing cost.

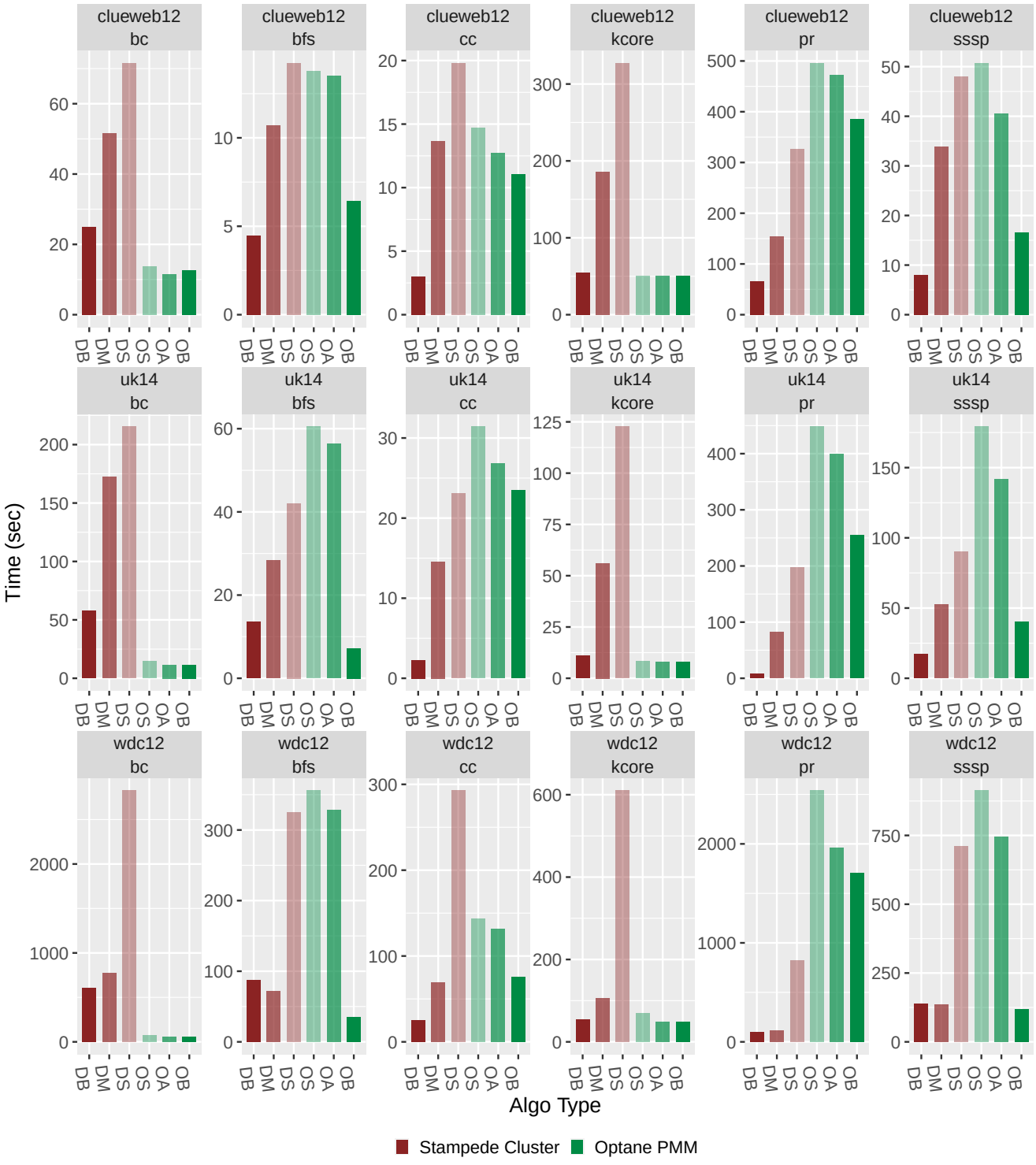


Figure 11: Execution time of benchmarks in Galois on Optane PMM machine and D-Galois on Stampede cluster with different configurations :- DB: Distributed Best (all threads on 256 hosts), DM: Distributed Min (all threads on min #hosts that hold graph), DS: Distributed Same (total 80 threads on min #hosts that hold graph), OS: Optane Same (same algorithm and threads as DS), OA: Optane All (same algorithm as DS, DM, and DB on 96 threads), OB: Optane Best (best algorithm on 96 threads).

typically restricts algorithmic choices for graph applications, Optane PMM allows running shared memory graph analytics *without changes to existing programs*. In other words, writing graph applications for Optane PMM is as easy as that for DRAM, whereas programming for distributed (or out-of-core) execution is much more complicated. Thus, Optane PMM is easy-to-use, affordable, and provides excellent performance for massive graph analytics.

Our study uses only a selection of graph benchmarks that exist in current graph analytics systems, but our findings generalize to other graph analytics benchmarks, including those with more attributes similar to edge weights in sssp. Additional attributes will increase the memory footprint and require more memory accesses: therefore, our principles of reducing memory accesses with efficient data-driven algorithms, accounting for near-memory cache hit rates, and intelligent NUMA allocation would matter even more with the increased memory footprint.

While our study was specific to Optane PMM, the guidelines below *apply to other large-memory analytics systems as well.*

- Studies using synthetic power-law graphs like kron and rmat can be misleading because unlike these graphs, large real-world web-crawls have large diameters. (Section 5)
- For good performance on large diameter graphs, the programming model must allow application developers to write work-efficient algorithms that need not be vertex programs, and the system must provide data structures for sparse worklists to enable asynchronous data-driven algorithms to be implemented easily. (Section 5)
- On large-memory NUMA systems, the runtime must manage memory allocation instead of delegating it to the OS. It must exploit huge pages and NUMA blocked allocation. NUMA migration is not useful. (Section 4)

Finally, our study identifies several avenues for future work. As Optane PMM suffers if near-memory is not utilized well, techniques can be developed to improve near-memory hit rate to increase efficiency of graph analytics on Optane PMM. In addition, this study focused on memory mode and showed graph analytics in memory mode is faster than that in app-direct mode. Work remains to be done to determine the best manner of using app-direct mode.

7. RELATED WORK

Shared-Memory Graph Processing. Shared-memory graph processing frameworks such as Galois [44], Ligra [52, 22], and GraphIt [64] provide users with abstractions to do graph computations that leverage a machine's underlying properties such as NUMA, memory locality, and multicores. Shared-memory frameworks are limited by the available main memory on the system in which it loads the graph into memory for processing: if a graph cannot fit, then out-of-core or distributed processing must be used. However, if the graph fits in memory, the cost of shared memory systems is less than out-of-core or distributed systems as they do not suffer disk reading overhead or communication overhead, respectively.

Optane PMM increases the memory available to shared-memory graph processing systems, and our evaluation shows that algorithms run with Optane PMM are competitive or better than D-Galois [20], a state-of-the-art distributed graph analytics system. This is consistent with past work in which it was shown that shared-memory graph processing on large graphs can be efficient [22], and our findings extend to cases where a user has large amounts of main memory (it is not limited to Optane PMM).

Out-of-core Graph Processing. Out-of-core graph processing systems such as GraphChi [33], X-Stream [49], GridGraph [66], Mosaic [38], Lumos [57], CLIP [2], and BigSparse [31] compute by loading appropriate portions of a graph into memory and writing back out to disk in a disciplined manner to reduce disk overhead. Therefore, these systems are not limited by the main memory like shared-memory systems. The overhead of disk operations, however, greatly impacts performance compared to shared-memory systems.

Distributed Graph Processing. Distributed graph systems such as PowerGraph [26], Gemini [65], D-Galois [20], and others [25, 39, 60] process large graphs by distributing the graph among many machines which increases both available memory and computational power. However, communication among the machines is required, and this can add significant runtime overhead. Additionally, getting access to a distributed cluster can be expensive to an average user.

Persistent Memory. Prior work on non-volatile memory includes persistent memory file systems [14, 61, 23, 12]; efficient, semantically consistent access to persistent memory [13, 56, 32, 41]; and database systems in persistent memory [3, 62, 55].

Optane PMM Evaluation. Many studies have evaluated the potential of Optane PMM for different application domains. Izraelevitz et al. [30] presented an analysis of the performance characteristics of Optane PMM with evaluation on SPEC 2017 benchmarks, various file systems, and databases. Optane PMM has also been evaluated for HPC applications with high memory and I/O bottlenecks [58, 59]. Malicevic et al. [40] use app-direct-like mode to study graph analytics applications using emulated NVM system. However, they only study vertex programs on very small graphs. Peng et al. [46] evaluates the performance of graph analytics applications on Optane PMM in memory mode. However, they only use artificial Kronecker [34] and RMAT [10] generated graphs, which, as we have shown in this work, exhibit different structural properties compared to real-world graphs.

Our study evaluates graph applications on large real-world graphs using efficient and sophisticated algorithms (in particular, non-vertex programs and asynchronous data-driven algorithms) and is also the first work, to our knowledge, to compare performance of graph analytics applications on Optane PMM using Galois with the state-of-the-art distributed graph analytics framework, D-Galois [20], on a production level cluster (Stampede [53]) as well as to the out-of-core framework, GridGraph [66], on Optane PMM.

8. CONCLUSIONS

This paper proposed guidelines for high-performance graph analytics on Intel's Optane PMM memory and highlighted the principles that apply to graph analytics for all large-memory settings. In particular, our study shows the importance of NUMA-aware memory allocation at the application level and avoiding kernel overheads for Optane PMM because poor applications of both concepts are more expensive on Optane PMM than on DRAM. In addition, it shows the importance of non-vertex, asynchronous graph algorithms for large memory systems as synchronous vertex programs do not scale well as graphs grow. We believe that Optane PMM is a viable alternative to clusters with similar computational power for graph analytics because they support a wider range of efficient algorithms while providing competitive end-to-end performance.

Acknowledgments

This research was supported by the NSF grants 1406355, 1618425, 1705092, and 1725322, and by the DARPA contracts FA8750-16-2-0004 and FA8650-15-C-7563. We used the XSEDE grant ACI-1548562 through allocation TGCIE170005. We thank Intel for providing the Intel Optane DC PMM machine. We thank the anonymous reviewers for their many suggestions in improving this paper.

9. REFERENCES

[1] Graph 500 Benchmarks, 2017.

[2] Z. Ai, M. Zhang, Y. Wu, X. Qian, K. Chen, and W. Zheng. Squeezing out All the Value of Loaded Data: An Out-of-core Graph Processing System with Reduced Disk I/O. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 125–137, Santa Clara, CA, July 2017. USENIX Association.

[3] J. Arulraj, M. Perron, and A. Pavlo. Write-behind logging. *PVLDB*, 10(4):337–348, 2016.

[4] A. Azad, G. A. Pavlopoulos, C. A. Ouzounis, N. C. Kyrpides, and A. Buluç. HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks. *Nucleic Acids Research*, 46(6):e33–e33, 01 2018.

[5] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 12:1–12:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[6] S. Beamer, K. Asanovic, and D. A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015.

[7] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[8] P. Boldi and S. 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.

[9] E. G. Boman, K. D. Devine, and S. Rajamanickam. Scalable matrix computations on large scale-free graphs using 2d graph partitioning. In *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, Nov 2013.

[10] D. Chakrabarti, Y. Zhan, and C. Faloutsos. *R-MAT: A Recursive Model for Graph Mining*, pages 442–446.

[11] D. Chazan and W. Miranker. Chaotic relaxation. *Linear Algebra and its Applications*, 2(2):199 – 222, 1969.

[12] Y. Chen, J. Shu, J. Ou, and Y. Lu. Hinfs: A persistent memory file system with both buffering and direct-access. *ACM Trans. Storage*, 14(1):4:1–4:30, Apr. 2018.

[13] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, New York, NY, USA, 2011. ACM.

[14] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[15] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, editors. *Introduction to Algorithms*. MIT Press, 2001.

[16] I. Corporation. Intel® vtune™ platform profiler analysis, 2019.

[17] I. Corporation. Intel® vtune™ amplifier, 2019.

[18] I. Corporation. Ipmctl Intel® Optane™ DC Persistent Memory, 2019.

[19] N. S. Dasari, R. Desh, and M. Zubair. Park: An efficient algorithm for k-core decomposition on multicore processors. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 9–16, Oct 2014.

[20] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '18, pages 752–768, New York, NY, USA, 2018. ACM.

[21] R. Dathathri, G. Gill, L. Hoang, V. Jatala, K. Pingali, V. K. Nandivada, H. Dang, and M. Snir. Gluon-async: A bulk-asynchronous system for distributed and heterogeneous graph analytics. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 15–28, 2019.

[22] L. Dhulipala, G. E. Blelloch, and J. Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, pages 393–404, New York, NY, USA, 2018. ACM.

[23] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[24] G. Gill, R. Dathathri, L. Hoang, and K. Pingali. A study of partitioning policies for graph analytics on large-scale distributed platforms. *PVLDB*, 12(4):321–334, 2018.

[25] Apache Giraph. http://giraph.apache.org/, 2013.

[26] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[27] L. Hoang, R. Dathathri, G. Gill, and K. Pingali. CuSP: A Customizable Streaming Edge Partitioner for Distributed Graph Analytics. In *Proceedings of the 33rd IEEE International Parallel and Distributed Processing Symposium*, IPDPS 2019, 2019.

[28] L. Hoang, V. Jatala, X. Chen, U. Agarwal, R. Dathathri, G. Gill, and K. Pingali. DistTC: High Performance Distributed Triangle Counting. In *2019 IEEE High Performance Extreme Computing Conference (HPEC 2019)*, HPEC '19. IEEE, 2019.

[29] L. Hoang, M. Pontecorvi, R. Dathathri, G. Gill, B. You, K. Pingali, and V. Ramachandran. A Round-Efficient Distributed Betweenness Centrality Algorithm. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'19)*, PPoPP, New York, NY, USA, 2019. ACM.

[30] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*,

abs/1903.05714, 2019.

[31] S. W. Jun, A. Wright, S. Zhang, S. Xu, and Arvind. Bigsparse: High-performance external graph analytics. *CoRR*, abs/1710.07736, 2017.

[32] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 399–411, New York, NY, USA, 2016. ACM.

[33] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[34] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani. Kronecker graphs: An approach to modeling networks. *J. Mach. Learn. Res.*, 11:985–1042, Mar. 2010.

[35] Linux. Mmap.

[36] Linux. Numactl.

[37] Linux. Transparent huge pages: Linux kernel.

[38] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 527–543, New York, NY, USA, 2017. ACM.

[39] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings ACM SIGMOD Intl Conf. on Management of Data*, SIGMOD '10, pages 135–146, 2010.

[40] J. Malicevic, S. Dulloor, N. Sundaram, N. Satish, J. Jackson, and W. Zwaenepoel. Exploiting NVM in Large-scale Graph Analytics. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '15, pages 2:1–2:9, New York, NY, USA, 2015. ACM.

[41] A. Memaripour, A. Badam, A. Phanishayee, Y. Zhou, R. Alagappan, K. Strauss, and S. Swanson. Atomic in-place updates for non-volatile main memories with kamino-tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 499–512, New York, NY, USA, 2017. ACM.

[42] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer. Web data commons - hyperlink graphs, 2012.

[43] U. Meyer and P. Sanders. Delta-stepping: A parallel single source shortest path algorithm. In *Proc. European Symposium on Algorithms*, ESA '98, pages 393–404, 1998.

[44] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM.

[45] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[46] I. B. Peng, M. B. Gokhale, and E. 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. ACM.

[47] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The TAO of parallelism in algorithms. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, PLDI '11, pages 12–25, 2011.

[48] T. L. Project. The ClueWeb12 Dataset, 2013.

[49] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, New York, NY, USA, 2013. ACM.

[50] L. G. Shapiro. Connected component labeling and adjacency graph construction. In T. Y. Kong and A. Rosenfeld, editors, *Topological Algorithms for Digital Image Processing*, volume 19 of *Machine Intelligence and Pattern Recognition*, pages 1 – 30. North-Holland, 1996.

[51] Y. Shiloach and U. Vishkin. An o(logn) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57 – 67, 1982.

[52] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, 2013.

[53] D. Stanzione, B. Barth, N. Gaffney, K. Gaither, C. Hempel, T. Minyard, S. Mehringer, E. Wernert, H. Tufo, D. Panda, and P. Teller. Stampede 2: The Evolution of an XSEDE Supercomputer. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, PEARC17, pages 15:1–15:8, New York, NY, USA, 2017. ACM.

[54] S. Stergiou, D. Rughwani, and K. Tsioutsiouliklis. Shortcutting label propagation for distributed connected components. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, WSDM '18, pages 540–546, New York, NY, USA, 2018. ACM.

[55] A. van Renen, V. Leis, A. Kemper, T. Neumann, T. Hashida, K. Oe, Y. Doi, L. Harada, and M. Sato. Managing non-volatile memory in database systems. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1541–1555, New York, NY, USA, 2018. ACM.

[56] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[57] K. Vora. LUMOS: Dependency-Driven Disk-based Graph Processing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 429–442, Renton, WA, July 2019. USENIX Association.

[58] M. Weiland, H. Brunst, T. Quintino, N. Johnson, O. Iffrig, S. Smart, C. Herold, A. Bonanni, A. Jackson, and M. 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, pages 76:1–76:19, New York, NY, USA, 2019. ACM.

[59] K. Wu, F. Ober, S. Hamlin, and D. Li. Early evaluation of

intel optane non-volatile memory with HPC I/O workloads. *CoRR*, abs/1708.02199, 2017.

[60] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, 2013.

[61] J. Xu and S. Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, 2016. USENIX Association.

[62] M. Zarubin, T. Kissinger, D. Habich, and W. Lehner. Efficient compute node-local replication mechanisms for nvram-centric data structures. In *Proceedings of the 14th International Workshop on Data Management on New Hardware*, DAMON '18, pages 7:1–7:9, New York, NY, USA, 2018. ACM.

[63] M. Zhang, Y. Wu, Y. Zhuo, X. Qian, C. Huan, and K. Chen. Wonderland: A novel abstraction-based out-of-core graph processing system. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 608–621, New York, NY, USA, 2018. Association for Computing Machinery.

[64] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe. Graphit: A high-performance graph dsl. *Proc. ACM Program. Lang.*, 2(OOPSLA):121:1–121:30, Oct. 2018.

[65] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A Computation-centric Distributed Graph Processing System. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 301–316, Berkeley, CA, USA, 2016. USENIX Association.

[66] X. Zhu, W. Han, and W. Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 375–386, Santa Clara, CA, 2015. USENIX Association.