PERFORMANCE ANALYTICS OF GRAPH ALGORITHMS USING INTEL OPTANE DC PERSISTENT MEMORY

by

Evan Unmann

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Computer Science

Charlotte

2021

Approved by:

_____
Dr. Erik Saule

_____
Dr. Dong Dai

_____
Dr. Yonghong Yan

ABSTRACT

EVAN UNMANN. Performance analytics of graph algorithms using intel optane dc persistent memory. (Under the direction of DR. ERIK SAULE)

The increased size of data sets is driving the need for increased memory storage and speed. Persistent memory, PMEM, has emerged as a new solution to the problem. It places itself in between main memory and storage in the memory hierarchy. PMEM offers the ability to increase the amount of addressable memory or the use of dynamic random access memory, DRAM, as a hardware managed cache with PMEM acting as main memory for the machine. In both cases, it becomes paramount to understand the strengths and weaknesses of PMEM. As PMEM is different hardware from DRAM, it has different performance which implies that code written for DRAM might not be optimized for PMEM. This manuscript delves into the performance metrics of PMEM to determine its effectiveness for standard graph algorithms. It also will serve as a guideline as to how to optimally use PMEM and when PMEM becomes advantageous to use.

To overcome the physical storage limits of memory, solutions such as distributed computing and out-of-core computing arose. Distributed computing allows multiple nodes of a computing cluster to compute and communicate in parallel. The disadvantage of this approach becomes the communication overhead between nodes. Out-of-core computing adds layers of slower memory closer to the node to avoid distributed computing. The additional memory typically comes in the form of hard drives or solid-state drives, SSD. Because of how programs interact with drives, code needs to be rewritten to take advantage of this approach, including accessing the memory, reordering the data in a more efficient read order, and localizing computations to minimize reading from drives. PMEM offers the first solution in which current programs can fully utilize the memory with little to no changes in code. This of-

fers a plug-and-play solution where distributed computing and out-of-core computing cannot.

This manuscript measures the performance of PMEM by running benchmarks to measure bandwidth, latency, and graph algorithm performance. Each benchmark is ran on PMEM and DRAM to compare against each other. The graphs selected for benchmarking are a combination of real-world graphs. Results shows that PMEM can be utilized to achieve performance comparable to DRAM.

DEDICATION

This manuscript is dedicated to my family, friends, girlfriend, and our pets for their never ending support and encouragement. Their constant supply of motivation drives me to be a better person academically and personally. I hope this serves as a testament to how much each one of you inspire me.

ACKNOWLEDGEMENTS

I would like to acknowledge my advisor Dr. Erik Saule for guiding me on this project, as well as the committee members Dr. Dong Dai and Dr. Yonghong Yan.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API An acronym for Application Programming Interface.

BFS An acronym for breadth-first search.

CRS An acronym for compressed row storage.

DD An abbreviation for the graph benchmark configuration where the graph and temporary variables in DRAM.

DDR4 An acronym for Double Data Rate 4.

DIMM An acronym for dual in-line memory module.

DP An abbreviation for the graph benchmark configuration where the graph in DRAM and temporary variables in PMEM.

DRAM An acronym for dynamic random access memory.

HPC An acronym for high performance computing.

HT An acronym for hyperthreading.

NHT An acronym for no hyperthreading.

PD An abbreviation for the graph benchmark configuration where the graph in PMEM and the temporary variables in DRAM.

PMDK An acronym for Persistent Memory Development Kit.

PMEM An abbreviation for persistent memory.

PP An abbreviation for the graph benchmark configuration where the graph and temporary variables in PMEM.

SIMD An acronym for single instruction multiple data.

SNAP An acronym for Stanford network analysis platform.

SpMM An acronym for sparse matrix-matrix multiplication.

SpMV An acronym for sparse matrix-vector multiplication.

SSD An acronym for solid-state drive.

TEPS An acronym for traversed edges per second.

CHAPTER 1: INTRODUCTION

In current times, there is an abundance of data. It seems there is an infinite amount of ways to gather information, form it into a data structure, and analyze it. As this continues, it generates an overwhelming need to compute optimally and at large scale. Typically, distributed memory systems are used to solve these problems. Each machine starts with a partition of data, and then sends and receives information as needed until the solution is found. In theory, adding more machines increases the parallelism, thus reducing the overall time of the computation. In practicality, most systems and algorithms do not scale well and are limited by a requirement, such as cost or space. So in an effort to both optimize computation and practical means, specialized hardware is created. This manuscript examines a new form of memory hardware named persistent memory, PMEM.

For many large scale computations, distributed memory architecture is required, as the entire data set cannot fit onto one machine. This allows the use of multiple nodes to compute in parallel. However, the major disadvantage of this approach is when the nodes need to communicate. For example, if a node needs data in which it does not have, but another node does, it needs to stall until it can ask and retrieve the data. Additionally, the bandwidth between nodes is low, so the system scales negatively with how much communication is required. Furthermore, programs need to be designed to take full advantage of distributed computing, so scaling from a single node to multiple nodes requires large amounts of effort. Another solution for data not being able to fit into main memory is out-of-core computing. With this solution, an additional layer of storage is added that can store all of the data. In this way, there can be a single node so all communication overhead from distributed computing is removed. This solution

can offer performance exceeding distributed systems. A drawback of this approach is the effort required to get that performance. For out-of-core computing, the additional layer of storage is typically a hard drive or SSD, which are typically slow. Accessing memory from these drives requires communication with the operating system, so incorporating this method would require dramatic restructuring of code. This often includes a layer of abstraction where parts of the graph are cached by software in main memory. Furthermore, to gain the performance, the data must be structured in such a way to minimize accesses to the drive. This requires preprocessing of the data.

PMEM inserts itself into the memory hierarchy in between main memory and storage. It offers a unique combination of acting similar to main memory and storage in that it can be directly addressable and also persist data between power cycles. It can be setup as main memory and used without any other changes to the machine. Also, it can be setup as an extension of main memory, where a simple library is used to allocate memory for use exactly like allocating memory using mmap. In this method, the memory would persist and is accessible for later use.

Another major benefit of using PMEM is cost. In distributed systems, each node is an entire computer which is expensive. Reducing the number of nodes significantly reduces costs in terms of initial costs and maintenance. For example, if each node has 1.5TB of main memory, then 4 machines are required to compute in parallel on 6TB of data. A single machine with PMEM can have 6TB of main memory, so the cluster of 4 nodes can be reduced to a single node. This completely removes the costs of 3 nodes. Additionally, PMEM does not require power to keep state, meaning it does not require a refresh like volatile DRAM. This can lead to power efficient storage of the data in main memory, reducing maintenance costs. Even if PMEM cannot fit the data into a single machine, it can still reduce the number of nodes required for distributed computing.

Graph algorithms were chosen specifically because of their access pattern nature, varying sizes, distinct characteristics, and wide-spread use. Graph algorithms tend to hop from vertex to vertex by edges. For the machine, this creates non-sequential memory accesses, which stresses memory performance. However, for a benchmark, it helps to test the latency of the memory. Some graph algorithms perform operations per vertex, creating sequential memory accesses and effectively computing a sparse matrix-matrix multiplication, SpMM, or sparse matrix-vector multiplication, SpMV. In this case, it helps to benchmark bandwidth. In both cases, benchmarking against those algorithms gives a general idea of how the memory will perform in real world situations. Distributed memory solutions for large scale graph algorithms see dramatic decrease in performance when scaled from shared memory to distributed memory systems. Since access patterns in graph algorithms are typically not sequential but disordered, significant communication between machines is required. That overhead causes the dramatic decrease in performance. The two graph algorithms chosen are PageRank and breadth-first search, BFS.

1.1 Related Works

An evaluation of PMEM on data structure primitives showed that common data structures code needed to be tweaked to perform well on PMEM [1]. They concluded that there is not a design that is universally great, but application specific designs can perform well. Furthermore, they found that certain operations tend to be expensive and should be avoided, such as PMEM allocation and persist transactions. In terms of algorithm design, much of the typical DRAM guidelines are suggested for PMEM, like locality.

A study of PMEM was conducted [2]. It includes an exhaustive study into PMEM, measuring latency, bandwidth, and database performance in both Memory Mode and App Direct Mode. The research shows that using PMEM instead of SSDs significantly increases performance. Also, it shows using PMEM as main memory shows no

sign of degradation until the DRAM cache misses become frequent. Performance of App Direct Mode was comparable to DRAM performance. They observe that "the major performance difference between Optane DC memory and previous storage media means that software modifications at the application level may reap significant performance benefits" [2].

An evaluation of PMEM for graph applications using Memory Mode was conducted [3]. It showed that PMEM can outperform distributed systems on large graphs. The geometric mean of speedup over the distributed system was 1.7x. Furthermore, the paper showed using PMEM as an out-of-core solution was significantly slower than using PMEM in Memory Mode.

A metric to determine the scalability of distributed systems was developed [4]. The metric offered insight for when a distributed system actually becomes more useful, after considering the overhead the system introduces. The paper continues to show that some modern distributed systems must have a significant amount of cores before it becomes effective. And for some configurations, the distributed system will always under perform.

Research into out-of-core computing found that integrating larger but slower memory closer to the node can significantly reduce performance bottlenecks in large distributed memory systems [5]. Some algorithms are bound by the performance of memory, the CPU cores wait idle for data to arrive. In these cases, reducing the amount of idle time is key to increase performance. Since PMEM offers the unique feature of increasing addressable memory, it is possible to store more data closer to the CPU.

GraphChi [6], a disk-based system for efficiently computing on graphs, further exemplifies the efficacy of keeping data local instead of distributed computing. The results showed that a large graph with billions of edges can be efficiently computed upon on a single machine. Again, since the graph can be stored entirely on disk, the

overhead of distributed computing is completely removed.

Locality and accessibility of data are crucial components of a machine's capability to achieve high performance. As the research shows, moving data closer to the CPU removes additional overhead while also simplifying the overall architecture. From these studies, it makes PMEM a viable consideration for aiding this cause.

1.2 Problem Statement

The foremost objective of this manuscript is to understand how to effectively use PMEM. The goal is to create benchmarks capable of measuring critical attributes of the memory, such as bandwidth and latency. Additionally, benchmarks running standard graph algorithms will provide insight on how the memory performs against real-world graphs.

1.3 Intel Optane DC Persistent Memory

Intel Optane DC Peristent memory takes the form of dual in-line memory module, DIMM. It adheres to the Double Data Rate 4, DDR4, physical module specification. It is a form of non-volatile memory, so data persists between power cycles and data storage does not require power. The memory was designed with the intention of being used with 2nd Generation Intel Xeon Scalable processors. Each socketed CPU can have up to 6 modules for a maximum of 3TiB of PMEM.

There are two modes which PMEM can operate in; App Direct Mode and Memory Mode. In App Direct Mode, PMEM is mounted like a drive. It can be accessed similarly to memory mapping to a file. Once opened, the memory is returned as a raw pointer to memory and can be operated on exactly like normal DRAM memory. Writes to PMEM are persisted, but to ensure persistence, a transactional programming paradigm must be followed. The transactions add overhead to writing to PMEM. AES 256 bit hardware encryption is used for security. "The encryption key is stored in a security metadata region on the module and is only accessible by the Intel

Optane DC persistent memory controller. If repurposing or discarding the module, a secure cryptographic erase and DIMM over-write is utilized to keep data form being accessed [7]".

In Memory Mode, PMEM is used as the main memory for the machine. DRAM becomes a hardware managed cache controlled by the CPU memory controller. In this way, DRAM effectively becomes another level of cache for the CPU. However, the persistence feature of PMEM is forfeited in this mode. The key for the AES 256 bit hardware encryption is discarded and regenerated at each boot.

App Direct Mode offers the ability to have persistence like a drive, but much faster latency and bandwidth. Additionally, the user can choose to use the memory as an extension of DRAM. This offers the unique ability to choose where memory is allocated, either from PMEM or DRAM. This manuscript will focus on using PMEM in App Direct Mode without transactions.

The Storage Network Industry Association formed a technical workgroup which created a unified programming model for persistent memory [7]. The three main capabilities of the model designed are "The management path allows system administrators to configure persistent memory products and check their health. The storage path supports the traditional storage APIs (Application Programming Interface) where existing applications and file systems need no change; they simple see the persistent memory as very fast storage. The memory-mapped path exposes persistent memory through a persistent memory-aware file system so that applications have direct load/store access to the persistent memory. This direct access does not use the page cache like traditional file systems and has been named DAX by the operating system vendors.[7]"

To adhere to the programming model, the Persistent Memory Development Kit, PMDK, was developed. It offers libraries to solve different use cases of PMEM. For example, libpmemobj offers a library for object based transactions, libpmem offers a

low level access to persistent memory, and libpmemlog provides APIs to log to files easily. PMDK is validated and performance-tuned by Intel.

1.4 Organization

This manuscript is structured as follows. Chapter 1 is an introduction into the topic, detailing current solutions to computing on large data sets. It discusses the weaknesses to those solutions, namely scalability in terms of efficient computing and cost. PMEM is introduced as another solution with potential to increase efficiency in computing while also reducing costs. The problem statement is outlined as investigating PMEM by measuring its performance against DRAM in memory and graph algorithm benchmarks. PMEM is explained in more detail as to its configurations, usage, and libraries which support it.

Chapter 2 describes the environment setting for the experiments. The machine used for benchmarks is described in detail. The outcomes of the memory benchmarks are outlined and discussed.

Chapter 3 explains the setup and outcomes of the graph benchmarks. First, the different type of memory configurations are explained, meaning the combinations of where data resides in memory. Then, PageRank is described with the optimizations that were implemented. Next the results of the PageRank benchmarks are shown and discussed. After, BFS is detailed in two variations, top down and bottom up. The optimizations for both are described. Then the results for both variations are shown and discussed. Finally, a discussion section combines the results of PageRank and BFS for deeper analysis.

Chapter 4 is the conclusion where the results are generalized. Insight into PMEM is offered as well as thoughts about the future of PMEM. Discussion about next steps in terms of research and possible applications are presented.

CHAPTER 2: Environment Setting

2.1 Machine Specifications

The machine consists of dual socketed Intel Xeon Gold 6254 CPUs, 192 GB DRAM, and 768 GB Intel Optane DC Persistent Memory. Each CPU has 18 physical cores, 36 logical cores, operating at 3.1 GHz with hyperthreading enabled. The CPU has an L1 cache size of 1.125MiB, L2 cache size of 18MiB, and an L3 cache size of 24.75MiB. Each CPU socket has 12 DIMM slots. All memory modules are installed on the first socket. Because of this, only the CPU in the first socket is being used for benchmarking, so all threads are bound to that CPU at application startup. There are 6 DIMM modules per memory type, so 6 32GB DRAM modules and 6 128GB PMEM modules. Both DRAM and PMEM are operating at 2666MHz. PMEM is operating in App Direct Mode. The compiler used was GCC 7.5.0. The libpmem library version 1.1 was used. The operating system used was Ubuntu 18.04.5 LTS.

To ensure the proper thread placement, OpenMP offers a specification for thread affinity. The specification allows the user to define a thread allocation strategy before program execution. Each time a thread is spawned, it is assigned to a place. A place can either be a hardware thread, core, or a socket. This allows the user to define exactly where threads will execute. For the purposes of this manuscript, it allows the usage of only the first socket, and to run comparisons between using hyperthreading and not. When using hyperthreading, a maximum of 36 threads are spawned and each is allocated to a unique hardware thread. When not using hyperthreading, a maximum of 18 threads are spawned and each is allocated to a unique core. For denote the use of hyperthreading in results, HT is used. To denote the use of no hyperthreading, NHT is used.

2.2 Memory Benchmarks

The benchmarks developed serve to measure the memory's core metrics, bandwidth and latency for reading and writing. These benchmarks provide insight on what the memory is capable of while also showing its strength and weaknesses. Additionally, the STREAM[8] benchmarks were modified to run on PMEM. Table 2.1 shows the peak performance claimed by Intel [7].

Table 2.1: A table of peak bandwidth per module of PMEM.

Module Capacity	128GB	256GB	512GB
Bandwidth Read 256 B	6.8 GB/s	6.8 GB/s	5.3 GB/s
Bandwidth Write 256 B	1.85 GB/s	2.3 GB/s	1.89 GB/s
Bandwidth Read 64 B	1.7 GB/s	1.75 GB/s	1.4 GB/s
Bandwidth Write 64 B	0.45 GB/s	0.58 GB/s	0.47 GB/s

The memory benchmarks first run using DRAM, then PMEM on the same machine. This allows a side-by-side comparison while ensuring all other hardware is exactly the same. To account for variability in results, each benchmark is ran 10 times and the average of the results are reported. The memory allocations are aligned to 16B for both DRAM and PMEM.

The read bandwidth benchmark measures the maximum read bandwidth from memory to the CPU. To increase the amount of memory loaded per instruction, the inputted array is casted to a larger element type of size 32B. Then, the summation of the array is performed. Since the addition takes less cycles to perform than loads and occurs in parallel with the loads, this effectively measures the time to load the array from memory to the CPU. To completely stress the system, the summation and loading is performed on all threads simultaneously by partitioning the array across all threads. Since the access pattern is iterative, the memory controller can prefetch values from memory, which maximizes memory usage. The bandwidth is calculated by $B = \frac{N}{t}$ where N is the size of the array in bytes, t is the time elapsed.

The read latency benchmark measures the average latency of one byte from memory

to the CPU. To ensure that the memory being loaded is not in the cache, the access pattern of the memory is randomized. In this way, the hardware memory controller is unlikely to prefetch the next value in memory while also making it highly unlikely that the same value in memory is loaded twice. In this way, the CPU's cache is avoided and the measurement is the time it takes for the value in memory to reach the CPU. Furthermore, the next random index generated relies on the current value loaded from memory. This forces the CPU to wait until the current load completes before issuing the next load. The latency is computed by $L = \frac{t}{N}$ where N is the number of bytes loaded, t is the time elapsed. This effectively computes the average latency for all loads.

The write bandwidth benchmark measures the maximum write bandwidth from the CPU to memory. To increase the amount of memory stored per instruction, the inputted array is casted to a larger element type of size 32B. Then, the CPU issues store instructions iteratively from the first element to the last in memory. To completely stress the system, the store instructions are performed on all threads simultaneously by partitioning the array across all threads. The bandwidth is computed the same as the read bandwidth.

2.3 Results

Tables 2.2 and 2.3 shows the results of running the memory benchmarks with and without hyperthreading. It is clear that bandwidth is maximize when not using hyperthreading. According to Intel, the maximum read bandwidth 6 modules is 40.8 GB/s and write bandwidth is 11.1 GB/s. The benchmark recorded a read bandwidth of 40.161 GB/s and write bandwidth of 3.047 GB/s. The measured read bandwidth is nearly as high as Intel's claims. However, the write bandwidth is 3.6x slower. From these results, it seems that writing to PMEM is costly and should be avoided if possible. The read bandwidth of PMEM is 2.6x slower than DRAM, but should be sufficient for computationally bound algorithms. The latency for PMEM is 2.8x

slower than DRAM, making cache misses more detrimental.

Table 2.2: A table of results from the memory benchmarks with HT.

Benchmark	DRAM	PMEM	Ratio
Read Bandwidth	65.480 GB/s	32.669 GB/s	0.50
Read Latency	142.02 ns	403.83 ns	2.84
Write Bandwidth	42.782 GB/s	1.254 GB/s	0.03
STREAM Copy	53.928 GB/s	2.613 GB/s	0.05
STREAM Scale	47.512 GB/s	1.288 GB/s	0.03
STREAM Add	54.286 GB/s	3.987 GB/s	0.07
STREAM Triad	51.161 GB/s	2.180 GB/s	0.04

Table 2.3: A table of results from the memory benchmarks with NHT.

Benchmark	DRAM	PMEM	Ratio
Read Bandwidth	106.815 GB/s	40.161 GB/s	0.38
Read Latency	137.14 ns	403.70 ns	2.94
Write Bandwidth	73.715 GB/s	3.047 GB/s	0.04
STREAM Copy	66.715 GB/s	5.523 GB/s	0.08
STREAM Scale	66.717 GB/s	5.555 GB/s	0.08
STREAM Add	75.298 GB/s	8.093 GB/s	0.11
STREAM Triad	75.367 GB/s	8.185 GB/s	0.11

CHAPTER 3: GRAPH BENCHMARKS

The graph algorithm benchmarks serve to show the performance of memory for standard algorithms. Graph algorithms were selected particularly for their irregular access patterns. Each graph is stored in memory as column row storage, CRS. Since most graphs tend to be sparse in matrix form, the CRS format reduces the overall memory size of the graph. In addition, this memory format gives the opportunity to use single instruction multiple data, SIMD, instructions for more parallelism. Real-world graphs were gathered from SNAP [9]. Those graphs were collected from social media websites. Details of the graphs are shown in Table 3.1.

Table 3.1: A table of real world graphs used for the graph benchmarks.

Graph	Vertices	Edges	Directed
Facebook	4,039	88,234	No
Epinions	75,879	508,837	Yes
Pokec	1,632,803	30,622,564	Yes
Stack Overflow	1,632,803	30,622,564	Yes
Live Journal	4,847,571	68,993,773	Yes
Orkut	3,072,441	117,185,083	No

For most algorithms, the use of temporary variables is required. For both of the graph algorithms implemented, this is especially true. In an effort to better understand the performance of the memory, each benchmark is ran in four combinations of using DRAM or PMEM. The combinations correspond to where the graph and temporary variables reside in memory. For this manuscript, all combinations were tested. Table 3.2 outlines the different combinations. A key insight to running these combinations is rooted in the operations that take place. In the case of the graph, it is never modified. So the graph essentially becomes read-only and hence the only difference between the DD and PD combinations becomes how fast the graph can be

read from the memory. The temporary variables require read and write operations which allows observations on how reading and writing to memory affect performance.

Table 3.2: A table outlining the different combinations the graph benchmarks were run. It shows where the graph and temporary variables reside in memory.

Abbreviation	Graph	Temporary
DD	DRAM	DRAM
DP	DRAM	PMEM
PD	PMEM	DRAM
PP	PMEM	PMEM

3.1 PageRank

The goal of PageRank is to calculate a rank value for each vertex based on the number and quality of neighbors. In terms of a benchmark, the algorithm iterates through the graph multiple times, overwrites a vector on each iteration, and performs a few arithmetic operations for each vertex. The performance hinges on the bandwidth of the memory, as the reading and writing is sequential and is performed multiple times.

The original purpose of PageRank was to rank web pages for search results on Google [10]. PageRank is an important algorithm because it computes the importance of something in relation to its peers. Therefore, it can be applied to anything in the form of a graph. For example, a variation can compute the impact factor of research papers [11]. Also, it can be applied to the analysis of proteins in biology [12]. Additionally, Twitter uses a variation to find a personalized PageRank to find suggestions for its users [13].

Let G be a graph with vertex set V and edge set E. Let d be a number representing a dampening factor. Then, PageRank is $PR(v) = \frac{1-d}{|V|} + d\sum_{u \in N(v)} \frac{PR(u)}{deg(u)}$. That equation can be solved using an iterative method. First, the initial probability at time step $t = 0$ is $PR(v, 0) = \frac{1}{|V|}$. Then, $PR(v, t+1) = \frac{1-d}{|V|} + d\sum_{u \in N(v)} \frac{PR(u,t)}{deg(u)}$ is the PageRank for the next time step. This is done for I iterations or until convergence,

i.e. for some small ϵ, $|PR(v,t) - PR(v,t+1)| < \epsilon$.

PageRank is a SpMV. Since PageRank is typically called multiple times with different dampening factors, it is advantageous to convert it to a SpMM. For example, it was shown that a recommender system benefits from this approach [14]. Upgrading PageRank from SpMV to SpMM is done by creating a matrix of ranks, instead of the normal vector of ranks. By doing this, the core computation can load the graph once per iteration and compute all of the PageRanks, instead of once per iteration per dampening factor. It has been shown that upgrading a SpMV to SpMM increases the number of operation relative to memory bandwidth [15]. Additionally, using a SpMM approach to increase centrality in BFS showed greater performance than a SpMV [16]. So for this manuscript, PageRank is implemented as a SpMM so both the CPU and memory can be utilized efficiently.

For the benchmark, PageRank is always run for $I = 100$ iterations, regardless of convergence. Since the PageRank values are not of interest, it is preferable to just run PageRank for a certain amount of iterations to have consistency across multiple experiments. This also removes the need to compare all PageRank values from the current time step and the previous.

To increase performance the PageRank read and write matrices are of size $|V|\text{x}Z$, where Z is the number of PageRanks being computed. Accessing the first PageRank for a vertex will cause an entire cache line of PageRanks to be pulled. This increases the loading performance when scaling to higher number of PageRanks. Additionally, the code is vectorized based on neighbors of vertices, rather than vectorizing on the PageRanks for a vertex. When vectorizing on neighbors, the vertex's neighbor's degree are computed in parallel. Then, each neighbor's PageRank is read from memory. Because of how the PageRank read matrix is set up, the first gather for all of the PageRanks causes the next PageRanks which will be loaded to also be pulled into the cache line. This causes a sequential load pattern from the PageRank read matrix.

Another optimization is swapping between two buffers for storing the PageRank during each time step. This reduces the need to allocate a new array on each time step iteration. Lastly, the vertices are partitioned across all threads for concurrency, since the computation for each vertex for an iteration is independent of other vertices.

3.1.1 Results

For all PageRank results, the amount of traversed edges per second, TEPS, is reported. The metric for PageRank is computed by $\frac{I \cdot |E| \cdot Z}{t}$, where Z is the number of PageRanks being computed and t is the time elapsed. This value allows the comparison between various number of PageRanks to show if increasing the computational density also increases efficiency. To account for variability in results, each benchmark is ran 10 times and the average of the results are reported.

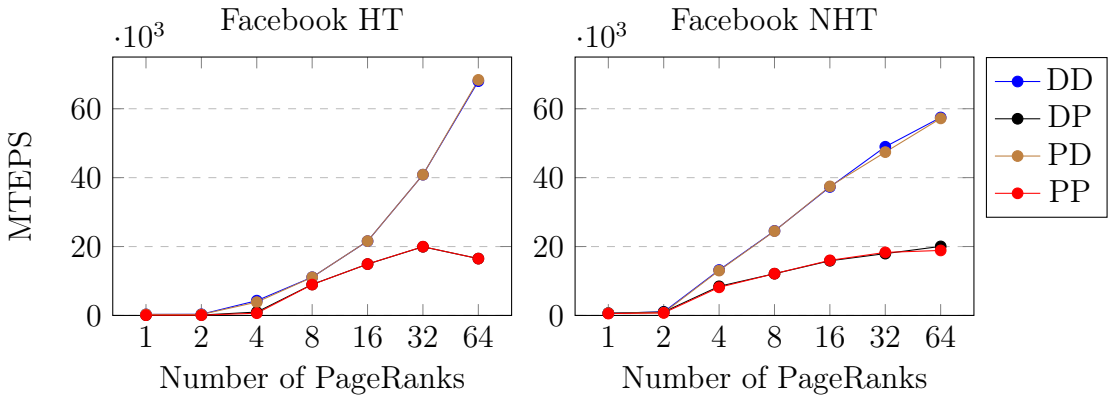

Figure 3.1: A comparison between the PageRank benchmark on the Facebook graph with HT and NHT.

Figure 3.1 shows the results for the Facebook graph. The results are a factor of 3-4 higher than other graphs because the Facebook graph is significantly smaller. The entirety of the graph can be stored in the L3 cache of the CPU. This is also the case for the Epinions graph, results shown in Figure 3.2. The Epinions graph does not see the significant gains of the Facebook graph though. This may be because the Epinions graph plus the temporary variables are evicting each other from the CPU cache.

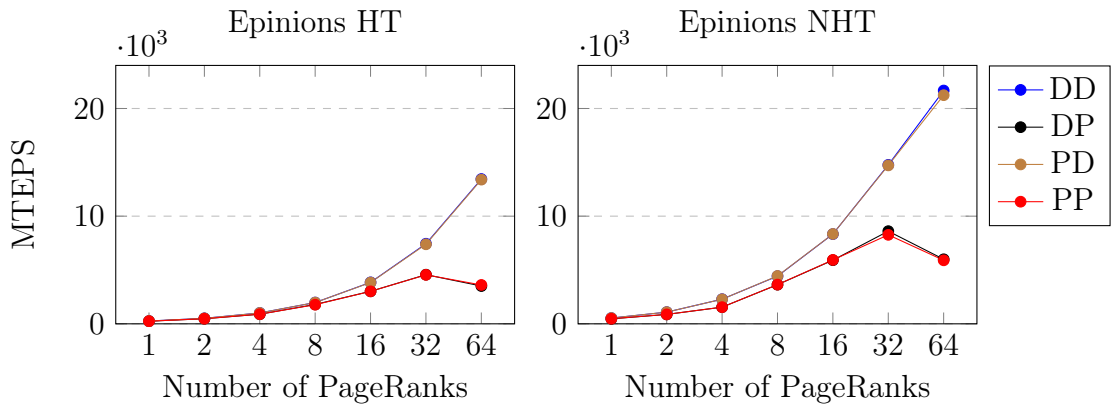

Figure 3.2: A comparison between the PageRank benchmark on the Epinions graph with HT and NHT.

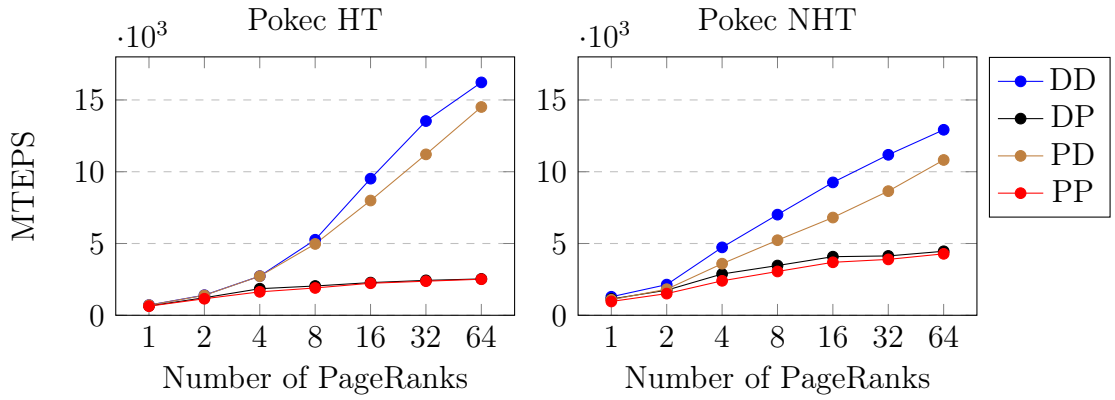

Figure 3.3: A comparison between the PageRank benchmark on the Pokec graph with HT and NHT.

For graphs significantly larger than the CPU cache size, seen in Figures 3.3, 3.4, 3.5, 3.6, a pattern can be seen. Hyperthreading increases performance for all of these graphs when the temporary variables are in DRAM. Hyperthreading degrades performance for all of these graphs when the temporary variables are in PMEM. The DD and PD combinations seem to scale well with the number of PageRanks.

When observing the affects of using temporary variables in PMEM, it is clear that performance is severely hindered. The results for both DP and PP are nearly identical, showing that when the temporary variables are in PMEM, the location of the graph in memory is not as important. Because the performance increases when not using hyperthreading, it is likely that writing to PMEM is the issue. As seen in

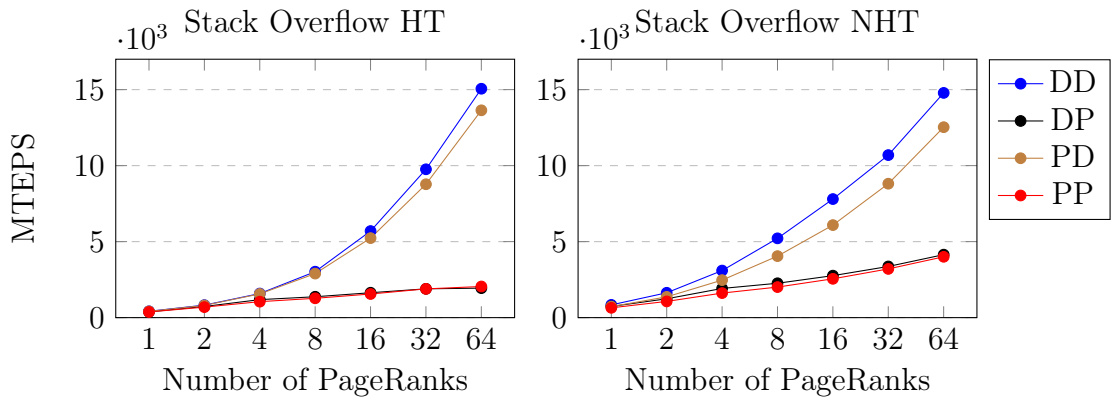

Figure 3.4: A comparison between the PageRank benchmark on the Stack Overflow graph with HT and NHT.

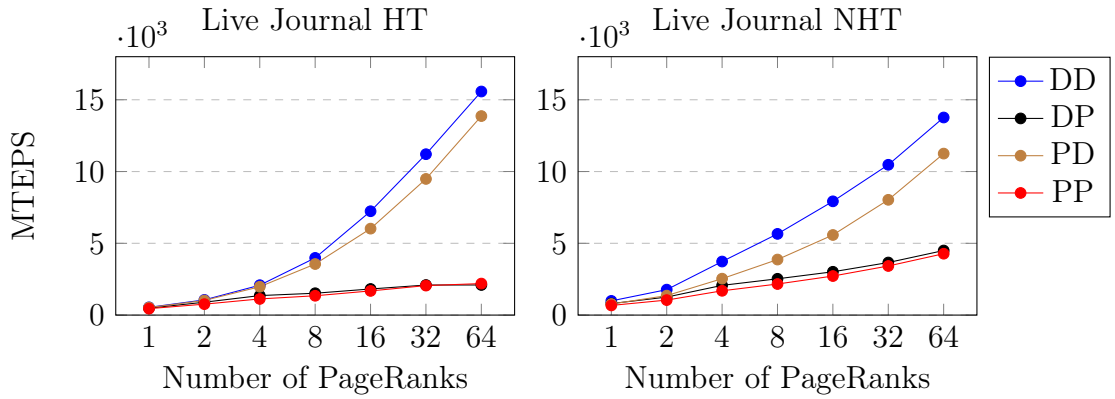

Figure 3.5: A comparison between the PageRank benchmark on the Live Journal graph with HT and NHT.

the measurements for write bandwidth, using hyperthreading decreases performance by over a factor of 2, which is similar to the affect seen for PageRank.

3.2 Breadth-First Search

BFS is a classic graph traversal algorithm. Its popularity and wide-spread use made it a benchmark for Graph 500. The algorithm is key in shortest path searches, such as Bellman-Ford, Dijkstra, and A*. It also can be used to solve problems in other fields. For example, BFS can be used to apply clausal resolution to temporal logic efficiently [17]. Also, it can be used to help plan and optimize work of sale persons [18]. Additionally, it can be applied to help the processing of images [19].

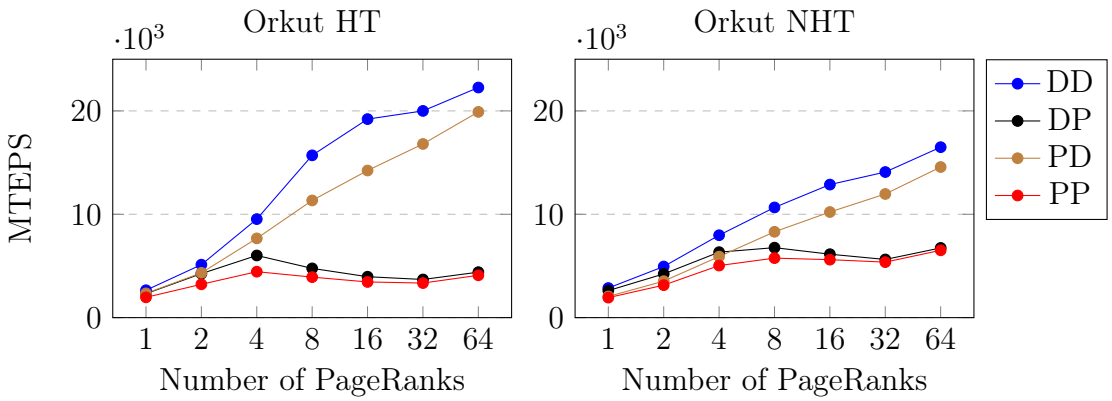

Figure 3.6: A comparison between the PageRank benchmark on the Orkut graph with HT and NHT.

Furthermore, BFS was used to implement a web structure mining algorithm [20].

The goal of BFS is to search for a path between a source vertex and a destination vertex. The algorithm starts by inspecting all neighboring vertices of the source vertex. Each neighbor is added to a frontier queue. Then, a vertex is dequeued from the frontier and checked if it is the destination. If it is, then the search is complete. If not, the vertex's neighbors are all appended to the frontier. This is performed until the frontier is empty or the destination vertex has been reached. This algorithm, named top down BFS, is detailed in Figure 3.7.

Another approach to BFS is to search in the opposite direction. Instead of looking at vertices in the frontier, all unvisited vertices can search for a neighbor in the frontier. If there exists a neighbor, that vertex is added to the next frontier. This is done until the frontier is empty of the destination vertex is found. This algorithm, named bottom up BFS, is detailed in Figure 3.8.

A direction optimized approach to BFS was developed to switch between top down and bottom up directions during a traversal [21]. Heuristics are used to determine when to switch direction at the start of each traversal level. In this way, the optimal direction is used at each level, thus leading to performance gains. The key component to this method is switching when it is actually beneficial. This requires tuning

```
function bfs-top-down(vertices, source)
  frontier ← {source}
  next ← {}
  depth ← [-1,-1,...,-1]
  depth[source] ← 0
  level ← 1
  while frontier ≠ {} do
    for v ∈ frontier do
      for n ∈ neighbors[v] do
        if depth[n] = -1 then
          parents[n] ← level
          next ← next ∪ {n}
        end if
      end for
    end for
    frontier ← next
    next ← {}
    level ← level + 1
  end while
  return depth
```

Figure 3.7: Conventional Top Down BFS

parameters that need to be tuned for each graph. To avoid tuning for each graph and memory combination, the direction optimized BFS is not shown. Instead, both top down and bottom up approaches are benchmarked. This allows the observation of the approaches against the memory combination without the concern of switching optimally.

For the benchmark, BFS does not search for a destination vertex, but continues to traverse the graph until all vertices have been exhausted. During each layer of traversal, the depth of visited vertices to the source vertex is recorded. In this way, the algorithm issues a series of reads and writes to non-sequential parts of memory. In BFS, undirected graphs will check each edge twice where directed graphs will only check each edge once.

To optimized top down BFS, at the start of each traversal layer, the vertices in the frontier are partitioned across all threads and the search is executed concurrently.

```
function bfs-bottom-up(vertices, source)
  frontier ← {source}
  next ← {}
  depth ← [-1,-1,...,-1]
  depth[source] ← 0
  level ← 1
  while frontier ≠ {} do
    for v ∈ vertices do
      if parents[v] = -1 then
        for n ∈ neighbors[v] do
          if n ∈ frontier then
            parents[n] ← level
            next ← next ∪ {v}
            break
          end if
        end for
      end if
    end for
    frontier ← next
    next ← {}
    level ← level + 1
  end while
  return depth
```

Figure 3.8: Conventional Bottom Up BFS

Because of this, additional modifications are required to keep the threads from fighting over resources. The main concern becomes pulling vertices off of the queue. To completely removed any locking behavior to avoid race conditions, there are two queues implemented both as arrays. In this way, one array is the read frontier, and the other becomes the write frontier. That means the vertices in the read frontier can be distributed equally among all threads, without the need to coordinate. It also means that vertices simply can be read from any position in the queue, rather than only at the front. To avoid locking the write frontier, each thread has access to its own local write frontier. Once each thread completes their BFS traversal, the threads combine their local frontiers. This combining also does not require any locking since the number of vertices in all local write frontiers is known. So each

thread can compute a range of indices into the write frontier that is guaranteed to not overlap with other threads. At the end of a level iteration, the read frontier is cleared. Then, the write frontier becomes the read frontier for the next iteration, and the read frontier becomes the write frontier for the next iteration.

To optimize bottom up BFS, at the start of each traversal layer, all vertices are partitioned across all threads and the search is executed concurrently. There exists a read and write frontier, which swap roles each iteration to prevent memory allocations for each iteration. The frontiers are implemented as arrays where each index is mapped to a vertex. This makes the lookup time to check if a vertex in the frontier take constant time. It also removes any concern of race conditions. After each level iteration, the read frontier is cleared.

3.2.1 Results

All results for BFS are reported in TEPS. To compute TEPS, the output of BFS is inspected to determine which vertices were visited. Then, the number of traversed edges can be computed by summing the degree of all visited vertices, S. The metric can be computed as $\frac{S}{t}$. The purpose of this formulation of TEPS as opposed to the traditional approach of $\frac{|E|}{t}$ is because the graphs may be disconnected. If the graph is disconnected, then not all vertices will be traversed, which inflates TEPS. Additionally, checking the number of vertices visited allows runs where only a small portion of the graph were visited to be discarded. These runs typically report TEPS in orders of magnitude higher than a run that traverses significantly more vertices. So the computation for this manuscript provides a more accurate measurement of TEPS while also ensuring enough vertices were traversed. A BFS traversal must visit at least 25% of vertices and 25% of edges in the graph, or else that traversal was not included in the results. Instead, another vertex was selected as the the source vertex. The results are an average of 10 BFS traversals starting at different source vertices.

The results for top down BFS are shown in Figures 3.9 and 3.10. Overall, the

DD combination performs the best with no hyperthreading. PD with hyperthreading performs better or similar compared to PD with no hyperthreading. Both DP and PP perform better with no hyperthreading, similar to PageRank and memory bandwidth results. Comparing the combinations, DP and PP offer significantly worse performance compared to DD and PD. This shows that temporary variables in PMEM causes a massive decrease in performance.

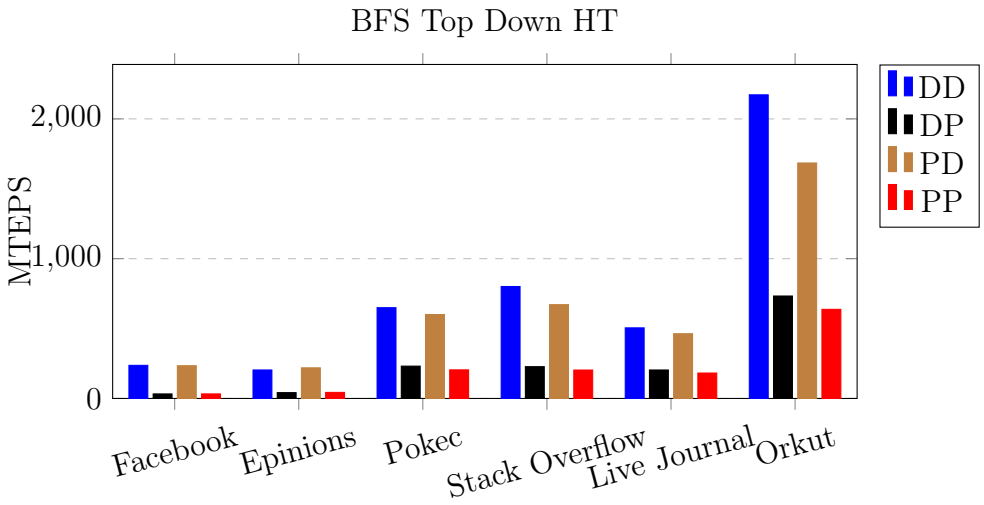

Figure 3.9: Results of the BFS benchmark top down with hyperthreading.

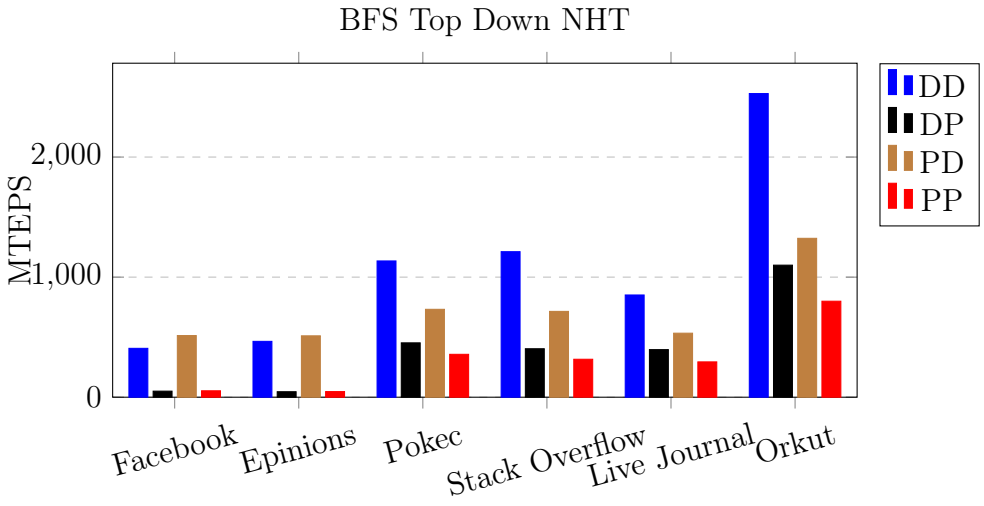

Figure 3.10: Results of the BFS benchmark top down with no hyperthreading.

The results for bottom up BFS are shown in Figures 3.11 and 3.12. Overall, DD

with no hyperthreading performs better than DD with hyperthreading. The other combinations, DP, PD, and PP, have similar results with no hyperthreading having slightly better performance. Similar to top down BFS, the DP and PP combinations slower than the DD and PD combinations, showing temporary variables in PMEM cause a dramatic decrease in performance.

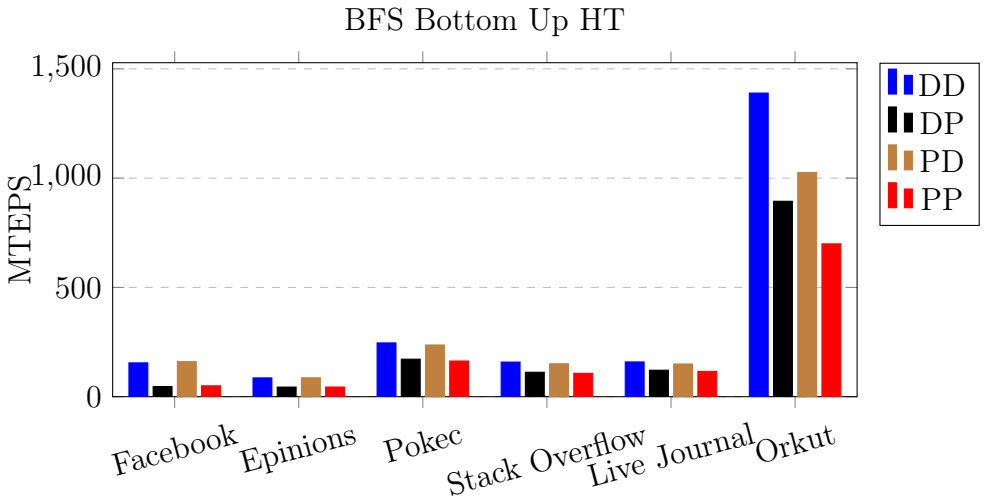

Figure 3.11: Results of the BFS benchmark bottom up with hyperthreading.

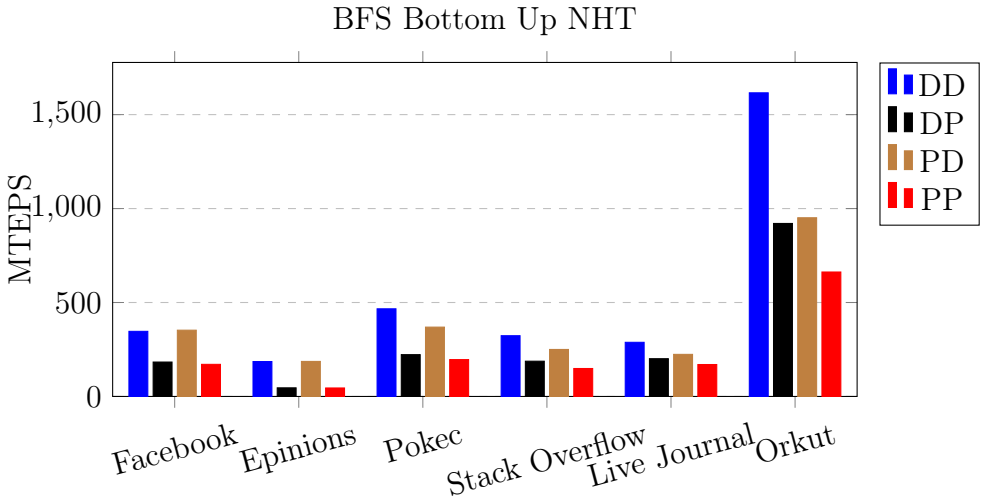

Figure 3.12: Results of the BFS benchmark bottom with no hyperthreading.

3.3 Discussion

From the results of PageRank and BFS, it is clear the best memory combination is DD and PD is second. The only difference between these run combinations is where the graph resides in memory. Additionally, the graph is only read from for both PageRank and BFS. So the differences in these combinations show the loss of performance when only reading the graph from PMEM. So for applications where the data set is mostly read form, and not written to, storing the data set in PMEM offers good performance.

Tables 3.3 shows a comparison of DD and PD combinations for the PageRank benchmark on the Live Journal graph. Even though PMEM bandwidth is a factor 2 slower, it does not seem to be causing a severe drop in performance. Since this is the case, it is likely that bandwidth is not the bottleneck for PageRank. This shows that for computationally bound algorithms, PMEM is a viable option. Furthermore, the performance of PMEM is scaling similar to DRAM.

Table 3.3: A table of results of the PageRank benchmark on the Live Journal graph with HT.

Number of PageRanks	DD	PD	Ratio
1	539.047	525.111	0.97
2	1059.3	1020.588	0.96
4	2083.336	1974.816	0.95
8	3983.992	3556.696	0.89
16	7233.536	6021.776	0.83
32	11207.712	9484.992	0.85
64	15574.336	13864.0	0.89

Table 3.4 shows the results of the BFS top down and bottom up benchmarks on the Live Journal graph. The performance loses were greater than PageRank. This can be explained by the lack of computation in BFS. So in the worst case, where most of the operations are just reading and writing to memory, PMEM is not scaling as bad as the memory benchmarks would suggest. Since BFS is typically used as a backbone

for other algorithms, adding computational density would close the gap between DD and DP.

Table 3.4: A table of results from the BFS benchmark on the Live Journal graph with NHT.

Benchmark	DD	PD	Ratio
Top Down	851.299	533.13	0.63
Bottom Up	288.185	224.028	0.78

As shown in Tables 3.5 and 3.6, hyperthreading performs worse than no hyperthreading for the PP combination. This was consistent between most benchmarks for PageRank and BFS. This suggests that too many threads reading and writing to PMEM concurrently can cause a reduction of performance. When combining the results for PageRank, where hyperthreading actually improved results for PD, this hints that the writing operations might be the cause of this. If reading were the issue, it would have shown in the PD results for PageRank as well. Because of this, it seems that too many threads writing to PMEM concurrently is an issue. In general, writing to PMEM seems to bring the biggest performance loss.

Table 3.5: A table comparing the results of the PageRank benchmark on the Live Journal graph with HT and NHT for the PP combination.

Number of PageRanks	HT	NHT	Ratio
1	448.246	672.942	1.50
2	761.932	1044.704	1.37
4	1124.968	1691.844	1.50
8	1345.904	2164.136	1.61
16	1683.392	2719.584	1.62
32	2054.726	3420.224	1.66
64	2194.752	4275.705	1.95

Summing up all of the results and analysis, it seems that using PMEM in App Direct mode and utilizing both DRAM and PMEM is the best option. It gives finer control over where the data resides in memory. Then, data can be store on PMEM or DRAM based on how much it is written to. PMEM shows great potential for

Table 3.6: A table comparing the results from the BFS benchmark on the Live Journal graph with HT and NHT for the PP combination.

Benchmark	HT	NHT	Ratio
Top Down	181.847	294.115	1.62
Bottom Up	115.667	169.945	1.47

algorithms where a majority of the data is read-only.

CHAPTER 4: CONCLUSIONS

Overall, benchmarks were developed to measure and compare the performance of DRAM and PMEM, including various combinations of each. The memory benchmarks showed the peak performance of both. The graph benchmarks served to show the performance for important foundational algorithms against real world graphs. The various run combinations gave insight into how PMEM is affected by different operations, as well as strategies to efficiently utilized DRAM and PMEM in conjunction.

As shown in this manuscript, PMEM offers performance comparable to DRAM. Even though PMEM read bandwidth is over a factor of 2 slower than DRAM, the DD and PD combinations results for both PageRank and BFS do not show this. PageRank was less affected by the bandwidth difference, especially as the number of PageRanks grew. This offers insight that for algorithms bottlenecked by operations, PMEM is advantageous to use. However, this also means that algorithms will have to be written to increase computational density to achieve better performance if they are more bandwidth bound.

The use of PMEM for temporary variables showed a dramatic degradation of performance. So the use of PMEM in App Direct Mode may be preferred over Memory Mode. App Direct Mode allows greater control over where data resides in memory. Memory Mode does not offer this and performance would hinge on the temporary variables staying in the DRAM level cache. If that data would be evicted and written back to PMEM, it would cause a drop in performance. Because of the lack of control over this, App Direct Mode is preferable for HPC applications. This also means minor changes to code is required such that memory is allocated from PMEM and DRAM

selectively.

Furthermore, the use of PMEM seems advantageous if the memory will be read only. As shown in the memory benchmarks, writing to PMEM is extremely slow. For applications where data is modified minimally, PMEM can offer good bandwidth while giving 4 times the capacity. If PMEM does need to be written to, usage of write optimized algorithms would be preferable.

For BFS, a study into direction optimization should be conducted. It was shown here that BFS in both top down and bottom up offer good performance using PMEM. The next step is to figure out if switching, more specifically the conversion between directions, is efficiently enough. Discovering the tuning parameters for direction optimized BFS will help to understand what operations and ideas are applicable to PMEM performance tuning.

Additionally, a comparison between PMEM, distributed computing, and out-of-core computing should be conducted. A computing cluster of up to 4 nodes should be compared to a single machine with PMEM. In this way, it can be observed whether or not PMEM can be used in place of 4 nodes, as both systems would have the same amount of main memory. A comparison because hardware costs should also be conducted as well as measuring power usage of PMEM.

Future studies can delve further into the performance of using DRAM and PMEM simultaneously. For single machine systems, like the environment for this manuscript, it was shown that using a hybrid approach still has good performance. For mobile devices, or even personal computers, this hybrid approach may be beneficial to use. For example, if the OS can efficiently keep state in PMEM, power cycles would have instant time since the OS does not need to load anything. The PMEM has the previous state and just needs to be accessed. In this way, booting performance can be improved. Furthermore, the power reduced by having state in PMEM can extend the life of mobile devices.

Another study into how to configure machines with both DRAM and PMEM should be conducted. The machine used for these experiments had 12 DIMM per socket. However, the bandwidth of a memory module relies on how many modules are installed. So for each module of PMEM installed, it reduces the bandwidth of DRAM and vice versa. There needs to be careful consideration about how many modules of each are installed in order to maximize performance.

Overall, PMEM is a viable option for HPC situations with massive amounts of data. It has the ability to perform well with computationally bound algorithms with minimal changes to hardware and code. For applications with mostly read operations, PMEM offers comparable performance with the added features of persistence. Because of this, reducing the amount of nodes in a distributed system without performance lost is possible. Additionally, using PMEM over out-of-core computing can be advantageous.

REFERENCES

[1] P. Götze, A. K. Tharanatha, and K.-U. Sattler, "Data structure primitives on persistent memory: an evaluation," in *Proceedings of the 16th International Workshop on Data Management on New Hardware*, pp. 1–3, 2020.

[2] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, *et al.*, "Basic performance measurements of the intel optane dc persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.

[3] G. Gill, R. Dathathri, L. Hoang, R. Peri, and K. Pingali, "Single machine graph analytics on massive datasets using intel optane dc persistent memory," *arXiv preprint arXiv:1904.07162*, 2019.

[4] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what {COST}?," in *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*, 2015.

[5] M. Jung, E. H. Wilson III, W. Choi, J. Shalf, H. M. Aktulga, C. Yang, E. Saule, U. V. Catalyurek, and M. Kandemir, "Exploring the future of out-of-core computing with compute-local non-volatile memory," *Scientific Programming*, vol. 22, no. 2, pp. 125–139, 2014.

[6] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a {PC}," in *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pp. 31–46, 2012.

[7] "Intel Optane persistent memory product brief." https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html, 2019.

[8] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[9] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection." http://snap.stanford.edu/data, June 2014.

[10] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[11] O. Küçüktunç, E. Saule, K. Kaya, and Ü. V. Çatalyürek, "Towards a personalized, scalable, and exploratory academic recommendation service," in *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, Aug. 2013. acceptance rate: 28%.

[12] G. Iván and V. Grolmusz, "When the web meets the cell: using personalized pagerank for analyzing protein interaction networks," *Bioinformatics*, vol. 27, no. 3, pp. 405–407, 2011.

[13] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh, "Wtf: The who to follow service at twitter," in *Proceedings of the 22nd international conference on World Wide Web*, pp. 505–514, 2013.

[14] O. Küçüktunç, K. Kaya, E. Saule, and Ü. V. Çatalyürek, "Fast recommendation on bibliographic networks with sparse-matrix ordering and partitioning," *Social Network Analysis and Mining*, vol. 3, no. 4, pp. 1097–1111, 2013.

[15] G. Erlebacher, E. Saule, N. Flyer, and E. Bollig, "Acceleration of derivative calculations with application to radial basis function: Finite-differences on the intel mic architecture," in *Proceedings of the 28th ACM international conference on Supercomputing*, pp. 263–272, 2014.

[16] A. E. Sarıyüce, E. Saule, K. Kaya, and Ü. V. Çatalyürek, "Regularizing graph centrality computations," *Journal of Parallel and Distributed Computing*, vol. 76, pp. 106–119, 2015.

[17] C. Dixon, "Temporal resolution using a breadth-first search algorithm," *Annals of Mathematics and Artificial Intelligence*, vol. 22, no. 1, pp. 87–115, 1998.

[18] E. Žunić, A. Djedović, and B. Žunić, "Software solution for optimal planning of sales persons work based on depth-first search and breadth-first search algorithms," in *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 1248–1253, IEEE, 2016.

[19] J. Silvela and J. Portillo, "Breadth-first search and its application to image processing problems," *IEEE Transactions on Image Processing*, vol. 10, no. 8, pp. 1194–1199, 2001.

[20] S. Jeyalatha and B. Vijayakumar, "Design and implementation of a web structure mining algorithm using breadth first search strategy for academic search application," in *2011 International Conference for Internet Technology and Secured Transactions*, pp. 648–654, IEEE, 2011.

[21] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–10, IEEE, 2012.