CP4101 Progress Report

Koh Yi Da A0187674L

AY21/22 Sem 1

1 Project Description

Sorting is an essential part of computing, and especially so in database systems and query processing. In recent years, the emergence of Non-volatile memory (NVM) hardware has transformed the landscape of database systems. NVM technology offers read and write access speeds comparable to traditional DRAM[7], while at the same time being non-volatile like Disk or SSD storage.

However, some key drawbacks of NVM hardware included limited write-endurance[5], as well as read/write bandwidth asymmetry[10]. In my project, I explore and seek to come up with a novel sorting algorithm that takes advantage of DRAM-NVM hybrid architectures. Experiments and evaluations are carried out on the commercially available Intel Optane DCPMM module. In the design and analysis of this novel sorting scheme, we seek to achieve two goals:

1: Minimal (ie. $O(n)$) write complexity in NVM to reduce wearing
2: Practical time efficiency and scalability

2 Literature Review

2.1 Sorting in Database Systems with NVM

Database systems in practical scenarios typically deal with enormous datasets, which can often be too large to fit into the DRAM. Sorting and many other query processing algorithms in these scenarios traditionally require the dataset to be split into chunks and brought independently in and out of the DRAM for processing, such as in *ExternalSort*[8]. Such movement of data chunks can iccur significant IO costs.

With the emergence of directly byte-addressable[1] NVM like the Intel Optane DCPMM, such processing can be greatly simplified and improved upon. The Optane is available in capacities of up to 512GB and is significantly cheaper than conventional DRAM of similar capacities, making it highly ideal for database

servers and storage systems. Furthermore, with the help of Intel's PMDK library, we are able to have direct access (DAX) to the NVM storage media at the application level, bypassing the kernel and file system[9].

2.2 NVM Aware Sorting - B*sort

There has been research into write-limited algorithms specifically targeted for NVM hardware. One such algorithm, named B*-Sort has been proposed by Liang et al[2]. This scheme is inspired by binary search trees (BSTs). Liang et al. describes the B*star sort as a "write-once" algorithm, as each element is guaranteed to be written exactly and only once on sorting a set of elements[2]. This algorithm exploits the nature of non-self-balancing BSTs. Given that there are no removals, every insertion into a non-balanced BST will only need to be written exactly once. In short, this algorithm uses a BST to maintain the sorted order of elements, and simply uses an in-order traversal after all insertions to return the sorted elements.

However, depending on the order of insertion, an unbalanced BST can have worst case $O(n)$ read complexity during insertion in the case of a heavily skewed tree. To circumvent this issue, B*star sort uses additional registers that create "tunnel entries" between nodes to speed up the traversal of the tree, without the use of any self-balancing mechanism. These "tunnel entries" are very lightweight, and stored in the DRAM as opposed to the NVM. As such, the $O(n)$ write once property is preserved. In the worst case analysis, B*sort is able to guarantee $O(n)$ writes and $O(n\sqrt{n})$ reads. In the average case analysis, B*sort can achieve $O(n \log n)$ read complexity [2].

This work was evaluated on a DRAM-simulated platform without actual NVM hardware. While this work showed promising results in a simulated environment, further works have shown that this algorithm performs far worse in reality on the actual Intel Optane NVM hardware[1].

2.3 Parallel Sorting - Samplesort

An area that is much more well studied is parallel sorting algorithms. Among the most commonly used sorting algorithms for multiprocessor or distributed systems is Samplesort, introduced by Frazer and Mckellar[4]. Samplesort is a generalization of Quicksort. Samplesort retrieves a sample from the input data, sorts the samples, and then uses these samples to split (as in Quicksort partitioning) the input data accordingly into multiple partitions. Each of these partitions can then be sorted (using Quicksort) in parallel by each processor. The concatenation of all these sorted partitions then becomes the final sorted output.

The purpose of evaluating parallel sorting algorithms is in consideration of the fact that most modern computing architectures are multicore or multiprocessor systems[6]. It thus makes sense to try to explore parallel sorting algorithms

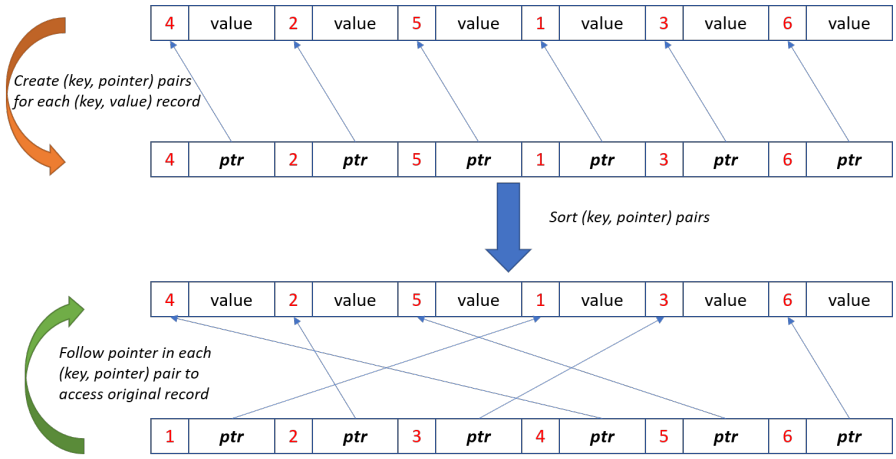

Figure 1: Using the pointer-indirection approach, we create $(key, pointer)$ pairs for each $(key, value)$ record, and sort these pairs instead. This way, we can increase performance by avoiding the expensive copying and writing of entire records, which can be very large.

to fully maximise the throughput and performance of these systems. The divide-and-conquer approach to Samplesort is similarly adopted in my novel sorting approach as well.

2.4 Pointer Indirect Sort

Unlike traditional database storage systems, records stored into NVM like the Intel Optane DCPMM are directly byte-addressable[1]. This feature is highly useful when trying to scale sorting methods according to varying record size. Y. Hua et. al. noted the significant benefit in sorting $(key, pointer)$ pairs instead of $(key, value)$ records when using NVM storage media. Instead of sorting and swapping out records directly in NVM, Y. Hua et. al. proposed building a mapping of $(key, pointer)$ pairs where each $pointer$ is an indirection (or address) to the original $(key, value)$ record[1].

This approach allows us to limit the size of each element being sorted, independent of the size of the $value$ of each record. This scheme not only improves the performance of the sorting, but also greatly reduces read and write volume to NVM during the sorting process. This idea is illustrated in Figure 1.

2.5 Intel Optane DCPMM Read/Write Performance

Many of the existing novel algorithms for sorting in NVM in previous works have largely been evaluated and benchmarked using simulations of NVM hardware, based on assumptions made about the hardware. Some of these assumptions do not hold on the Intel Optane DCPMM hardware that this project is implemented on.

One important observation from a study by Yang et. al. has shown that reads on NVM can be up to 3 times *slower* than writes[7]. This is directly in contrast with assumptions made in other works, that writes to NVM are much slower than reads. In fact, it was also shown empirically that the Optane's write latency is comparable to DRAM, but its read latency can be up to 3x worse[7]. Furthermore, the maximum write bandwith of the Optane DCPMM is about 2.83x lesser than its maximum read bandwidth[10].

3 Current Progress

3.1 SplitSort - Write-Limited Parallel Sorting

Algorithm 1 SplitSort on N NVM-resident records, using M threads

1: Sample K elements from N records ($K << N$) (**in parallel**) (DRAM)
2: Sequentially sort K elements (DRAM)
3: Split K sorted elements into P equal Partitions (**in parallel**) (DRAM)
4: **for** $i = 1, 2, \ldots, P$ **do** (**in parallel**)
5: Find the middle element X in Partition i (DRAM)
6: Open a new NVM file region to store the BST in this partition R (NVM)
7: Create a new BST in region R using element X as the root node (NVM)
8: Create a *mutex* associated with this region/partition (DRAM)
9: **for** $j = 1, 2, \ldots, M$ **do** (**in parallel**)
10: Scan the j^{th} $\frac{N}{M}$ elements from the N records (DRAM)
11: **for** each of the $\frac{N}{M}$ elements, E **do**
12: Binary search for the partition P_z which E's key belongs in (DRAM)
13: Acquire *mutex* for partition P_z, block if necessary (DRAM)
14: Insert element E into the BST at Partition P_z (NVM)
15: Release *mutex* for partition P_z
16: Initialize an array of P arrays of records, *Result* (NVM)
17: **for** $k = 1, 2, \ldots, P$ **do** (**in parallel**)
18: Perform an inorder traversal of the BST at Partition P_k, write each record S into the array $Result[k]$ (NVM)
19: Return *Result*

Building upon B*sort's idea of using BSTs, I implemented and devised, under Prof Tan Kian Lee's supervision, a new sorting algorithm that draws upon ideas from Samplesort and B*sort. *Splitsort* (tentative name) is a parallel, $O(n)$-guaranteed write-complexity algorithm that takes advantage of the byte-addressability of the Intel Optane DCPMM. The high level pseudo code is shown above in **Algorithm 1**: SplitSort.

A high-level walk-through of the algorithm using $N = 24$, $K = 12$ and $P = 4$ is illustrated in Figures 2, 3 and 4 below.

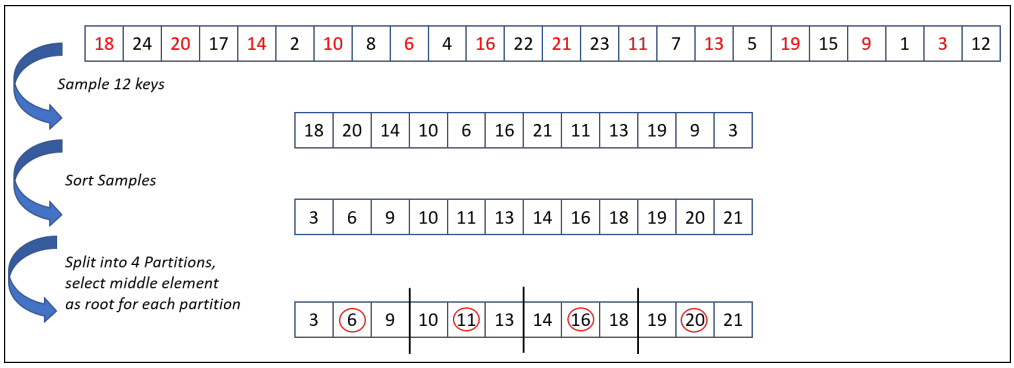

Figure 2: Initial phase of a sample run of *SplitSort* with 24 elements. $K = 12$ elements are sampled systematically and sorted in DRAM. Then, the sorted samples are split into $P = 4$ partitions. The middle element of each partition is then selected as a root, and a BST in NVM is created for each partition using their respective roots. Note that only the sorting of the sample is sequential. The partitioning and creation of the BST roots can be done in parallel using M threads.

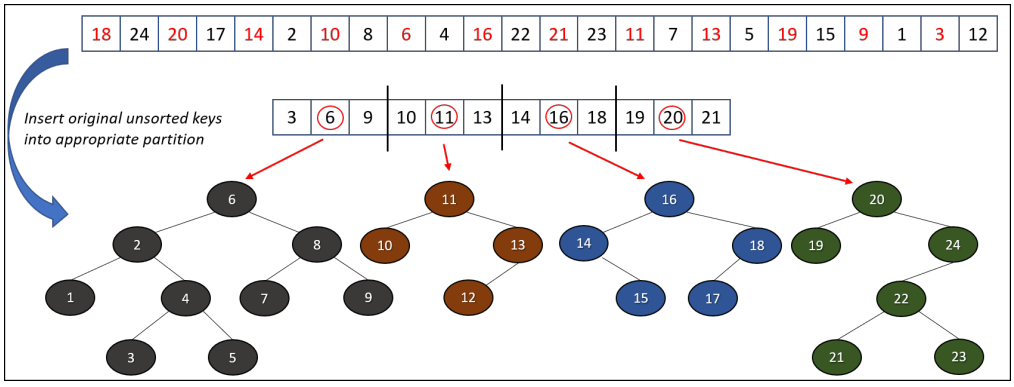

Figure 3: Second phase of a sample run of *SplitSort* with 24 elements. After the BSTs are created at each partition, each item from the original unsorted input is inserted (in parallel) into its appropriate partition's BST. Each thread is then given an exclusive working set of unsorted items to insert. Each partition is protected by a *mutex* should there be any concurrent access. Note that each BST is in NVM, and all BST node insertions are into NVM.

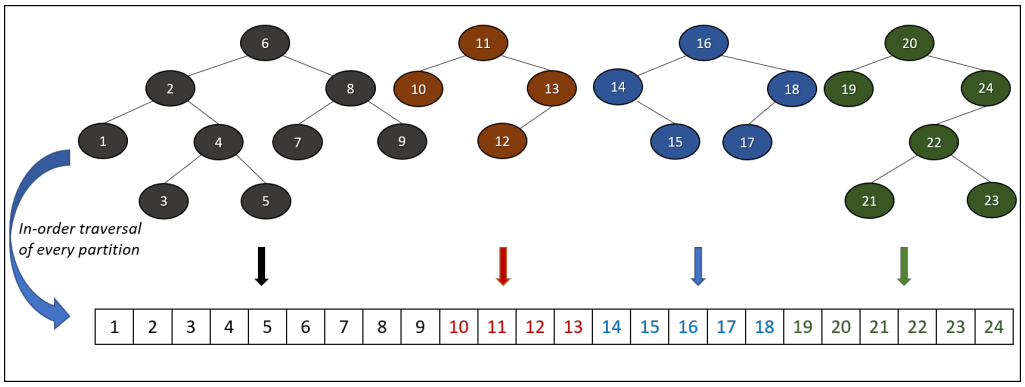

Figure 4: Final phase of a sample run of *SplitSort* with 24 elements. After all elements are inserted, we perform an in-order traversal of every BST at each partition, using M threads in parallel. The concatenation (from left to right) of the traversals of each BST in each partition is then the final sorted data.

3.2 SplitSort - Big O Analysis

Given that there is sampling performed on the original records, the performance of *SplitSort* is sensitive to the underlying distribution of the data. For the purposes of my analysis and benchmarking up till this point, **I assume a uniform distribution over the input unsorted records**.

3.2.1 Sampling and Partitioning

Given M threads and K samples, where $K >> M$, using evenly-space samples we can obtain the samples in $O(K/M)$ time.

We then employ a sequential sort on K samples in DRAM, which entails $O(K \log K)$ time.

Splitting the K sorted elements into P evenly sized partitions using M threads, assuming $P >> M$, takes $O(\frac{P}{M})$ time as we simply need to obtain the lower bound for the elements in each partition, and the middle element of each partition.

3.2.2 Searching and Inserting into appropriate Partition

We set aside any contention due to locking of each partition in this part. The binary search for the correct partition will first take $O(\log P)$ time on each thread. The expected size of each of the P partitions is $O(\frac{N}{P})$. Furthermore, it can be shown that the average height of a BST generated by a independent random insertions is $O(\log a)$[11].

Finally, given that each of the M threads has roughly $\frac{N}{M}$ elements to insert, the total complexity on each thread in the insertion phase is thus $O(\log P + \frac{N}{M} \log(\frac{N}{P}))$ on average, and $O(\log P + \frac{N}{M}(\frac{N}{P})^2)$ in the worst case.

3.2.3 Inorder Traversal

Again, the complexity of the in-order traversal depends on the distribution of input data and thus the largest size of any partition. The time taken for the in-order traversal is bottle-necked by the largest partition.

If we assume a uniform distribution, and that each partition has roughly the same number of sorted records, then each partition will have on average $O(\frac{N}{P})$ elements. Assuming a work distribution where each thread traverses $O(\frac{P}{M})$ partitions sequentially, the time complexity for the parallel inorder traversal is on average $O(\frac{N}{M})$.

3.2.4 Write Complexity

All metadata for each partition, such as its lower (or upper) bounds are assumed to be relatively small, and thus stored completely in **DRAM**. All of the K samples obtained during the sampling phase are also assumed to (or can be configured to) fit completely into **DRAM**. Thus, each original record to be sorted is written exactly once (as a $(key, pointer)$ pair) into it's respective BST in NVM.

As such, **we are also able to achieve the "write-once" guarantee as described in B*sort, which translates asymptotically to $O(n)$ writes to NVM.**

3.3 SplitSort - Implementation details

3.3.1 Sampling Strategy

For the initial implementation, we use a simple systematic sampling strategy, and select every $\frac{N}{K}$ records (evenly spaced out).

3.3.2 Pointer-Indirection

Similar to what was suggested by Y. Hua et al, I used a pointer-indirect approach to sorting all the records. Instead of sorting on the records directly in NVM, we create $(key, pointer)$ pairs for each record, and sort these pairs instead. At the end of the algorithm, we perform a dummy access of each record by de-referencing the pointer to the actual record. This is to account for the additional reading overhead introduced by the pointer-indirection.

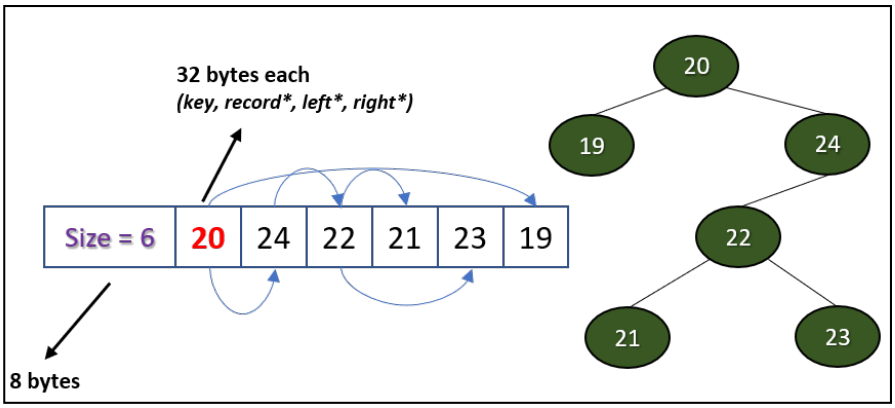

Figure 5: Layout of each partition's BST in contiguous memory. The start of the memory region is a sentinel 8-byte integer denoting the number of nodes. The following region are the actual BST nodes. The order of nodes in the region is exactly the insertion order (the root, highlighted in red is always inserted first).

3.3.3 Concurrency Control

As can be seen in the algorithm, each partition, which contains a BST is protected by a *mutex*. The work distribution for each thread is split across the input unsorted data. As such, the partitions that each thread needs to access to insert each element is not known at compile time. In the general case, every partition could be concurrently accessed by multiple threads.

The level of contention heavily depends on the distribution of the input data and the sampling strategy, as the partitioning is a direct result of those two factors. In highly skewed distributions, it could likely be the case that many records are placed into the same partition, leading to high contention between threads, and CPU time wasted on waiting for the *mutex* ownership.

3.3.4 Partition Memory Management

To manage each partition, we need to keep track of its *metadata*, as well as the actual BST of records stored into this partition. Partition *metadata* to keep track of include the *lowerbound* of the keys that it should store, a pointer to the *rootnode* of the BST stored within this partition and a *mutex* to protect concurrent access to this partition. These metadata are stored in the **DRAM** as they are relatively lightweight, and remain constant in total size.

In contrast, the BST for each partition is stored in **NVM**. The entire BST is stored as a contiguous block of memory inside the NVM. The start of the memory region contains a single 8 byte integer, which indicates the current number of nodes in the BST. The root of the BST is then stored directly after

this bookkeeping integer. Each new incoming node in the BST is then placed directly behind the previously inserted node in the contiguous memory region. This layout of a BST is illustrated in Figure 5.

3.3.5 Dynamic Partition Resizing

As addressed earlier, the eventual size of each partition and the size of its BST is not known at compile time, and depends on the input distribution and sampling strategy. In the current implementaton, the algorithm is not adapative to the input distribution. Some partitions are likely to ultimately contain more nodes than other, and could, theoretically, in the worst case, contain **all** the nodes.

This presents a problem when initializing the NVM memory region for each partition during the initialization phase. Initializing each partition to be the worst case size is not practical as it severely impacts performance, and does not scale well. Instead, I initialize the size of each partition to be some **constant factor** of the **expected partition size**, assuming a uniform distribution across the input data. The equation below illustrates this idea:

Unit Size of Each Partition $= c(\frac{N}{P})$ BST nodes, for some constant c.

In the event where the initial memory region for a partition is completely full, a thread attempting to insert a new node will dynamically create a new memory region (of the same size as the initial region) for this partition in NVM, and associate this new contiguous region with the same partition. Any new incoming nodes are subsequently inserted into the newly created region instead. This dynamic provisioning of new NVM regions can be done repeatedly whenever memory in a partition runs out. This method is preferred over re-allocating a larger NVM region each time, and copying over the contents of the old region as that could possibly lead to more than $O(n)$ writes to NVM.

3.4 Experiments and Discussion

3.4.1 Hardware and Software specifications

SplitSort is implemented using C++17, GCC version 9.3.0. The implementation also uses the OpenMP library for parallelization and the Intel PMDK Persistent Memory Development Kit library for direct access (DAX) to the NVM storage, bypassing the kernel space. The Optane DC PMM is configured to $App - Direct$ mode, and is directly exposed to the application.

For performance evaulation, the code was run and compiled (without any optimization flags) on the optane.d2.comp.nus.edu.sg machine, equipped with an Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz, with 32 cores and up to 64 hardware threads, operating on Ubuntu 20.04.2 LTS.

Test 1		Time/seconds			
		SplitSort ***(NVM)***	*STL::sort* ***(NVM)***	*SplitSort* ***(DRAM)***	*STL::sort* ***(DRAM)***
Number of Threads	1	**75.805**	**22.342**	**27.326**	**8.856**
	4	**23.274**	-	**10.964**	-
	8	**13.516**	-	**5.808**	-
	16	**8.835**	-	**3.402**	-
	32	**7.340**	-	**2.488**	-
	64	**8.103**	-	**2.216**	-

Figure 6: Results for *Test 1*, using 2^{24} records, each of 256 bytes size. *SplitSort* was configured to take 2^{14} samples, and to divide these samples into 2048 partitions. The total size of the input is 4 GB.

Test 2		Time/seconds			
		SplitSort ***(NVM)***	*STL::sort* ***(NVM)***	*SplitSort* ***(DRAM)***	*STL::sort* ***(DRAM)***
Number of Threads	1	**1944.080**	**415.374**	**796.702**	**170.87**
	4	**605.418**	-	**277.814**	-
	8	**341.436**	-	**152.038**	-
	16	**207.542**	-	**82.665**	-
	32	**150.479**	-	**52.548**	-
	64	**149.184**	-	**38.397**	-

Figure 7: Results for *Test 2*, using 2^{28} records, each of 256 bytes size. *SplitSort* was configured to take 2^{18} samples, and to divide these samples into 4096 partitions. The total size of the input is 64 GB.

3.4.2 Experimental Setup and Assumptions

In this initial experiment, we compare the time-efficiency between C++17/GCC 9.3.0's Standard Library (STL) **sequential** *sort* algorithm (which is a variation of Musser's *Introsort*[3]) against our implementation of *SplitSort* with varying number of threads. In all tests, we use the above described pointer-indirect approach, whereby $(key, pointer)$ pairs are sorted instead of the actual $(key, value)$ records. After sorting, we then traverse all $pointers$ in sorted order to account for the additional pointer indirection overhead.

We ran two sets of tests against both algorithms, with a varying number of threads used in the *SplitSort* approach, and a single thread in the C++17 STL *sort* approach. Within each set of tests, we use the same random seed to generate a **uniformly distributed** set of $(key, value)$ records in the NVM memory region. For each test, we also compare between storing all $(key, pointer)$ pairs (and BST nodes for *SplitSort*) in **in-NVM vs. in-DRAM**. We also assumed that the total size of K ($K << N$) sample records fit entirely into DRAM.

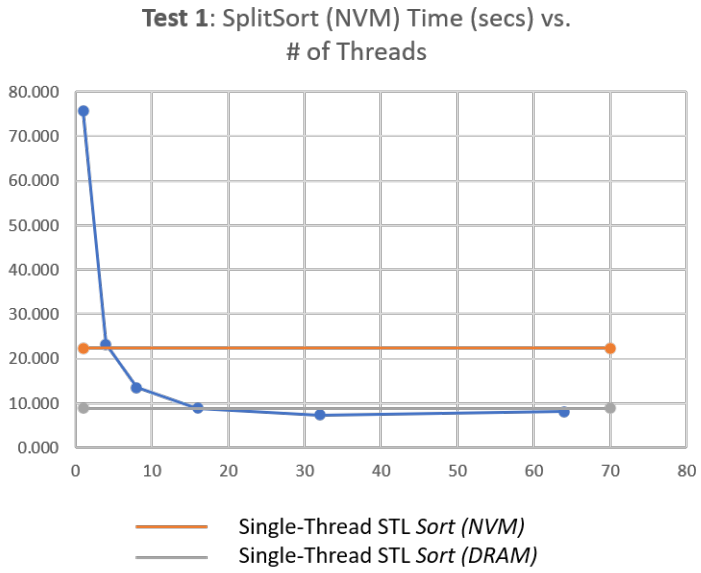

Figure 8: *Test 1* results: Graph of *SplitSort* time taken (in NVM) vs. Number of Threads used. The orange and grey line represent single-threaded STL *sort*'s time taken in NVM and DRAM respectively.

3.4.3 Experimental Results

Test 1 was performed using an input of $\mathbf{N} = 2^{24}$ records of 256 bytes each. The total input size is $4GB$. The results can be seen in Figure 6 and 8. The parameters and memory usage for *SplitSort* in this test are below:
P (No. of Partitions): 2048
K (No. of Samples): 2^{14}
Size of Key in Record: 8 bytes
Unit Size of each Partition: $1.25 * \frac{N}{P}$ nodes
Total Partition Size (Initial): 640.016 MB
Total Partition Size (Final): 776.581 MB

Test 2 was performed using an input of $\mathbf{N} = 2^{28}$ records of 256 bytes each. The total input size is $64GB$. The results can be seen in Figure 7 and 9. The parameters and memory usage for *SplitSort* in this test are below:
P (No. of Partitions): 4096
K (No. of Samples): 2^{18}
Size of Key in Record: 8 bytes
Unit Size of each Partition: $1.25 * \frac{N}{P}$ nodes
Total Partition Size (Initial): 10240 MB
Total Partition Size (Final): 10557.5 MB

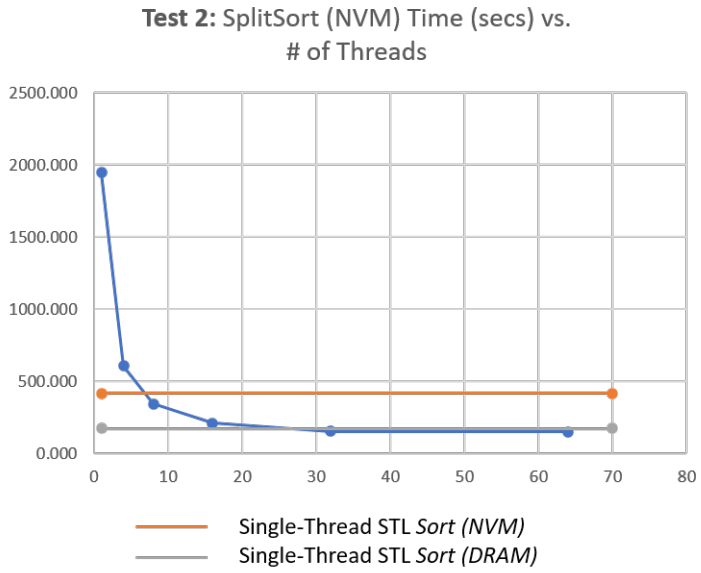

Figure 9: *Test 2* results: Graph of *SplitSort* time taken (in NVM) vs. Number of Threads used. The orange and grey line represent single-threaded STL *sort*'s time taken in NVM and DRAM respectively.

3.4.4 Discussion

We observe that the single threaded performance of our *SplitSort* algorithm is observably slower than the STL *sort* algorithm. *SplitSort* in **NVM** is about 3.39x slower than the STL *sort* in NVM in *Test 1*, and about 4.68x slower in *Test 2*. In **DRAM**, single threaded *SplitSort* is about 3.09x slower in *Test 1*, and 4.66x slower in *Test 2*. Such performance of *SplitSort* in a single-threaded context is expected, as the BST structure in each partition is not cache-friendly, as compared to the in-place comparisons and swapping in the STL *sort* algorithm.

However, we note that **as we increase the number of threads, the performance of** *SplitSort* **increases drastically as well**. In *Test 1*, we observe that *SplitSort*'s performance in **NVM** with just 4 threads is already comparable to the STL *sort* in **NVM**. At 8 threads, *SplitSort* already outperforms the sequential STL *sort* by about 1.65x, in **NVM**. At 32 threads, *SplitSort* in **NVM** even outperforms STL *sort* in **DRAM** by about 1.21x.

In *Test 2*, we observe that *SplitSort*'s performance in **NVM** with 8 threads outperforms the STL *sort* in **NVM** by about 1.21x. At 32 threads, *SplitSort* in **NVM** similarly outperforms STL *sort* in **DRAM** by about 1.14x. As such, **given enough threads, we are able to scale the performance of** *SplitSort* **in NVM to be as good as sequential STL** *sort* **in NVM or DRAM.**

4 Future Work

4.1 DRAM-NVM Hybrid Partition Storage

Given that the Intel Optane is unlikely to be used as a complete replacement for DRAM, it is thus worth to explore optimizations that take advantage of such a hybrid memory architecture, given the fact that reads and write to DRAM are still ultimately than to NVM. Some further optimizations to the storage in each partition include:

1: Storing a proportion of the elements in each partition in DRAM, instead of all in NVM.

2: Storing the earlier depths of the BST in DRAM, or in a small cache-friendly NVM region, while storing the deeper depths of the BST in a larger NVM region.

4.2 Adaptive Sorting

Ultimately, Quicksort is still a faster single-threaded algorithm compared to sorting via BSTs, due to cache locality. An idea to be considered is to:

1: If an entire partition's elements can fit into DRAM (or a factor of the DRAM size), we can simply perform in-DRAM Quicksort on the elements of this partition, as opposed to building a BST.

4.3 Distribution-Aware Sampling/Partitioning

The performance of the algorithm can be very sensitive to the input distribution. If we are able to extract insights into the distribution of the input data, perhaps we can further optimize the partitioning strategy to prevent some partitions from becoming significantly larger than others. An idea to be considered is:

1: Given prior knowledge (or acquired knowledge) about the input distribution, we can perhaps vary our sampling strategy to create better partitions that are more equally populated.

4.4 Performance Tuning

Increasing the number of partitions does not necessarily result in better performance, given the same number of threads. More investigation needs to be done to find the best configuration of settings, under different input sizes, distributions (eg. Zipfian, Normal etc) and constraints (limited DRAM size) for maximum performance. Important factors to investigate include:

1: Increasing Partition Count - Smaller BSTs on average per partition, less contention between threads, larger file creation overhead.

2: Increasing Sample Size - Better representation of input distribution, more sampling overhead.

3: Increasing Initial Partition Size - Fewer additional NVM memory allocation, larger file creation overhead and memory overhead.

4: Increasing Thread Count - Higher degree of parallelism, potentially more synchronization overhead, larger impact on other applications.

4.5 Benchmarking

The benchmarking is currently done for time-efficiency only, and against the GCC C++ Standard Library's sequential sort. More experimentation needs to be done in the area of:

1: Benchmarking against other parallel sorts, C++17 STL parallel sorts.

2: Benchmarking against other NVM-aware algorithms.

3: Benchmarking against traditional sorting algorithms (eg. Quicksort, MergeSort etc)

4: Benchmarking NVM write frequency against other NVM-aware and traditional sorting algorithms.

4.6 Integration

Given that *SplitSort* can be tuned to use varying levels of parallelism, it might be worth investigating it's performance within a larger query processing pipeline or system under constraints (eg. only allowed up to use 8 cores for sorting). Furthermore, the pointer-indirect approach and in-order traversal also introduces additional reading overhead for the end-user, which might impact performance in the query processing pipeline.

References

[1] Y. Hua et al. (2021) Redesigning the Sorting Engine for Persistent Memory. Database Systems for Advanced Applications

[2] Liang et al. (2020) B*-Sort: Enabling Write-Once Sorting for Nonvolatile Memory. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 39

[3] Musser, David R. (1997) Introspective Sorting and Selection Algorithms. Software: Practice and Experience. 27

[4] Frazer, W. D.; McKellar, A. C. (1970) Samplesort: A Sampling Approach to Minimal Storage Tree Sorting". Journal of the ACM. 17 (3): 496–507

[5] Qureshi, M.K., et al. (2009) Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In: 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)

[6] Schauer, Bryan. (2008) "Multicore Processors: A Necessity"

[7] Yang, J., Kim, J., Hoseinzadeh, M., et al. (2020) An empirical guide to the behavior and use of scalable persistent memory. In: Proceedings of the 18th USENIX Conference on File and Storage Technologies, pp. 168–182

[8] Khorasani, E., Paulovicks, B.D., Sheinin, V., Yeo, H. (2011) Parallel implementation of external sort and join operations on a multi-core network-optimized system on a chip. In: Xiang, Y., Cuzzocrea, A., Hobbs, M., Zhou, W. (eds.) ICA3PP 2011. LNCS, vol. 7016, pp. 318–325. Springer, Heidelberg

[9] Intel Corporation. Persistent Memory Development Kit. (Retrieved Nov 2021) Available at https://docs.pmem.io/persistent-memory/getting-started-guide/what-is-pmdk.

[10] Izraelevitz, J et al (2019) Basic Performance Measurements of the Intel Optane DC Persistent Memory Module, pp. 20-21

[11] Flajolet, P., Odlyzko, A. (1982) The Average Height of Binary Trees and Other Simple Trees. In Journal of Computer and System Sciences 25, pp. 171-213