Parallel Oblivious Sorting in SGX Enclave

Tianyao Gu, Tian Xie

December 2023

Abstract

Oblivious algorithms have been a popular research topic in cybersecurity as it guarantees a simulatable memory access pattern and thereby provably resists side channel attacks. We implemented a parallel oblivious sorting algorithm in C++ using Intel SGX Enclave, a hardware-based trusted-computing environment offering privacy and authenticity. We leveraged OpenMP to achieve multithreading on a 36-core CPU, and further improves parallelism through SIMD. We increased the arithmetic density of the algorithm to make it efficient in an external memory model. We also parallelized data swapping across the enclave boundary through prefetching and write back buffers. We reduced memory fragmentation of the program to minimize the consumption of Enclave Page Cache. Our implementation of the oblivious sorting algorithm achieves a speedup of tofillx over the original serial implementation with 8 threads, and tofillx with 32 threads. As a byproduct, we also obtained a parallel oblivious random shuffling algorithm, which features tofillx speedup with 8 threads, and tofillx speedup with 32 threads.

1 Background

1.1 Background of Intel SGX

Intel SGX is a hardware-based trusted-computing environment that offers privacy and authenticity. It provides a secure enclave, known as the Enclave Page Size (EPC), which is a protected region of memory. The enclave is encrypted and authenticated by the CPU, isolated from the rest of the system, including the operating system and other applications. The CPU ensures that the enclave is not tampered with and that the enclave is not swapped out to disk.

Intel SGX v1 supports up to 128 MB of enclave memory. Although Intel SGX v2 could in principle support up to 1 TB of enclave memory, the EPC size is still very limited on most consumer-grade CPUs, and even for high-end CPUs, it often takes too long to initialize a large enclave for many applications. Swapping data in or out the enclave requires running heavy-weight cryptographic primitives such as encryption, decryption, and authentication. Moreover, these cryptographic primitives have a startup cost, which means data should be transferred at the granularity of pages (at least 4 kB). This is similar to the disk I/O model in the conventional operating systems (hence we will borrow the term I/O in the rest of the report to refer to such data swaps). As a result, the algorithm must be carefully designed to minimize the amount of data that needs to be transferred across the enclave boundary. In other words, algorithms should have high arithmetic density and good spatial locality.

While SGX allows parallel execution of threads within the enclave, it also poses some limitations. First, SGX thread is mapped directly to logical processor (to avoid the control of OS). Therefore, we cannot create more threads than the available cores, which decreases the flexibility of thread scheduling. Second, while OpenMP has recently become supported in Intel SGX, the functionality is limited, and there are few documentation and examples. For instance, we are not able to turn on dynamic scheduling in `#pragma omp parallel for` loops. Finally, SGX has a non-negligible impact on the performance of the program, and especially on the total memory bandwidth, as is shown in [6, 7]. Consequently, the performance of algorithms are likely to be memory-bound at high thread counts.

1.2 Background of Oblivious Sorting Algorithms

Our motivation is to securely outsource data and computation to an untrusted worker equipped with secure processors such as Intel SGX. Although secure processors ensure data privacy through encryption, existing literature reveals that attackers can exploit memory access and page swap patterns during computations to glean information about the data.

Oblivious algorithms provide a verifiable shield to counter such side-channel attacks. This is because "Obliviousness" essentially demands that memory access and page swap patterns remain independent of secret data.

In addition, sorting stands out as a fundamental building block of various oblivious computation applications. Specifically, oblivious sorting is key to common graph algorithms [3, 4, 10] including breadth-first search [3, 4], connected components [11], minimum spanning tree/forest [11], clustering [10], list ranking [11], tree computations with Euler tour [11], and tree contraction [11]. Oblivious sorting can also be used for securely initializing common ORAM algorithms [14] including Path ORAM [12], which has been deployed at a large scale in practice [13]. Moreover, any computational task that can be efficiently expressed as a streaming-Map-Reduce algorithm has an efficient oblivious implementation using oblivious sort [4, 10].

Our project is based on a prior research paper by Gu et al. [5]. The paper proposed and implemented an efficient oblivious sorting in hardware enclaves, which achieves asymptotically optimal runtime and outperforms all the previous works concretely. However, the algorithm, as presented and implemented, is single-threaded, posing limitations on its performance.

In our continuation of this work, we aim to enhance performance by parallelizing the oblivious sorting algorithm. As a byproduct, we obtain a parallel oblivious random shuffling algorithm, which is also a crucial primitive in oblivious computation.

1.3 Background of Flex-way Butterfly Oblivious Sort

To achieve oblivious sorting, Asharov et al. [1] proposed the idea to first design an *oblivious random bucket assignment* algorithm, which randomly assigns each input element to one of $O(N/Z)$ output buckets, each of poly-logarithmic capacity Z; further, the obliviousness property requires that the access patterns do not leak the destination bucket of each element. Then, from the oblivious random bucket assignment, they get an *oblivious shuffling* algorithm, which randomly permutes the inputs without revealing the permutation. Finally, they can apply any non-oblivious, comparison-based sorting algorithm to the shuffled array.

Flex-way Butterfly Oblivious Sort [5] adopts this idea and achieved the *oblivious random bucket assignment* using a multi-way butterfly network. The input of the network consists of $(1+\epsilon)N/Z$ buckets of size Z, where each bucket contains $Z/(1+\epsilon)$ real input elements and is padded with filler elements. Each element is attached a random tag.

The network relies on an important building block called multi-way MergeSplit, which merges elements from k buckets and obliviously splits the real elements into k buckets according to the modulus of their tags. Because each bucket is padded with filler elements, the no bucket will overflow except with negligible probability. Figure 1 demonstrates a 3-level butterfly network that assigns elements to 18 buckets.

Concretely, a solver is applied to determine the optimal parameters of the bucket and butterfly network, which minimizes the total runtime while ensuring an overflow probability within 2^{-60}.

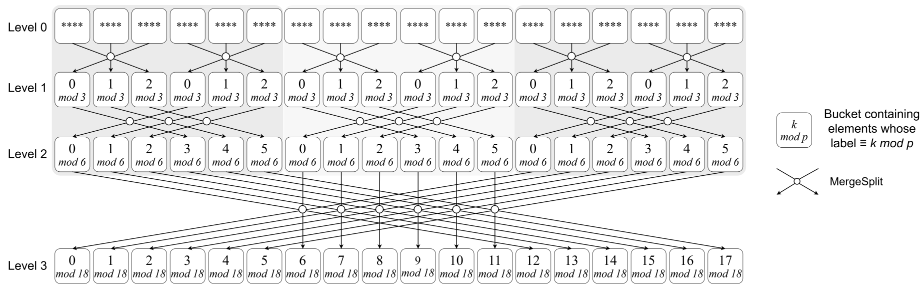

Figure 1: A 3-level multi-way butterfly network.

As elements are routed to their destination buckets, bitonic sort [2] is executed within each bucket, filter out the filler elements, and remove the tags. At this stage, we have obtained an oblivious random shuffling algorithm. The final step is to run external merge sort [8] on all the real elements and get the final sorted array.

When implementing the algorithm in Intel SGX, we need to consider the memory limitations of the SGX enclave, since transferring data across the boundary of the enclave requires expensive cryptographic operations. Naively emulating the butterfly network level by level would incur an extra I/O overhead of $\Theta(\log N)$. Therefore, we want to route elements through multiple levels each time we fetch them into the enclave, as is demonstrated in Figure 2. This optimization significantly increases the arithmetic density of the algorithm and thereby improves performance. We can also combine multiple stages of the algorithm in each batch to further reduce the I/O cost. For instance, the first level of the external mergesort can be piggybacked with the last level of the butterfly network. Please refer to [5] for more details.

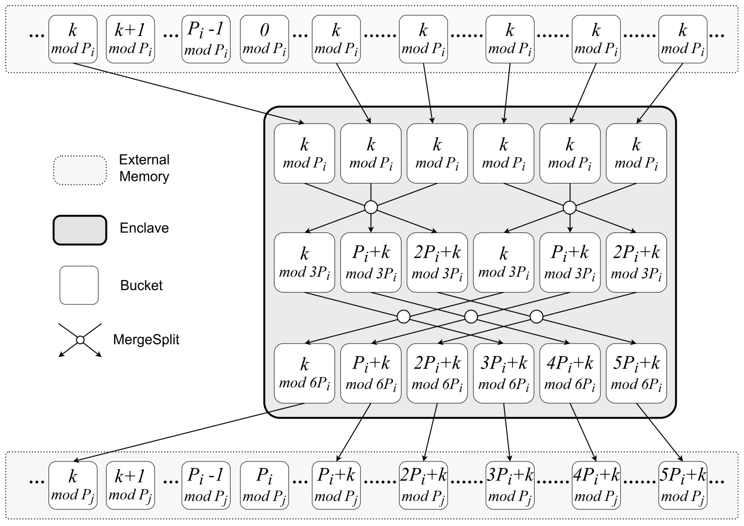

Figure 2: Route elements through multiple levels of butterfly network in each batch.

2 Parallelism Abstraction

2.1 Parallelism in Butterfly Network

Emulating the butterfly network is the most time critical part of the algorithm. We observe that merge-split operations on each butterfly network level are independent of each other, and therefore we can parallelize them. In addition, we can also parallelize allt the bitonic sort instances at the end of the shuffling phase. Notice that we can only perform parallelization within each batch since threads cannot access data outside the enclave directly.

2.2 Parallelism in Multi-way Merge-Split

The multi-way merge-split operation itself is also parallelizable. It involves a recursive divide-and-conquer algorithm, which can be parallelized by OpenMP.

However, we may not achieve a perfect speedup on the first few recursive calls, since the algorithm involves components that are difficult to parallelize (e.g., searching for an Euler tour on a graph). Moreover, for practical implementation, we need to be careful about the parallelism granularity. Setting the granularity too small would incur too much overhead on thread creation and synchronization.

2.3 Parallelism in External Merge Sort

The non-oblivious external merge sort involves two phases. First, we read data into the internal memory (EPC in our case) in batches and sort data in each batch. Second, we merge all the sorted chunks using a priority queue (this step may need to be performed recursively for very large data size and page size).

The first step can be parallelized by running a parallel merge sort in each batch. The priority queue in the second step is more difficult to parallelize. Nevertheless, we can still parallelize the I/O with external memory, as will be elaborated in section 2.4.

2.4 Parallelism in I/O

We can parallelize the I/O operations with external memory. Specifically, we can parallelize the cryptographic operations when swapping data between the enclave and the untrusted memory. For streaming data, we can apply prefetching and read-back buffers. In theory, when the data exceeds the available physical RAM, we may also swap data to disk in parallel and overlap communication with computation. However, this is more difficult to achieve since disk I/O requires system calls, which are not supported in SGX. Consequently, we must first make `OCALL`s to switch to the untrusted host and perform asynchronous I/O operations.

2.5 Parallelism in Element Movement

Our oblivious sorting algorithm assumes that each element consists of a sorting key and a payload. The payload length is not fixed and can be arbitrarily long. We also assumed an indivisible model [9], i.e. we cannot separate the payload from the element or perform any computation on the element (such as running compression algorithms). Thus, the payload must be moved along with the sorting key.

Therefore, for elements with large payload, we may parallelize the data movement via SIMD. One particularly interesting case is the oblivious swap operation, i.e., how to conditionally swap two elements without revealing the condition. It turns out that we can apply SIMD to oblivious swap multiple memory words in parallel, as is elaborated in section 3.5.

2.6 Parallelism in Pseudo-Random Numbers Generation

The algorithm requires a large number of secure pseudo-random numbers in the shuffling phase. We can parallelize the generation of pseudo-random numbers by using multiple pseudo-random number generators.

3 Implementation

3.1 Overview

We implemented the algorithm based on the open-source code provided by the authors of the original paper [5], available at `https://github.com/odslib/oblsort`. The code is written in C++ and uses the Intel SGX SDK. We forked the repository and open-sourced our parallel implementation at `https://github.com/gty929/oblsort`. Below, we describe our modifications to make the program parallel, as well as some major optimizations. While we test the code on a high-end server equipped with 36 cores, 512GB EPC cache and 1TB RAM, our implementation also works for the consumer-grade processors, as well as the virtual machines provided by cloud service providers, which have more limited hardware resources. Figure 3 shows a high-level diagram of our implementation on a toy example.

3.2 Parallelize Butterfly Network

We applied the `#pragma omp parallel for` directive to parallelize the butterfly network in each batch. Since all the merge-split operations on each level have equal workload, we applied static scheduling to divide the workload evenly among threads and avoid the overhead of dynamic scheduling. We also applied the `#pragma omp parallel for` directive to parallelize all the bitonic sort operations in each batch.

One modification we made to the original code is to change the butterfly routing schedule from recursive to iterative. In the serial implementation, the butterfly network is divided into two smaller butterfly networks and a merge pass. While the divide and conquer scheduling slightly increases the cache locality of the algorithm, our experiment shows that OpenMP in SGX cannot fully exploit the parallelism in such a recursive program. Therefore, we instead

To maximize parallelism, we always fetch as many buckets as possible into the enclave in each batch. We also optimized the parameter solver to ensure that there are enough parallel tasks on each butterfly network level (by reducing bucket size and balancing the merge-split ways on each level.)

3.3 Parallelize External Merge Sort

In the serial version, the external merge sort takes up to 1/4 of the total runtime. Therefore, it is important to parallelize the external merge sort as well.

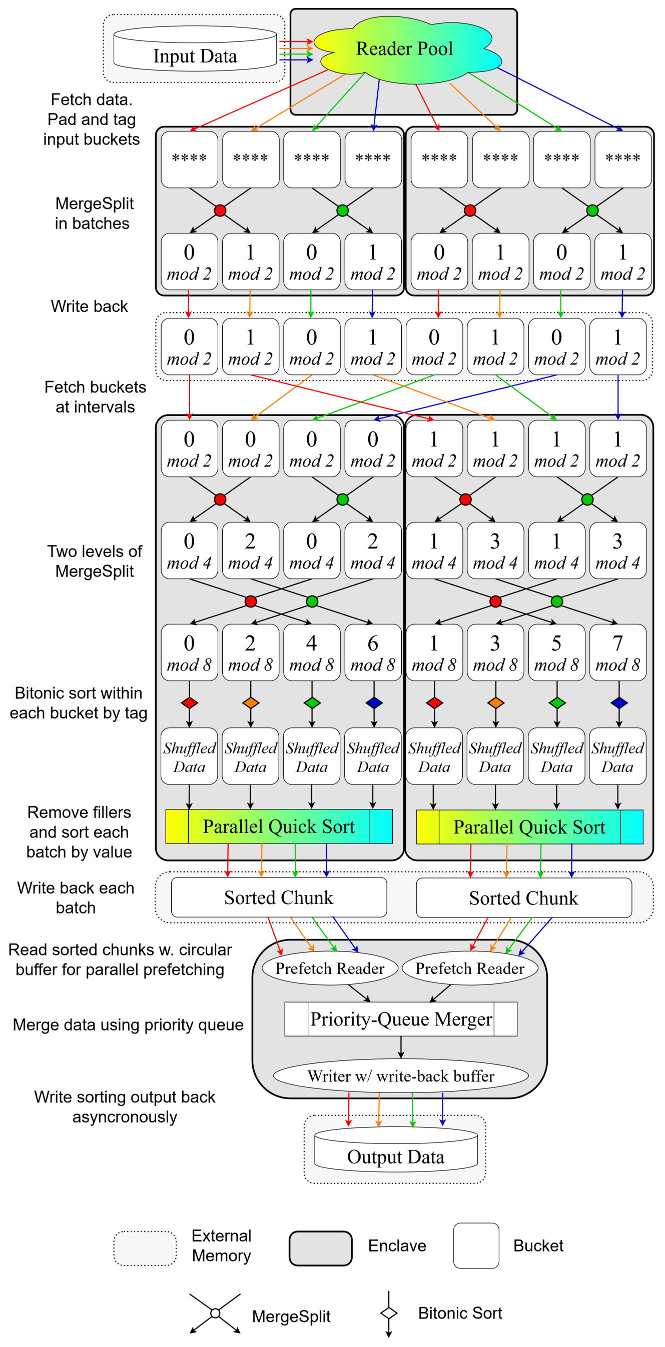
Input Data
Reader Pool
Fetch data. Pad and tag input buckets
MergeSplit in batches
Write back
Fetch buckets at intervals
Two levels of MergeSplit
Bitonic sort within each bucket by tag
Shuffled Data
Remove fillers and sort each batch by value
Parallel Quick Sort
Write back each batch
Sorted Chunk
Read sorted chunks w. circular buffer for parallel prefetching
Prefetch Reader
Merge data using priority queue
Priority-Queue Merger
Write sorting output back asyncronously
Writer w/ write-back buffer
Output Data
External Memory
Enclave
Bucket
MergeSplit
Bitonic Sort

Figure 3: Diagram of Parallel Oblivious Sort on a toy example. Each color represents a thread. Colorful components are multi-threaded.

In the external merge sort, the first step is to read data into the internal memory (EPC in our case) in batches and sort data in each batch. We parallelized this step by running a parallel quick sort in each batch using `#pragma omp task`. While in theory merge sort can achieve better work balancing, we selected quick sort for two main reasons. First, while merge sort incurs slightly fewer comparisons, it causes more element swaps, which is more expensive in our case since we need to swap the payload as well. Second, quick sort is easy to be implemented in place. This is important since we need to sort the data in the enclave and cannot allocate extra memory outside the enclave. While merge sort can also be implemented in place, it has higher constant factors and is more difficult to implement.

The second step of the external merge sort is to merge all the sorted chunks using a priority queue. This step is more difficult to parallelize. One solution would be to break the large tree of priority queue to multiple smaller ones and let each thread manage a subtree. Then, we can use a smaller central priority queue to merge the top elements of all the small heaps. In practice, however, we noticed that for most reasonable sized test cases, the heap size is not very large, and the synchronization overhead of the implementation above would slow down the performance.

Therefore, we only parallelized the I/O with external memory. When fetching the sorted chunks, we create a reader for each sorted chunk, which has a circular buffer of size b to prefetch the data. When the i-th page of the sorted chunk has been fully consumed, the reader spawns an asynchronous task to fetch the $(i + b)$ th page and overwrite the buffer of the consumed page. Since the elements have been randomly shuffled before running the external merge sort, all the readers (except the last one) are expected to be consumed at the same speed. Therefore, even for a small constant b, with high probability the page-fetching will complete when the page is actually needed by the merger. In case, the page is not ready yet, the merger will busy wait until the page is ready. We also applied write-back buffers when writing the merged data to external memory. We will elaborate on the details in the next section.

3.4 Parallelize I/O

We identify three I/O patterns in the algorithm and used different strategies to parallelize them.

- **Aligned** When each thread reads or writes equal chunks of data that is aligned with the encryption blocks. In this case, the threads have no contention on the encryption blocks, and we can directly apply `#pragma omp parallel for` to parallelize the I/O operations.

  An example of this case is the swapping of intermediate data in the butterfly network (i.e., all the I/O except reading the initial input and writing the shuffled output). We deliberately set the encryption block size to be equal to the bucket size, and make them perfectly aligned. Therefore, the buckets can be copied into or out of the enclave in parallel.

- **Unaligned and Unordered** In this case, each thread reads or writes unequal amount of data or the data is not aligned with the encryption blocks, yet we don't care about the order the data being read or written. In this case, we can initiate multiple readers or writers (we let the number match the available threads), each of which is responsible for a roughly equal sized chunk of data that is aligned with the encryption blocks. Moreover, we maintain a central reader/writer pool manager. Each time a thread wants to read or write data, it first requests a reader/writer from the pool manager. The pool manager then assigns a reader/writer to the thread, if available. The reader/writer then reads or writes data in parallel with other readers/writers. When the reader/writer finishes its job, it returns to the pool manager. When a reader/writer is fully consumed, it is destroyed and the thread requests a new reader/writer from the pool manager. The contention is minimized since synchronization with the pool manager only happens when a task begins or ends and when a reader/writer is created or destroyed.

  An example of this case is the input and output of the shuffling phase, since we cannot assume that the input and output page size align with the algorithmic parameters, and the number of data in each task is not perfectly equal (especially for the output since as bucket has a different number of real elements). On the other hand, we don't care about the order of input and output data in a random shuffling algorithm.

- **Unaligned and Ordered** Finally, we consider the case where each thread reads or writes unequal amount of data or the data is not aligned with the encryption blocks, and we need to read or write data in an ordered and streaming fashion (either one or multiple streams). In this case, we can apply prefetching when reading the data and write-back buffers when writing the data. Since the data is read in a streaming fashion, we can prefetch the data to fill the buffer in parallel. Similarly, we can buffer the data to be written and write them back in parallel when the buffer is full. The buffer size is chosen to be large enough to ensure that the I/O operations are fully parallelized.

  One inefficiency of this approach is that we need an extra copy of the data due to the buffer, and the buffer also consumes some extra EPC space.

  An example of this case is the input and output of the external merge sort. We applied prefetching when reading the data from multiple sorted chunk, and applied write-back buffers when writing the merged data to external memory.

3.5 Parallelize Element Movement

We applied SIMD to parallelize the element movement. Specifically, we used C++ intrinsics to accelerate element-wise oblivious swaps. We incorporated different instruction sets such as AVX512, AVX2, and SSE2 for broader processor

compatibility. Specifically, we used the following instructions:
_mm256_mask_blend_pd, _mm256_set1_epi32, _mm256_and_si256,
_mm256_xor_si256, _mm_mask_blend_pd, _mm_and_si128, _mm_xor_si128.

We observed that the 512-bit wide SIMD operations are actually slower than running two 256-bit operations in series on our processor. This is probably because the 512-bit operations are not fully pipelined due to the limited number of execution ports. Therefore, we used 256-bit operations for the AVX512 instruction set.

Notice that we always pad elements to a multiple of 8 bytes for best performance. Moreover, to minimize the number of SIMD operations, we pad data of size $32k - 8$ bytes to $32k$ bytes, where k is an integer.

For other places where we need to move elements, we used the memcpy function in the C++ standard library, which is already optimized with SIMD for the processor.

3.6 Parallelize Pseudo-Random Number Generation

We implemented the pseudo-random number generator with AES counter mode, which features hardware acceleration on our processor. Since the generator is stateful, it is inefficient to use a central pseudo-random generator and synchronize it among multiple threads. Instead, we use multiple pseudo-random generators, each of which is responsible for generating pseudo-random numbers for a subset of the elements.

3.7 Optimize Memory Utilization

We want to minimize the memory consumption of the algorithm to maximize the number of elements that can be processed in each batch. Memory fragmentation hence becomes a major concern. While the original serial implementation has addressed this issue, we need to be more careful when parallelizing the algorithm. One specific problem is that each thread requires a temporary buffer to perform the multi-way merge-split operation, and frequently calling new and delete would worsen the memory fragmentation. Another issue is that the OpenMP library also allocates some memory for thread management in the heap, which is not released until the end of the program and causes fragmentation after the algorithm completes. If we run the algorithm a second time, there might not be enough consecutive memory.

To solve these issues, we allocate a large chunk of memory at the beginning of the sorting function and use it as a memory pool. We then implement a custom memory allocator that allocates memory from the pool and releases memory back to the pool. This way, we can avoid memory fragmentation both during and after the execution of the function.

4 Results

4.1 Experimental Setup

We evaluated the performance of our sorting and shuffling algorithms on an Intel Xeon Platinum 8352S processor, which has 36 cores running at 2.2 GHz frequency. The processor supports Intel SGX v2 and up to 512 GB of EPC cache. Nevertheless, we only uses up to 1.3 GB of EPC cache in our experiments for faster initialization and wider compatibility with consumer-grade processors. The processor supports AVX512, AVX2, and SSE2 instruction sets, and hardware AES acceleration. We used Ubuntu 22.04 as the operating system.

We validate the correctness of our program via unit testing, and evaluated its performance at different thread counts, input size and EPC size.

4.2 Speedup Results

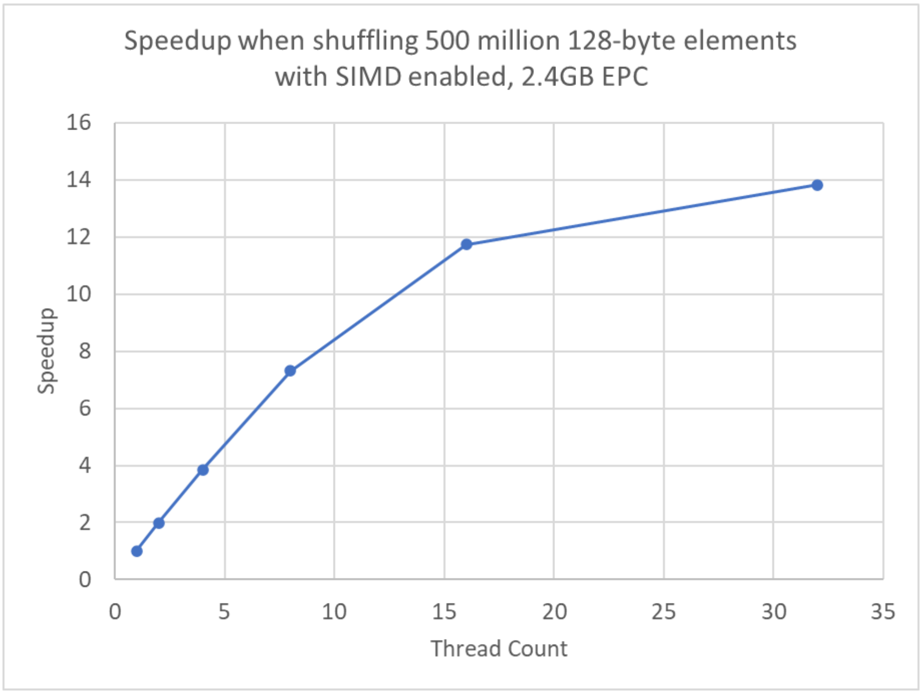

Figure 4: Speedup Diagram of parallel oblivious shuffling

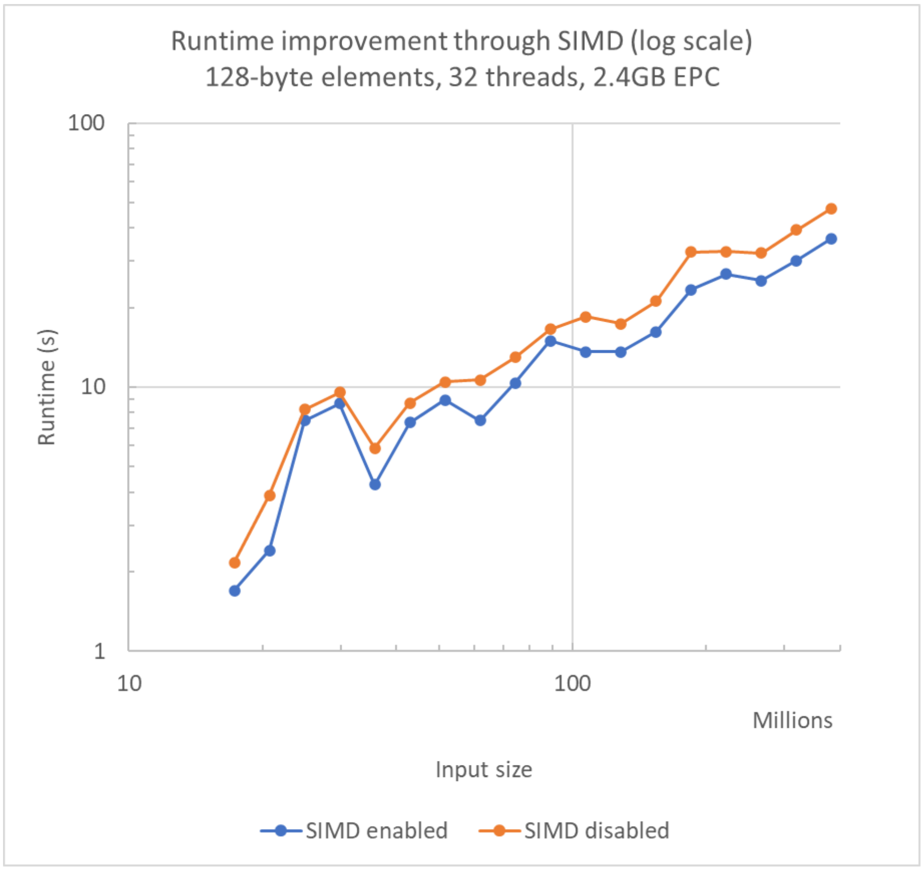

Figure 5: Speedup Diagram with SIMD Enabled

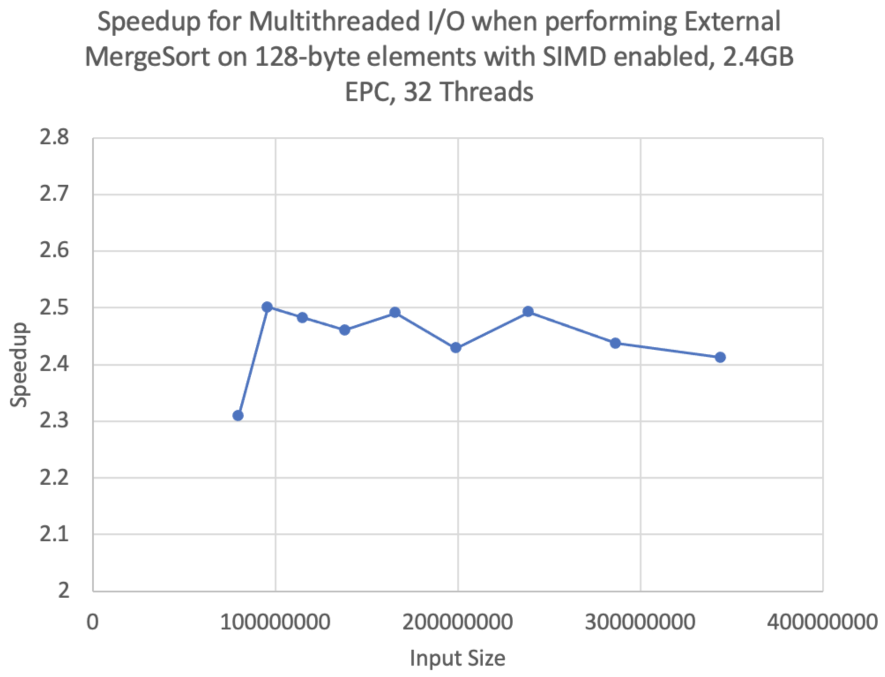

Figure 6: Speedup Diagram for External MergeSort with Multi-threaded I/O operations

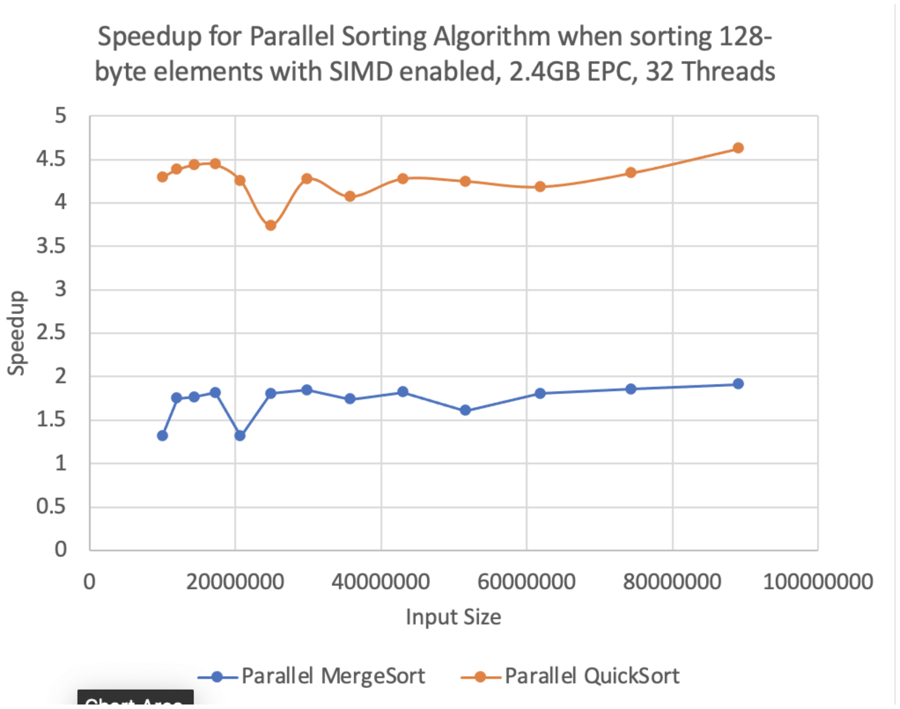

Figure 7: Speedup Diagram for Specific Parallel Sorting Algorithms We Adopted in Oblivious Sorting

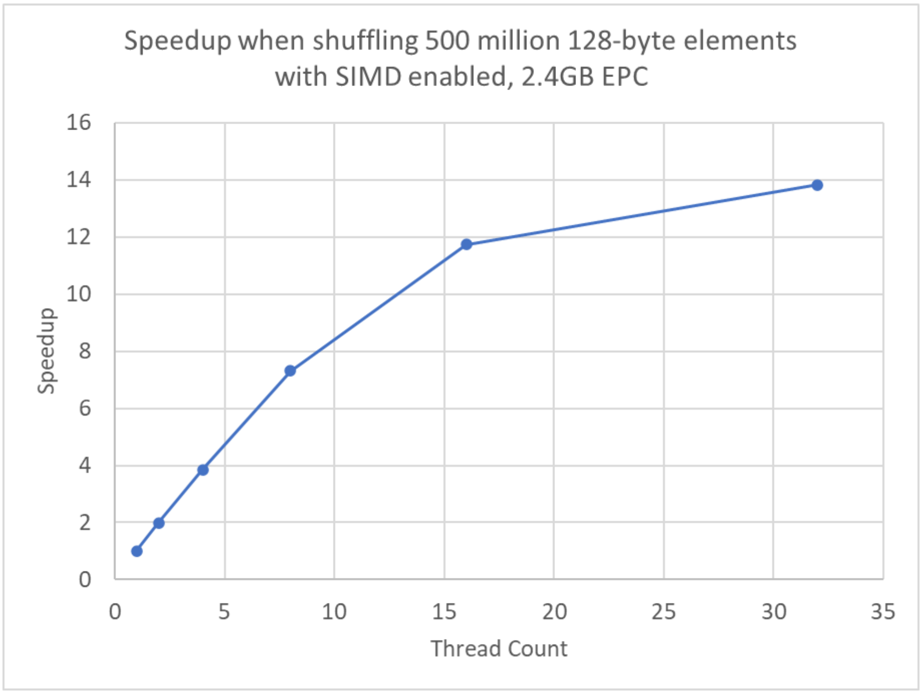

Figure 8: Overall Speedup for Parallel K-Way Butterfly Networking Oblivious Sorting

References

[1] Gilad Asharov et al. "Bucket Oblivious Sort: An Extremely Simple Oblivious Sort". In: *SOSA*. 2020.

[2] Kenneth E. Batcher. "Sorting Networks and Their Applications". In: *AFIPS*. 1968.

[3] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. "Data-oblivious Graph Algorithms for Secure Computation and Outsourcing". In: *ASIA CCS*. 2013.

[4] Michael T. Goodrich and Michael Mitzenmacher. "Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation". In: *ICALP*. 2011.

[5] Tianyao Gu et al. *Efficient Oblivious Sorting and Shuffling for Hardware Enclaves*. Cryptology ePrint Archive, Paper 2023/1258. `https://eprint.iacr.org/2023/1258`. 2023. URL: `https://eprint.iacr.org/2023/1258`.

[6] Aisha Hasan, Ryan Riley, and Dmitry Ponomarev. "Port or Shim? Stress Testing Application Performance on Intel SGX". In: *2020 IEEE International Symposium on Workload Characterization (IISWC)*. 2020, pp. 123–133. DOI: `10.1109/IISWC50251.2020.00021`.

[7] Muhammad El-Hindi et al. "Benchmarking the Second Generation of Intel SGX Hardware". In: *Proceedings of the 18th International Workshop on Data Management on New Hardware*. DaMoN '22. Philadelphia, PA, USA: Association for Computing Machinery, 2022. ISBN: 9781450393782. DOI: `10.1145/3533737.3535098`. URL: `https://doi.org/10.1145/3533737.3535098`.

[8] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973. ISBN: 0-201-03803-X.

[9] Wei-Kai Lin, Elaine Shi, and Tiancheng Xie. "Can We Overcome the $n \log n$ Barrier for Oblivious Sorting?" In: *SODA*. 2019.

[10] Chang Liu et al. "ObliVM: A Programming Framework for Secure Computation". In: *IEEE Symposium on Security and Privacy*. 2015.

[11] Vijaya Ramachandran and Elaine Shi. "Data Oblivious Algorithms for Multicores". In: *SPAA '21: 33rd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, 6-8 July, 2021*. ACM, 2021, pp. 373–384.

[12] Emil Stefanov et al. "Path ORAM: An Extremely Simple Oblivious RAM Protocol". In: *J. ACM* 65.4 (2018), 18:1–18:26.

[13] *Technology Deep Dive: Building a Faster ORAM Layer for Enclaves*. `https://signal.org/blog/building-faster-oram/`. 2022.

[14] Afonso Tinoco, Sixiang Gao, and Elaine Shi. "Enigmap : External-Memory Oblivious Map for Secure Enclaves". In: *Usenix Security*. 2023.