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 optimized the memory utilization of the program to minimize the consumption of Enclave Page Cache. We also overlapped computation and communication through prefetching and write back buffers.

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, which is similar to the disk I/O model in the conventional operating systems. Therefore, 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.

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}.

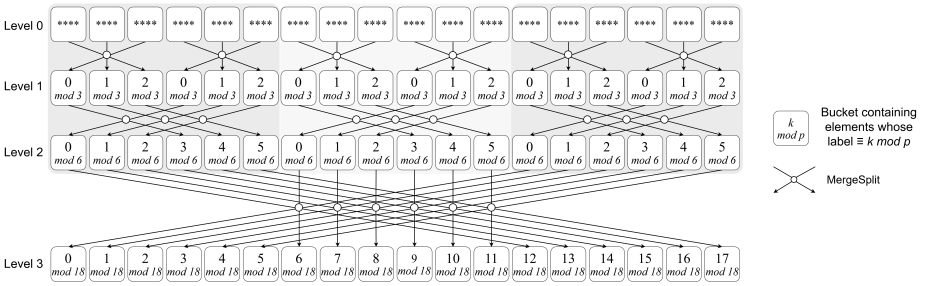

Figure 1: A 3-level multi-way butterfly network.

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.

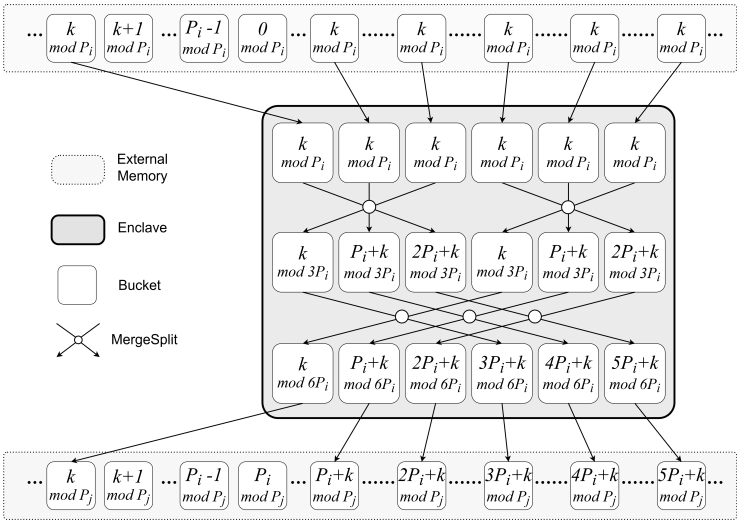

Figure 2: Route elements through multiple levels of butterfly network in each batch.

2 Approach

2.1 Identify Parallelism Abstraction

2.1.1 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 the bitonic sort within each bucket. Notice that we can only parallelize the merge-split operations within each batch since we cannot access data outside the enclave directly.

2.1.2 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 theoretical 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.

2.1.3 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 2.1.4.

2.1.4 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.

2.1.5 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 2.2.4.

2.2 Implementation

2.2.1 Parallelize Butterfly Network

2.2.2 Parallelize External Merge Sort

2.2.3 Parallelize I/O

2.2.4 Parallelize Element Movement

3 Results

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.