algorithms

MDPI

Article

Revisiting the Design of Parallel Stream Joins on Trusted Execution Environments

Souhail Meftah [1,2,*], Shuhao Zhang [3], Bharadwaj Veeravalli [1] and Khin Mi Mi Aung [2]

[1] College of Design and Engineering, National University of Singapore (NUS), Singapore 117575, Singapore; elebv@nus.edu.sg
[2] Institute of Infocomm Research, Agency for Science, Technology and Research (A*STAR), Singapore 138632, Singapore; mi_mi_aung@i2r.a-star.edu.sg
[3] Information Systems Technology and Design (ISTD), Singapore University of Technology and Design (SUTD), Singapore 487372, Singapore; shuhao_zhang@sutd.edu.sg
* Correspondence: stusm@i2r.a-star.edu.sg

Abstract: The appealing properties of secure hardware solutions such as trusted execution environment (TEE) including low computational overhead, confidentiality guarantee, and reduced attack surface have prompted considerable interest in adopting them for secure stream processing applications. In this paper, we revisit the design of parallel stream join algorithms on multicore processors with TEEs. In particular, we conduct a series of profiling experiments to investigate the impact of alternative design choices to parallelize stream joins on TEE including: (1) execution approaches, (2) partitioning schemes, and (3) distributed scheduling strategies. From the profiling study, we observe three major high-performance impediments: (a) the computational overhead introduced with cryptographic primitives associated with page swapping operations, (b) the restrictive Enclave Page Cache (EPC) size that limits the supported amount of in-memory processing, and (c) the lack of vertical scalability to support the increasing workload often required for near real-time applications. Addressing these issues allowed us to design *SecJoin*, a more efficient parallel stream join algorithm that exploits modern scale-out architectures with TEEs rendering no trade-offs on security whilst optimizing performance. We present our model-driven parameterization of *SecJoin* and share our experimental results which have shown up to 4-folds of improvements in terms of throughput and latency.

Keywords: stream join; trusted execution environment; software guard extensions; message passing interface; high performance computing

Citation: Meftah, S.; Zhang, S.; Veeravalli, B.; Aung, K.M.M. Revisiting the Design of Parallel Stream Joins on Trusted Execution Environments. *Algorithms* **2022**, *15*, 183. https://doi.org/10.3390/a15060183

Academic Editors: Yunquan Zhang and Liang Yuan

Received: 31 March 2022
Accepted: 24 May 2022
Published: 25 May 2022

1. Introduction

Since the widespread adoption of IoT and 5G technologies, a growing number of industries shifted their data-processing paradigm to stream-processing as a major enabler to real-time, data-driven applications. The joining of multiple data streams is a common operation that is relevant to many stream processing applications, such as online data mining and interactive query processing [1]. Due to its significant computational complexity, significant research efforts have been devoted to the design and evaluation of parallel stream join algorithms exploring modern multicore architectures. However, there has been an underwhelming amount of work investigating how to achieve secure parallel stream join on confidential data streams. With the introduction of the general data protection regulation (GDPR) [2], complemented by numerous national counterparts worldwide such as the personal data protection act (PDPA) [3], industry actors could no longer turn a blind eye to privacy concerns due to the hefty penalties associated with regulatory non-compliance.

The appealing properties of secure hardware solutions such as trusted execution environment (TEE) including (1) low computational overhead, (2) privacy guarantees, and (3) reduced attack surface, have prompted considerable interest in adopting them for secure stream processing applications. A TEE protects data and codes by loading

them in a non-addressable and encrypted memory area called an enclave, and provides some form of attestation to its trust. Theoretically, applications deployed inside an enclave have the potential to achieve the same performance as when they are developed on conventional processors.

Unfortunately, parallelizing stream joins on TEE is non-trivial and poses three major challenges:

(1) the *enclave definition language* (EDL) puts a limitation on the data-types that can be communicated to the enclave, which often prompts major code refactoring and potential serialization efforts as only the most basic data-types are supported;
(2) the restrictive enclave size effectively limits software development capabilities and performance in applications that are data-driven or memory-intensive. A larger memory usage entails more page swap operations, triggering more cryptographic primitives, and hence, more computational overhead;
(3) the ECall and OCall function call interfaces that enable untrusted code and enclaves to communicate seamlessly impose a heavy performance penalty of 10 k–18 k CPU cycles whenever the application needs to enter or exit an enclave for any system call, including I/O operations. Heavy performance penalties would have to be endured if an algorithm is inappropriately designed based on TEEs. Unfortunately, there is no study on the design of parallel stream join algorithms in distributed environments with TEEs.

Related Works We review three types of related works: (i) works that focus on parallel stream joins, (ii) works leveraging trusted execution environments; and (iii) other possible privacy-preserving paradigms to achieve security for join processing.

(i) Parallel Stream Joins. Multiple works [4–6] have explored different ways to leverage parallel architectures, focusing on the efficiency of sliding-window processing. Ref. [7] introduced a partitioned, in-memory merge tree to curb the challenges rising from indexing highly dynamic data, whilst [8] proposed a shared memory parallel SHJ algorithm on multi-core processors for equi-based stream joins. Since the introduction of Streaming Joins as a Service (SJaaS) [1] by Facebook, the research trend switched to focusing on scalability and reliability. Ref. [9] proposed a distributed stream join processing that supports window-based joins and online data aggregation. Streaming HyperCube [10] is an algorithm that ensures a balanced load across all compute nodes optimally. None of these previous works considered privacy in their computations.

(ii) The Use of TEEs. Intel SGX's outreach efforts attracted the attention of many researchers in the systems and databases field. CreDB [11] is a datastore that mimics Blockchain guarantee of integrity using TEEs. Evidence shows that it can be used as a drop-in replacement for No-SQL stores such as MongoDB without adverse performance effects and with the added integrity benefit. EdgelessDB [12] is an SQL-based solution that architects a database for the SGX environment. It keeps the data on disk strongly encrypted and only decrypts data within the enclaves. Similarly, ProDB [13] provides a minimal adaptation of a conventional DBMS with oblivious RAM protocol on hardware enclave. Some solutions adopted other TEEs such as Arm TrustZone. DBStore [14] is an example of a DBMS that leveraged the technology to enhance the security of mobile devices. None of these solutions introduced novelties to the field of join processing. However, we take note of Opaque [15] and ObliDB [16], two solutions focused on obliviousness protocols that adapted some join algorithms for TEEs.

(iii) Other Privacy-Preserving Paradigms. The security guarantees offered by TEE are far from comprehensive and have been criticized by many works [17]. One of the most notorious security limitations of existing TEE architectures is their vulnerability to Micro-architectural attacks [18]. The most prominent ones explored by the research community are side-channel attacks (detailed in Software Grand Exposure [19]), and transient execution attacks [20]. We prompt the reader to understand the variety of issues [21] that need to be considered before accepting a TEE design as secure. As alternatives, technologies such as fully homomorphic encryption and secure multi-party computations grew in popularity.

However, these remain for the most part impractical in terms of computational complexity, hardware demands and overall system's runtime performance. For instance, an FHE-based system implemented in IBM's HELib [22] is 3 to 5 orders of magnitude slower than its plaintext counterpart for basic integer arithmetic. This gap in performance is further widened in complex systems when FHE-specific overheads such as bootstrapping are accounted for. Furthermore, cyphertext expansion issues put stringent constraints on Hardware/RAM requirements. Numerous efforts were deployed to curb the challenges associated with FHE [23,24], but to the best of our knowledge, the technology can never achieve near real-time performance as it was not initially designed with such requirement in mind.

Our Contribution. In this work, we present three main contributions:

(i) We evaluate the impact of the new hardware constraints brought forth by TEEs in general, and SGX in particular, on the performance of parallel stream join algorithms. Namely, we conduct experiments pertaining to the different design alternatives to parallelize stream joins such as executions approaches, join methods and partitioning schemes. We aim to offer a better understanding of how those design aspects interact with modern multicore processors when TEE hardware is involved. Through detailed profiling studies with our benchmark on SGX-powered Microsoft Azure virtual machines, we make the following key observations. First, directly porting native code on TEE hardware through third party solutions can induce severe performance penalties. Second, joining large workloads on SGX triggers the process of EPC paging which results in an exponentially increasing performance gap as the size of inputs increases. Third, the overhead introduced by enclave calls makes eager processing costly.

(ii) Studying these issues allowed us to identify the different data-related and hardware-related parameters involved in stream join operations on SGX. We present and discuss a model of the performance overhead of running these operations under a very constraining hardware and suggest an optimal parameterization of the design alternatives of stream joins on SGX.

(iii) Finally, we share our open source implementation of *SecJoin*, a model-guided secure stream join algorithm that aims to retain all the security guarantees of Intel SGX while minimizing the performance overhead compared to the program running in a native (non-SGX) environment. The evaluation based on three real-world workloads shows that both our model and *SecJoin* are effective in improving performance and demonstrate scalability to very large workloads whilst achieving up to 4-folds of improvements in terms of throughput and latency.

Outline of the paper. The remainder of this paper is organized as follows: We first present the relevant preliminaries and background in Section 2, then elaborate on the challenges of parallelizing stream joins on TEE in Section 3. Next, we discuss the different design aspects of to consider on multicore processors with TEE in Section 4 and explain the results from our profiling study in Section 5. Based on that, we discuss our proposed model and present our experimental results in Section 6. Finally, we discuss future works with closing notes in Section 7.

2. Preliminaries and Background

Now, we (i) formally define the stream join operation and highlight some relevant works in Section 2.1, (ii) discuss TEEs in general and Intel SGX in particular in Section 2.2, and (iii) describe the threat model we consider in Section 2.3. We summarize the notations used throughout this paper in Table 1.

Table 1. Notations used in this paper.

Notations	Description
$x = \{t, k, v\}$	An input tuple x with three attributes
R, S	Two input streams to join
$skew_{key}$	Key skewness (unique or zipf)
$skew_{ts}$	Timestamp skewness (uniform or zipf)
$dupe$	Average number of duplicates per key
v	Input arrival rate (tuples/ms)
w	Window length (ms)
N_R	Total Number of Tuples in Stream R
N_S	Total Number of Tuples in Stream S
$\|X\|$	Maximum Size of Tuple $\{R; S\}$ or resulting Join $\{J\}$
M_X	Number of Tuples in Join-Matrix Partition over Stream $\{R; S\}$
λ_X	Tuple arrival rate for $\{R; S\}$
Mem_{Encl}	The amount of effective enclave memory available
#Encl	The Number of Secure Enclaves Available
#Thr	The Number of CPU Threads Available
$\#EPC_S$	The Number of Enclave Page Cache Swaps within a given Enclave
L_{ECall}	The Access Latency of an Enclave Call (ECall)
L_{OCall}	The Access Latency of an Outside Call (OCall)
L_{EPC}	The Latency of an Enclave Page Cache Swap
$T_{\bowtie}^{Encl}$	The execution time of a join in a secure enclave
C	Sum of Initialization and shutdown overheads for app using exact enclave memory size
C_{Init}	Enclave Initialization overhead per extra EPC page
C_{Shut}	Enclave Shutdown overhead per extra EPC page

2.1. Stream Joins

The need to obtain joining results "early" (before having read an entire input stream) has been a long identified problem by the research community. We define a *tuple* x as triplet $x = t, k, v$, where t, k and v are the timestamp, key, and payload of the tuple, respectively. We define the *input stream* (denoted as R or S) as a list of tuples chronologically arriving at the system (e.g., a query processor).

In this work, we focus on *intra-window join*. It is particularly important for emerging application demands that require maintaining large buffers of historical states [25,26]. In the following, we simply denote it as *stream join* for brevity.

Definition 1 (Stream join)**.** *Given input streams R and S and a window w, the **stream join** joins a pair of subsets (i.e., R', S') such that $R' \bowtie S' = \{(r \cup s) | r.key{=}s.key, r.ts \in w, s.ts \in w, r \in R, s \in S\}$, where each result tuple $(r \cup s)$ has a timestamp, key, and value of* $\max(r.ts, s.ts)$*, $r.key$, and $r.value \parallel s.value$, respectively.*

Earlier work on stream join [27,28] historically focused on its single-thread execution efficiency focusing on taking care of out-of-memory issue [28–32] or providing higher statistical quality of intermediate aggregation results [33,34]. To cope with the rapid growth of volume of data streams, much effort has been recently put into designing distributed and parallel stream join algorithms [26]. However, to the best of our knowledge, no one has

attempted answering the question of how to design efficient parallel stream join algorithms with security guarantees.

2.2. Trusted Execution Environments

Trusted Execution Environments (TEEs) are expected to provide hardware-enforcement mechanisms, such as sealed storage, memory encryption, and hardware secrets, to protect private computing from untrusted users and processes. They also must enable secure inter-enclave communication through local and remote authentication assertion, also known as attestation. This effectively protects against malicious parties with root privileges and creates a reverse sandbox that protects enclaves from remote attacks, operating systems, hypervisors, firmwares and drivers. As the adoption of TEEs is becoming increasingly popular in the industry, many more flavors of the technology are being commercialized. Yet, the most widespread solutions remain Intel's Software Guard Extentions (SGX) [21], AMD's Secure Encrypted Virtualization (SEV) [35], and ARM's TrustZone [36].

SGX-based TEEs. In our study, we focus on Intel SGX as the most promising solution available today for general-purpose computing. The SGX threat model assumes all privileged software is potentially malicious and provides integrity and confidentiality guarantees by isolating the enclave's code and associated data from the operating system, hypervisor and other hardware attached to the system, effectively reducing the attack surface of an application as illustrated in Figure 1.

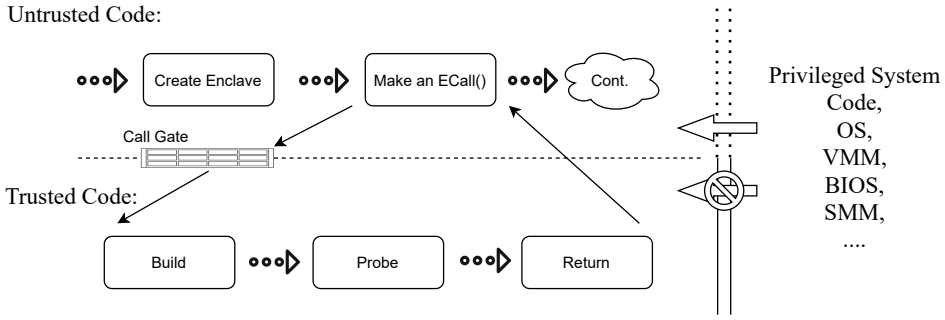


Figure 1. Application execution flow on Intel SGX.

It encrypts the enclave memory with a 128-bit key that randomly changes every power-cycle. Intel's industry partners such as Fortanix, Anjuna and Scone have offered runtime security frameworks that promise the capability of porting existing applications to an SGX Enclave without the need for code modifications. These services, although successfully curbing development overheads, do not offer proper application-specific optimizations and end-up suffering from the high performance penalties imposed by SGX.

2.3. Threat Model

Intel SGX offers integrity and confidentiality protection to both the code and the data laying within its enclaves. These security guarantees, as illustrated in Figure 1, are designed to hold even in the event of a privileged system, operating system or BIOS compromise. However, the protection of the components is beyond the scope of SGX. We assume the streaming data generators, as well as the communication gateway in charge of batching are trusted, and that data in transit between the stream generators and communication gateway is transferred following industry-standard security protocols.

3. Challenges of Parallelizing Stream Joins on TEEs

The appealing properties of TEEs motivate us to adopt them for secure stream joins in this work. However, this approach brings forth numerous hurdles; and there is still a lack of out-of-the-box solutions available in the industry.

3.1. Challenges

In this section, we summarize three major challenges of Parallelizing Stream Joins on TEEs.

Challenge 1: Enclave Definition Language (EDL). Accessing and exiting an enclave as well as marshaling the parameters passed accross the trusted/untrusted domains is done through custom routines called ECall and OCall. Marshaling the parameters communicated into the enclave is meant to curb the security vulnerabilities associated, such as Spectre [37]. However, this puts a limitation on Hash-based join algorithms [38,39] since the hash table data-type cannot be natively communicated to an enclave, which often prompts major code refactoring and potential serialization efforts as only the most basic data-types are supported.

Furthermore, the SGX Software Development Kit (SDK), provided by intel to kick-start developments using their technology, defines a syntax reference that does not seem to support all common programming features. For instance, among many more, private methods, switchless calls, and reentrant calls are not supported. Whether Intel is planning to improve on the flexibility of its platform whilst maintaining the same level of security is unclear at the time of writing this paper.

Challenge 2: Restrictive enclave size available. In most SGX-powered commercial cloud solutions, only 128 MB is allocated to the Processor Reserved Memory (PRM), of which 93 MB is for the Enclave Page Cache (EPC) and 35 MB is reserved for the metadata. Although larger enclave memory is starting to recently be supported, the upper-bound remains constraining. There are a multitude of reasons why the amount of memory available to an enclave application is this limited; on top of the list is the integrity tree depth and size which scales badly with the amount of memory being protected, leading to poor cacheability, high bandwidth penalties and memory capacity overheads [40]. This especially limits sort-based join algorithms that are not optimized for NUMA Systems [41], as large streams of data cannot be directly loaded for sorting within the enclave.

Challenge 3: Enclave routines' performance overhead. Previous works have shown that SGX has a considerable trusted memory footprint that causes performance degradations of up to 1000 folds [42]. The ECall and OCall function call interfaces that enable untrusted code and enclaves to communicate seamlessly impose a heavy performance penalty of 10 k–18 k CPU cycles whenever the application needs to enter or exit an enclave for any system call, including I/O operations. Moreover, for all applications that exceed the amount of memory available within the enclave, a page swapping mechanism will be triggered. Given the encryption and security checks involved, up to hundreds of thousands of CPU cycles are entailed within each page-swap operation [43]. Enclave initialization and destruction overheads also correlate with the buffer size, which increases exponentially once we exceed the EPC memory available. For instance, for a buffer size of 160 MB, bare metal median enclave startup and shutdown times are 5.4×10^9 and 1.15×10^8 CPU cycles respectively [44].

3.2. Motivating Experiments

In this section, we present our motivating experiments.

Cache Effects on TEEs. Our micro-benchmarks show that exceeding the available EPC size is not the sole performance bottleneck of Intel SGX. Rather, exceeding the L3 cache also causes non-negligible performance detriments as it requires cache evictions which trigger security checks and cryptographic operations. We utilize the parallel memory bandwidth benchmark tool (pmbw) [45] to monitor the performance of 64-bit read and write operations using variable-length arrays on our Azure server (Described in Section 5.3). We then port it on Graphene-SGX [46] to compare the throughput from within an enclave. We notice that the SGX performance of the instructions almost perfectly matches the native runs, until we reach the last level cache (LLC); after which, a clear performance gap is noticed when the output of 20 runs is averaged as illustrated in Figure 2.

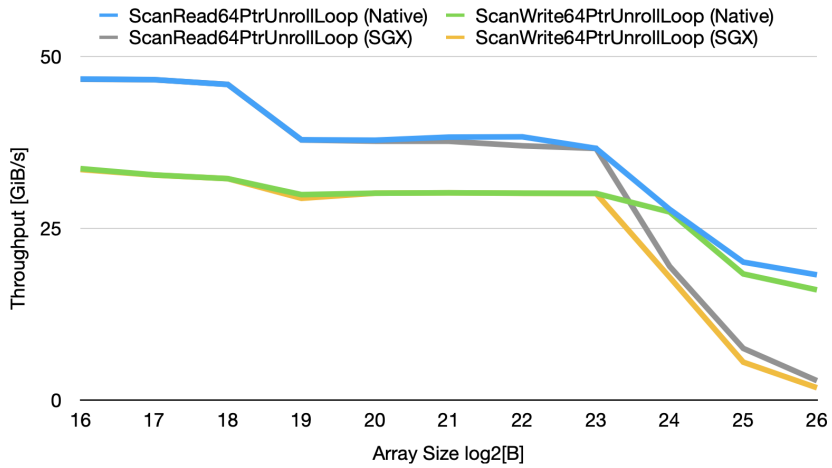


Figure 2. Caching Effects when executing read and write instructions, natively and on SGX for different array sizes.

Memory Allocation Overhead on TEEs. Next, we emphasize the importance of meticulous job scheduling on SGX by evaluating the overhead of allocating enclave memory during the initialization phase. We notice a significant "jump" in latency when the EPC size is reached in terms of encrypted memory demand, after which the slope depicting the change in latency rate increases from 1.5 ms/Mib to 4.7 ms/Mib as shown by the average of 30 runs illustrated in Figure 3. Note that the numbers depicted are for the sole memory allocation. Active usage and swapping of the memory pages in a complex application would entail an exponential increase of such overhead as shown in our profiling study in Section 5.

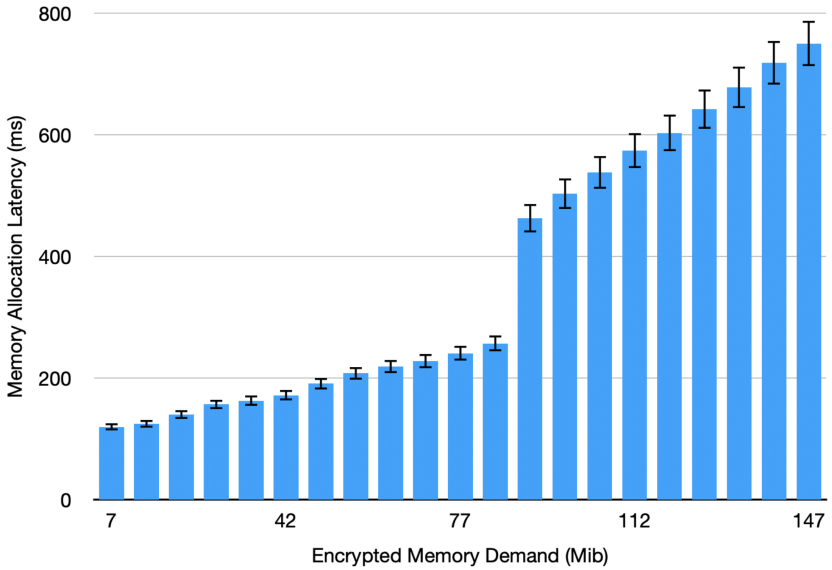


Figure 3. Memory allocation overhead on SGX with increased Encrypted Memory Demand.

4. Design Aspects of Parallelizing Stream Joins on Multicore Processors with TEEs

In the following, we introduce four design aspects of parallelizing stream joins on TEEs: (i) execution approaches in Section 4.1, (ii) join methods in Section 4.2, (iii) partitioning schemes in Section 4.3, and (iv) distributed scheduling strategies in Section 4.4. We summarize them in Table 2.

Table 2. Summary of Design Aspects.

Design Aspects	Design Decisions	Description
Execution Approaches	Lazy	Joins a complete set of tuples
	Eager	Joins a subset of tuples
Join Methods	Hash-Based	Builds hash tables before probing
	Sort-Merge	Sorts both relations before merging
Partitioning Schemes	Logical Partitioning	Passes pointers instead of values
	Physical Partitioning	Passes values instead of pointers
	Join-Matrix Partitioning	Content-insensitive matrix
	Join -Biclique Partitioning	Content-sensitive bipartite graph
Distributed Scheduling Strategies	Static Initialization	Assumes equal partition sizes
	Dynamic Adaptation	Adjusts partition sizes dynamically

4.1. Execution Approaches

We consider two fundamental *execution approaches* of parallel stream join: *lazy* and *eager*. The *lazy* approach initially buffers all input tuples of from both input streams over a given time window, and then joins a complete set of tuples. In contrast, the *eager* approach actively joins subsets of input tuples upon their arrival. Due to context and application dependency, it is difficult for researchers and practitioners to converge on an optimal approach to adopt. For instance, a lazy approach may be beneficial as it reduces enclave function call overheads and it can apply specific data-dependent optimizations before processing joins. However, it may also introduce additional processing latency due to the non-trivial performance overhead of EPC page swaps. In contrast, the length of a subset being processed under the eager approach is tunable. By tuning N_R and N_S, we may be able to achieve better processing performance through balancing the trade-offs between function call and memory allocation overheads.

4.2. Join Methods

Despite the large body of work on efficient join processing, join methods remain fundamentally different and hence, only contextually comparable in performance. Suggested optimizations need to be hardware-aware and able to generalize to arbitrary data flows. To that end, we consider in this study our own implementation of the Symmetric Hash Join (SHJ) [27,47]'s implementation of the Multiway Sort Merge (MWAY) as examples of Hash-Based and Sort-Merge join algorithms respectively. The essence in SHJ lies in interleaving the probe and build processes. It maintains a hash table for each stream. Each arriving tuple is immediately inserted into the corresponding hash table and probed in the opposite one. This process repeats until all arriving tuples are consumed. In contrast, the key idea in Sort-Merge algorithms is to first sort all relations by the join key before merging them. MWAY improves on the idea by efficiently partitioning the relations prior to independently sorting them. This enhances performance for NUMA systems. In the context of SGX, the inhibiting enclave size restricts the number of tuples that can loaded at once for sorting, hindering the effectiveness of the algorithm in the case of large datasets. Similarly, SHJ assumes both hash tables fit entirely in (secure) memory [31]. We would like to examine how these variables affect the overall system performance.

4.3. Partitioning Schemes

Parallel join algorithms are based on the theoretical foundation that the sets of records manipulated by a database query processing system can be partitioned into disjoint subsets, such that join results are computed independently across records. For lazy approaches, we may *physically* or *logically* partition input tuples into individual threads. The goal of the physical relation partitioning is to break at least the smaller input (i.e., tuples from *R*) into pieces that fit into the caches. Thus, it avoids the hash table being shared among

threads. However, it brings the additional cost of replicating tuples. Alternatively, we may only logically partition input tuples among threads by passing pointers. For eager approaches [5,6,10,26], there are two stream partitioning schemes that have been proposed. We revisit the impact of both of them on TEEs: (1) *join-matrix* [26] and (2) *join-biclique* [9], where the former is content-insensitive, and the latter is content-sensitive. Intuitively, the join matrix model designs a join between two datasets R and S as a matrix, where each dimension corresponds to one relation. Alternatively, the join-biclique model organizes the processing units as a complete bipartite graph, where each side corresponds to a relation. It is superior in memory efficiency, but is sensitive to the consumption of network bandwidth for tuple routing. Figure 4 illustrates both partitioning schemes. Note that in the JM case, groupings are optional. Each join partition can be independently processed by a separate process.

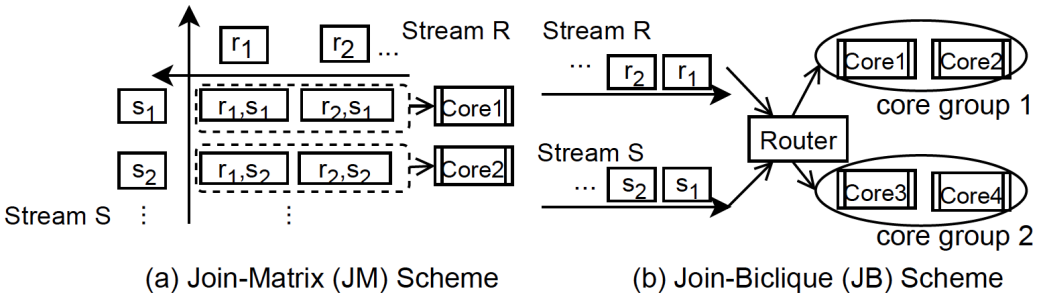


Figure 4. Join execution order under the Join-Matrix Scheme.

4.4. Distributed Scheduling Strategies

Whilst dynamic scale-out servers for SGX are not yet supported by cloud service providers, we believe that the technology could greatly benefit from horizontal scaling, especially given the currently imposed limitation on enclave sizes. Scale-out systems come in different flavors, but they essentially consist of an interconnected cluster of nodes distributing the processing load across multiple machines following user directives. We consider MPICH [48], a Message Passing Interface (MPI) implementation that combines in its design goals: wide portability and high performance. MPI is a message-passing API that provides abstractions for processes by providing them with ranks according to the communication groups they belong to, enabling a variety of virtual topologies that organize the application's semantics efficiently. It is considered the de-facto programming system on supercomputers and provides a natural interface for easier adoption by existing HPC applications [49]. In Algorithm 1, we elaborate on our suggested static initialization method in distributing the workload of secure join operations. The method assumes a constant tuple arrival rate, based on which optimal configuration parameters, such as the number of enclaves and join-matrix dimensions, are calculated as detailed in Section 6. Although inflexible to changes in the rate of arrival, this distribution method has the benefit of minimizing all TEE-induced overheads. In contrast, the adaptive distribution strategy periodically recalculates all parameters based on the varying arrival rate, which optimizes hardware resources utilization. However, this induces recurring enclave destruction and re-initialization. In both approaches, data orchestration is managed by a single central machine that receives the original streams and manages inter-machine communications using MPI.

Algorithm 1: Pseudo-code for an MPI Join distribution

```
call MPI_Init();
call MPI_Comm_size();
call MPI_Comm_rank();
//Initialize Enclaves on all machines
while Data_is_available do
    if world_rank==0 then
        inputs ← Buffer_stream(M_S, M_R);
        output_stream ← Serialize(inputs);
        string ← output_stream.str();
        length ← string.length();
        call MPI_Send(&Length);
        call MPI_Send(&inputs);
    end
    Set count to 0;
    if count % wold_size == world_rank then
        call MPI_Recv(&Length);
        Declare char buffer[Length];
        call MPI_Recv(&buffer);
        //Process join on local Enclave
        count++;
    end
end
```

5. Profiling Study

We now describe the methodology of our profiling study. We outline our evaluation goals in Section 5.1, discuss the datasets considered for the benchmarking workload in Section 5.2, describe the experimental setup in Section 5.3, present our preliminary results in Section 5.4.

5.1. Evaluation Goals

Our work aims to identify the alternative designs of parallel stream joins on TEEs, and to understand how those designs interact with TEEs-enabled modern multicore processors when running different real-world workloads. With the detailed profiling study, we hope to identify some hardware and software approaches to resolving the performance issues and point out the directions for the design and implementation of more efficient and secure parallel stream join algorithms.

5.2. Benchmark Workload

We follow the benchmark proposed in Table 3 to conduct the profiling study. The benchmark workload contains three real-world datasets: (1) *Rovio* continuously monitors the user actions of a given game to ensure that their services work as expected [50]; (2) *YSB* (Yahoo Stream Benchmark) [51] describes a simple job that identifies the campaigns of advertisement events and stores a window count of relevant events per campaign; (3) *DEBS* refers to a social network dataset published by the DEBS'2016 Grand Challenge [52]. When necessary, we resort to data duplication to ensure both streams have at least 1 million arriving tuples. Table 3 summarizes the tuple (buffer) sizes and attributes joined over in each dataset for our experiments.

Table 3. Join attribute and size of tuple per dataset.

	Tuple Size (in Bytes)	Attributes Joined Oven in R	Attributes Joined Oven in S
Rovio	30	1.2	1.2
YSB	100	1	1
DEBS	5000	3.5	4.6

5.3. Evaluation Setup

For our evaluation, we consider a cluster of Microsoft Azure's DCsv2 series of virtual machines that leverage Intel SGX. These machines are backed by 3.7 GHz Intel® Xeon E-2288G (Coffee Lake) with SGX technology. With Intel® Turbo Boost Max Technology 3.0, they can go up to 5.0 GHz. Azure offers 4 different configurations in this suite of VMs. We namely consider the Standard_DC8_v2, which is the most powerful SGX configuration offered by a cloud service provider we could find at the time of writing this paper. It offers, 8 physical cores; 32 Gib of memory; 400 Gib of SSD storage; 8 data disks; maximum cached and temporary throughput of 16000/128 IOPS/Mbps (Cache size in Gib); expected network bandwidth of 2 Mbps; and an EPC memory of 168 Mib.

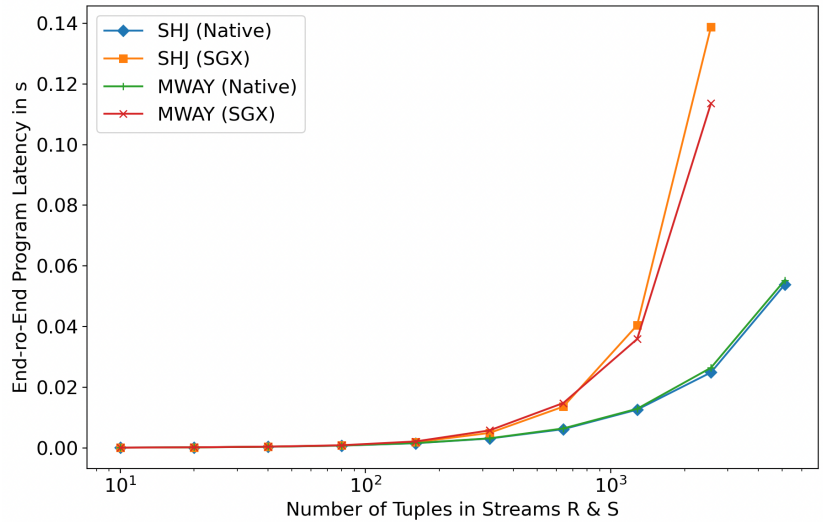
5.4. Preliminary Results

In the following, we show how the three design aspects (i.e., execution approaches, partitioning schemes, and join methods) interact with TEEs-enabled multicore processors.

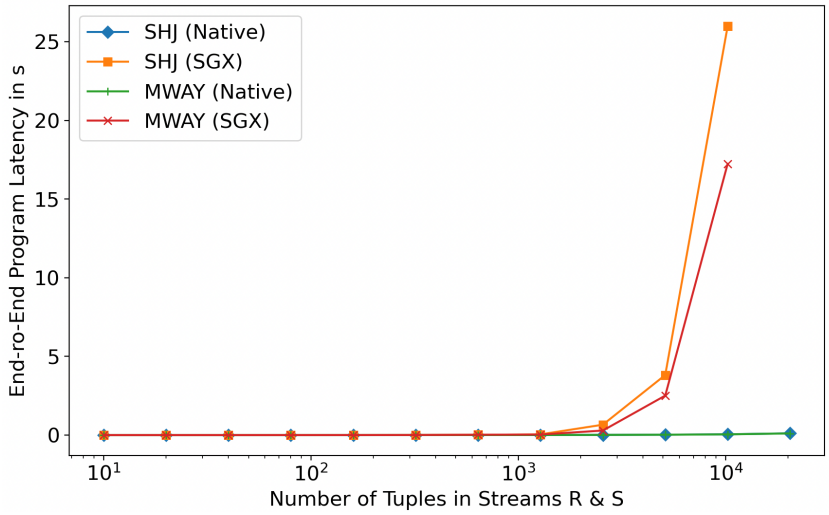
The Effect of Varying Execution Approaches. In practice, most off-the-shelf sort-merge algorithms cannot be readily ported on an SGX enclave without anticipating a considerable performance loss. Sort-merge algorithms require sorting the entire relations first, which can prove troublesome with very large datasets given the limited enclave memory available. Nonetheless, algorithms which have been optimised for NUMA architectures such as MWAY show considerable flexibility that can be leveraged in TEE. Nonetheless, as illustrated in Figures 5 and 6, lazy approaches in joining dramatically impact performance. We observe an exponentially increasing gap in both latency and throughput between the insecure and secure processing nodes as the number of joins to be processed increases. While the native mode is able to achieve up to 250,000 tuples/s on the YSB dataset, the SGX counterpart achieves less than 60,000 tuples/s. The significant throughput differences for different input sizes are mainly due to enclave memory swaps operations as the large data cannot be loaded and processed within the limited memory available to the enclave. In contrast, aggressive eager approaches also suffer from enclave call performance overhead that adds up for in each subset of tuples processed. Hence, a well informed hybrid approach needs to be considered.

The Effect of Varying Partitioning Schemes. In order to evaluate the impact of the varying partitioning schemes configurations, we first run mini-batches of different size streams R and S, such that: $M_R \times M_S = 10^6$. Such a small number of inputs fits entirely within the enclave memory. Averaging the results over 20 runs, we notice that square partitioning (where $M_R = M_S$) consistently yields minimal latency (up to $10\times$ less), whilst all other configurations fit within the reverse bell curve illustrated in Figure 7. This behavior is consistent across datasets and execution approaches.

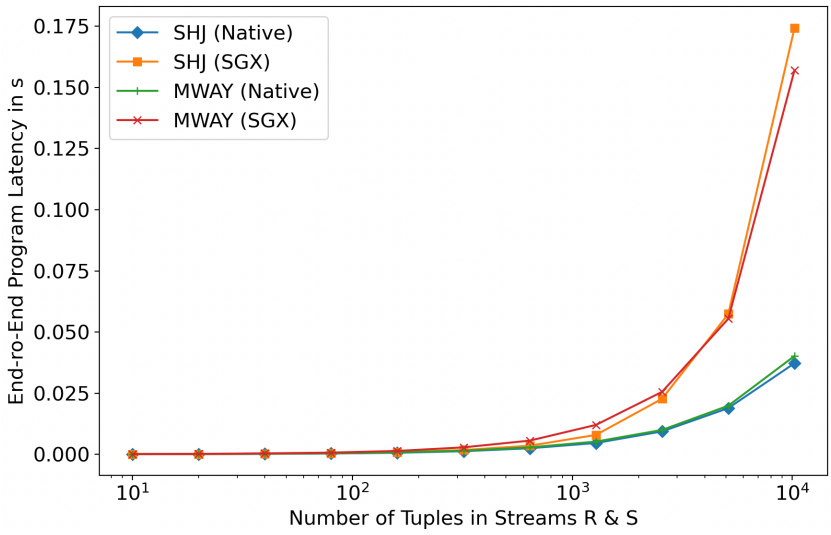
The Effect of Varying Join Methods. In Figures 5 and 6, we observe that with a few exceptions, SHJ generally outperforms MWAY in terms of throughput and latency for smaller workloads. This native behavior is carried forward to the secure implementation where small input sizes achieve a performance that is comparable to native runs. For larger workloads requiring EPC paging, MWAY appears to have a slight edge.



(**a**) DEBS

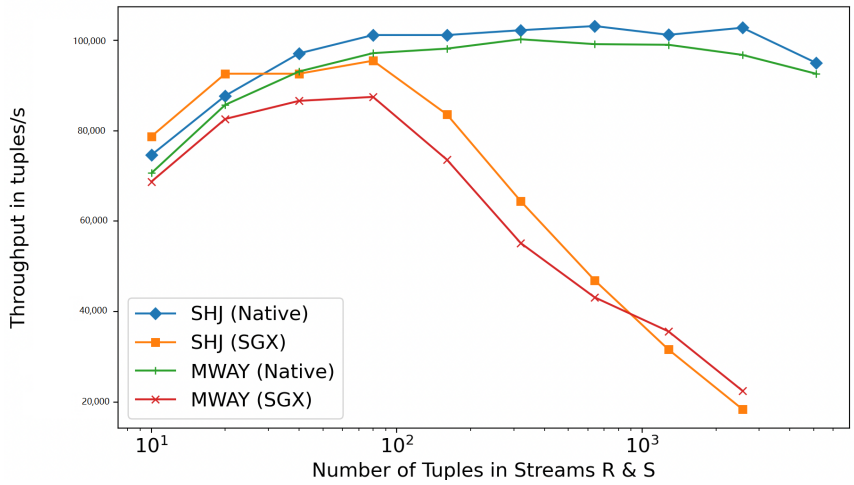


(**b**) Rovio

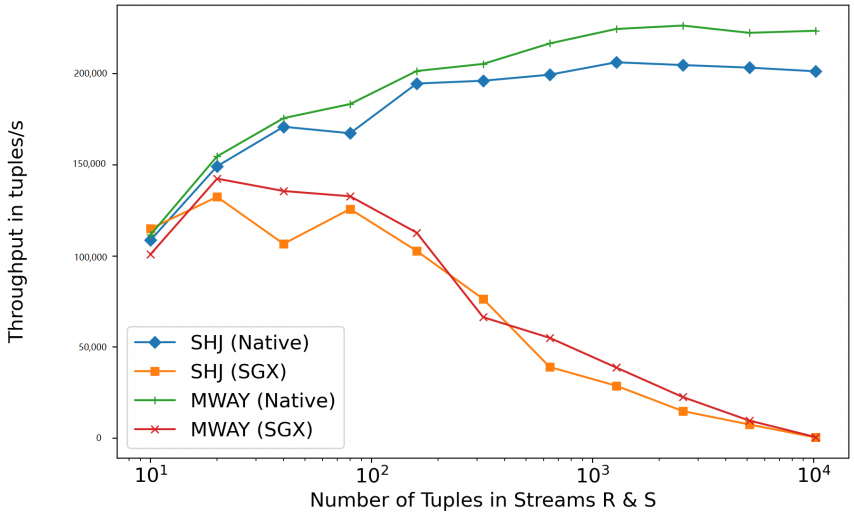


(**c**) YSB

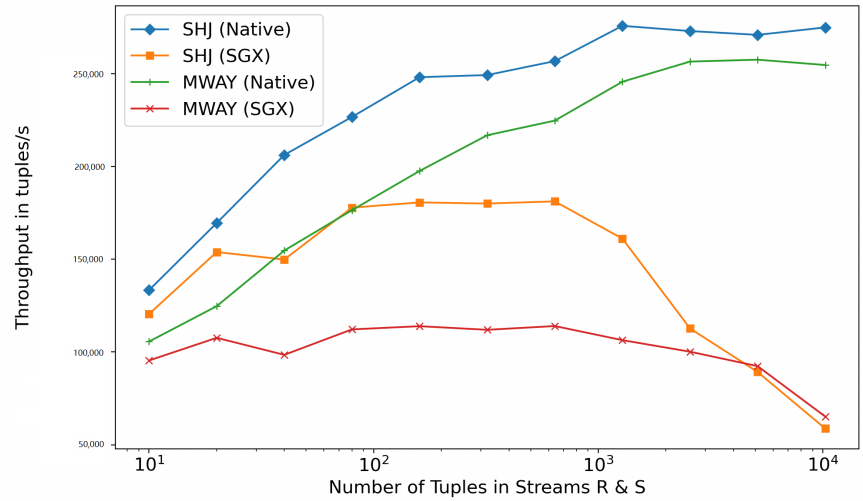
Figure 5. Average Latency with Different Processing Volume.



(**a**) DEBS



(**b**) Rovio



(**c**) YSB

Figure 6. Average Throughput with Different Processing Volume.

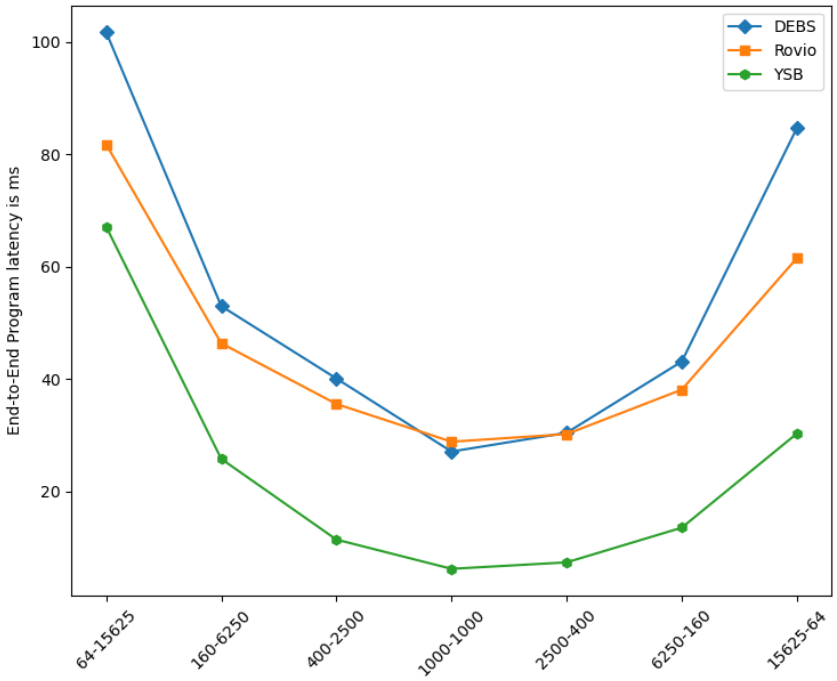


Figure 7. Effect of Partition Matrix Shape.

6. Towards More Efficient Parallel Stream Join on Multicore Processors with TEEs

In the baseline scenario of a standard stream join code ported directly to the sgx environment without any consideration for architectural differences, the performance overhead imposed by the SGX environment will be modeled as depicted in Equations (1) and (2). The secure enclave will have to allocate memory for both input streams as well as the output join. The size of the input streams is known to the developer beforehand, whilst the output join's size can only be determined after the computation is complete. Hence, the maximum memory required is allocated. We assume the size of the hashtable key values and auxiliary variables to be negligible in comparison.

$$\#EPC_S \approx \frac{(N_R \times |R| + N_S \times |S| + N_R \times N_S \times |J| - Mem_{Encl}}{4 \times 10^3} \tag{1}$$

$$T_{\bowtie}^{Encl} \approx L_{ECall} + L_{OCall} + \#EPC_S \times (L_{EPC} + C_{Init} + C_{Shut}) + C \tag{2}$$

During enclave initialization and shutdown (C), all EPC pages must be processed and deallocated respectively. Hence, the more memory is available to the enclave, the higher the initialization and shutdown overheads are expected to be. Note that Mem_{Encl} refers to the effective amount of memory available to the enclave as opposed to the total amount of memory accounted for. For instance, in a 128 Mib Enclave, Mem_{Encl} is 93 Mib.

With the introduction of SGX2 [53], the instruction set of SGX was extended to support dynamic memory allocation and deallocation which allows us to free the memory occupied by the input tuples as they are being processed into the hashtable. SGX2 also brings an enhancement to the previous TCS by allowing their allocation at runtime. This is enabled by the new capability of changing page permissions post-enclave-initiation. However, as elaborated on in [53], concurrency-related complexity in SGX2 is greatly increased over SGX1.

In contrast to the single-threaded baseline expressed in Equations (1) and (2), we consider multiple threads accessing the same enclave. Note that the threads are not dynamically spawned inside the enclave, but rather defined at the untrusted code level and access the enclave independently in parallel. In this context, the ECalls and OCalls performed will evoke a series of EPC page swaps to process the parameters associated;

the overhead of which will accumulate linearly in the worst-case scenario, as captured by Equation (3).

$$T_{\bowtie}^{Encl} \approx (\frac{N_R}{M_R} + 1) \times (\frac{N_S}{M_S} + 1) \times L_{ECall} + C \tag{3}$$

Finally, we consider a runtime environment whereby multiple secure enclaves are initialised. Ideally, the enclaves can be locally initialised on the server provided it can sufficiently scale-up. Or alternatively, an MPI (scale-out) configuration of servers can be established to provide the resources required. In such configuration, provided proper code design, the EPC size is no longer a limitation and only a single call into the respective enclaves need to be made, effectively reducing all hardware-induced overheads to the bare possible minimum expressed in Equation (4).

$$T_{\bowtie}^{Encl} = L_{ECall} + C \tag{4}$$

6.1. Optimal Partitioning Configuration

The key observation in Equation (4) motivates the use of an MPI distributed computational model for efficient stream processing on TEE. Yet, the cautious avoidance of the computationally expensive EPC page swaps operations is essential to achieve efficiency. Hence, we adopt the Join-Matrix partitioning scheme as it naturally lends itself to our distributed model.

Configuring the matrix dimensions of the partitioning scheme is approached pragmatically. With the assumption that streams R and S are arriving at the same rate (i.e., have an equal throughput), a buffering process is established to achieve the join-matrix dimensions calculated in Equation (5). These dimensions will ensure the efficient use of enclave memory without triggering extra EPC page swaps. Note that the tuple sizes are expressed in bytes.

$$M_R = M_S = \lfloor |-(|R| + |S|)/2|J| \pm \frac{\sqrt{(|R| + |S|)^2 - 4 \cdot Mem_{Encl} \cdot |J|}}{2 \cdot |J|} | \rfloor \tag{5}$$

However, when the throughputs of R and S are vastly different, the adoption of a square matrix will entail large waiting times, negatively impacting the overall performance of the system. Hence M_R and M_S need to satisfy the generalized Equation (6) that takes into account the rate of tuple arrivals, effectively optimising the waiting time during the batching process. For both cases, the number of batches processed independently by secure enclaves is expressed in Equation (7).

$$\begin{cases} M_R = \lfloor |-\lambda_R \cdot |R|/2 \cdot \lambda_S \cdot |J| - |S|/2 \cdot |J| \\ \pm \frac{\lambda_R \cdot \sqrt{(|R| + |S| \cdot \lambda_S/\lambda_R)^2 - 4 \cdot Mem_{Encl} \cdot |J|}}{2 \cdot \lambda_S \cdot |J|} | \rfloor \\ M_S = \frac{\lambda_S}{\lambda_R} \cdot M_R \end{cases} \tag{6}$$

$$\#Batches = \lceil \frac{N_R}{M_R} \rceil \cdot \lceil \frac{N_S}{M_S} \rceil \tag{7}$$

6.2. Optimal Hardware Requirements

Following the batching strategy described in Section 6.1 and considering different tuple arrival rates, #*Encl* can take up any value in the range [1, #*Batches*] applying a trade-off between runtime and hardware cost. In the event where hardware availability is not a constraint, the user can always over-allocate enclaves as MPI has been proven to

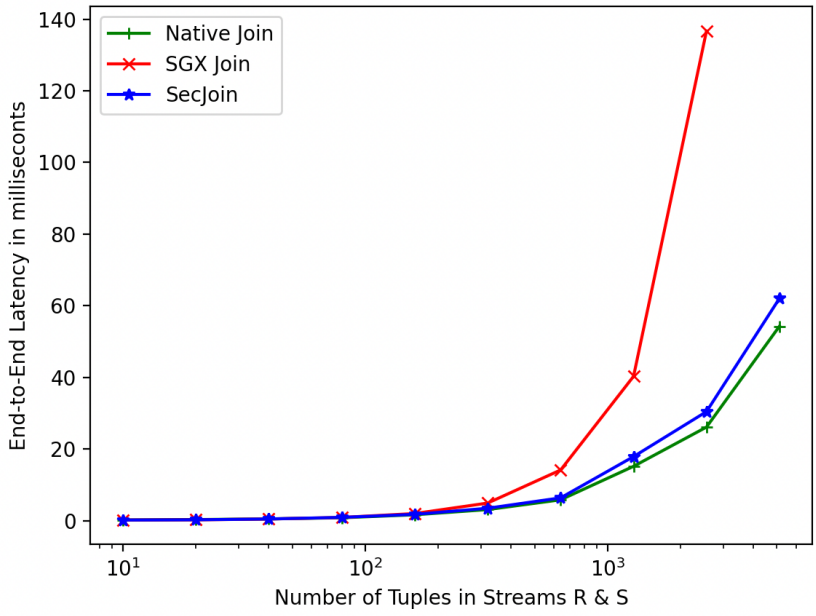
successfully scale up to thousands of nodes on a streaming workload [54]. Nonetheless, we devise Equation (8) to define an upper-bound, past which no performance gain is noted, with M_R and M_S being the values expressed in Equation (6).

$$\#Encl = \lceil \frac{N_R}{M_R} \rceil + \lceil \frac{N_S}{M_S} \rceil - 1 \tag{8}$$

6.3. Put It All Together

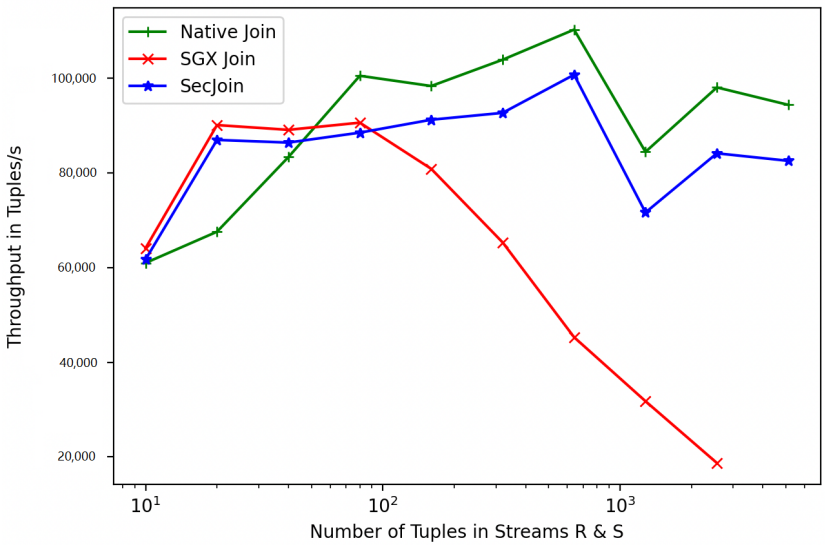
A good design for parallel stream join processing on SGX is one that benefits the data from all the security guarantees that TEE offers whilst minimizing the performance overheard introduced in comparison to running the same join operation in a non-SGX environment (referred to in what follows as a Native join). Currently, the performance gap between a Native join and its SGX implementation is quite large and further increases exponentially as the number of tuples in the input stream increases. This is impractical for large-scale secure join operations. Hence, considering the current scale-up limitations of SGX hardware, a model-guided scale-out solution is required to enable the optimal use of hardware resources available given the metadata provided by the user (See Section 6).

For simplicity, we refer to our custom solution as *SecJoin*. To investigate the effectiveness of our model-driven parametrization of *SecJoin*, we first write an SGX implementation of the Symmetric Hash Join algorithm following the Join-Matrix partitioning scheme and execute a join operation over two columns. The experiment is repeated over streams of incrementally equal sizes. As expected, we notice in Figure 8 that the SGX implementation matches the Native code in performance at first, then develops a performance gap that increases exponentially as we increase the number of tuples until the program is eventually killed by the kernel around the 5000 input mark due to lack of memory. Meanwhile, the SecJoin implementation consistently delivered a quasi-native performance with negligible overhead and reliable scalability potential. Analysing the sample run utilising 5000 tuples, we realise that over four-fifths of the end-to-end program latency is dedicated to SGX-induced overheads such as memory page swaps. On the other hand, SecJoin limits such overheads to a ratio of <5% as illustrated in Figure 9. Although introducing a new communication overhead due to the use of MPI for data orchestration, SecJoin still achieves up to 4-folds of improvement in both latency and throughput.



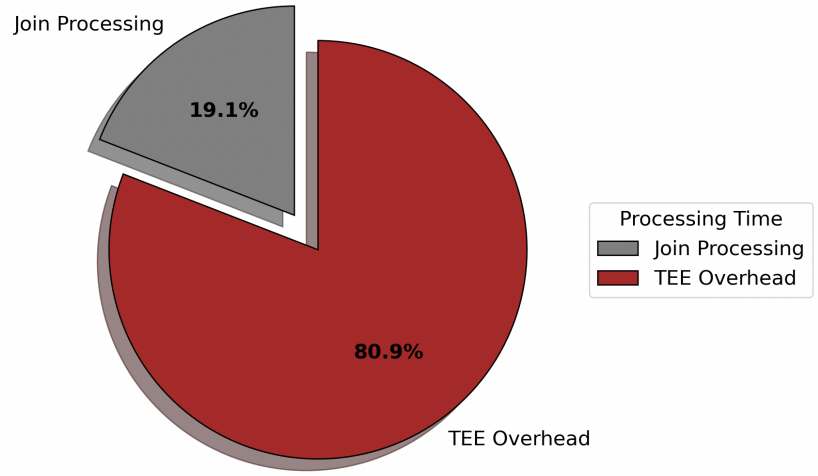
(**a**) Latency Comparison

Figure 8. *Cont.*

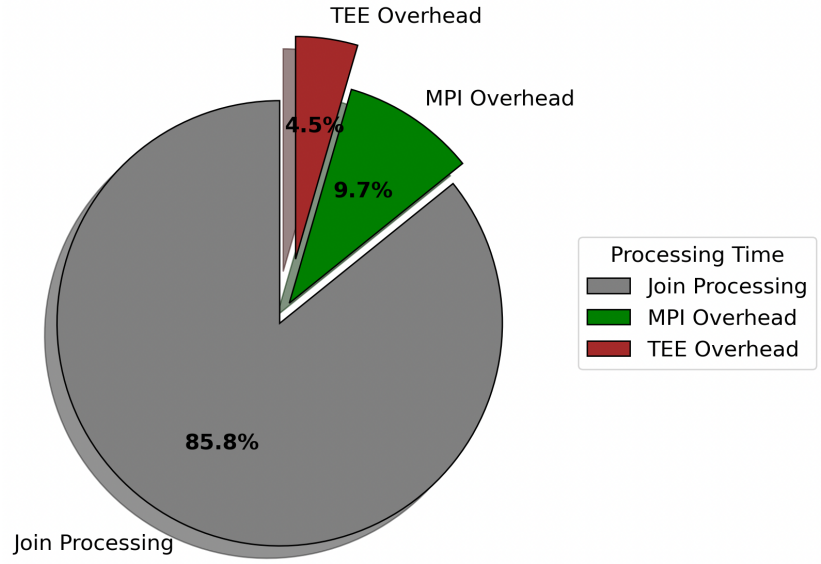


(**b**) Throughput Comparison

Figure 8. Performance Evaluation of SecJoin Compared to Native and SGX Joins.



(**a**) Native SGX Implementation Latency



(**b**) SecJoin Implementation Latency

Figure 9. Breakdown of End-to-End Program Execution Time.

7. Conclusions and Future Works

In this paper, we emphasized the importance of data privacy in join processing and outlined the current limitations of existing enablers. We conducted a profiling study to demonstrate that out-of-the-box join algorithms dramatically underperform in a TEE environment and expressed the need for a hardware-aware design to make the adoption of privacy practical. We presented our theoretical model to guide the design of such solution and calculate the different parameters to be considered. We evaluate our model through *SecJoin*. Results of the distributed algorithm demonstrated immense scalability potential as well as performance improvements of up to 4-folds. The Limitation of this study lies in its applicability to SGX 1 only as SGX 2 was not officially released yet at the time of writing this paper. We anticipate significant changes to the framework to be brought forth by Intel. For future works, we would like to extend *SecJoin* by implementing a dynamic resource allocation module that can automatically adjust to changing tuple arrival rates, enabling a true scale-out solution that can be deployed to the cloud. Also, we would like to extend our study and experimentation to encompass further join algorithms and design considerations that were not discussed in this work. Finally, we are considering a security-by-design approach in re-imagining join algorithms for TEE hardware.

Author Contributions: S.M.: Conceptualisation, Software, Validation, Writing—Original Draft; S.Z.: Conceptualisation, Methodology, Writing—Review and Editing; B.V.: Supervision, Writing—Review and Editing; K.M.M.A.: Supervision, Resources, Funding Acquisition. All authors have read and agreed to the published version of the manuscript.

Funding: Souhail Meftah and Khin Mi Mi Aung are supported by A*STAR under its RIE2020 Advanced Manufacturing and Engineering (AME) Programmatic Programme (Award A19E3b0099). Shuhao Zhang is supported by a SUTD Start-up Research Grant (SRT3IS21164).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

Sample Availability: An artifact description of this work is made available to the readers at: https://drive.google.com/file/d/1vQa9ck1RofsRVklf9fYlZB33uPMQjDOl/view?usp=sharing (accessed on 23 May 2022).

References

1. Jacques-Silva, G.; Lei, R.; Cheng, L.; Chen, G.J.; Ching, K.; Hu, T.; Mei, Y.; Wilfong, K.; Shetty, R.; Yilmaz, S.; et al. Providing Streaming Joins as a Service at Facebook. *Proc. VLDB Endow.* **2018**, *11*, 1809–1821. [CrossRef]
2. Linden, T.; Khandelwal, R.; Harkous, H.; Fawaz, K. The Privacy Policy Landscape After the GDPR. *arXiv* **2018**, arXiv:1809.08396.
3. Wong YongQuan, B. Data privacy law in Singapore: The Personal Data Protection Act 2012. *Int. Data Priv. Law* **2017**, *7*, 287–302. [CrossRef]
4. Najafi, M.; Sadoghi, M.; Jacobsen, H.A. SplitJoin: A Scalable, Low-latency Stream Join Architecture with Adjustable Ordering Precision. In Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC 16), Denver, CO, USA, 22–24 June 2016; pp. 493–505.
5. Roy, P.; Teubner, J.; Gemulla, R Low-Latency Handshake Join. *Proc. VLDB Endow.* **2014**, *7*, 709–720. [CrossRef]
6. Teubner, J.; Mueller, R. *How Soccer Players Would Do Stream Joins*; Association for Computing Machinery: New York, NY, USA, 2011; pp. 625–636. [CrossRef]
7. Shahvarani, A.; Jacobsen, H.A. *Parallel Index-Based Stream Join on a Multicore CPU*; Association for Computing Machinery: New York, NY, USA, 2020; pp. 2523–2537.
8. Behzadnia, P. Shared-Memory Parallel Hash-Based Stream Join in Continuous Data Streams. In Proceedings of the DEXA, Bratislava, Slovakia, 14–17 September 2021.
9. Lin, Q.; Ooi, B.C.; Wang, Z.; Yu, C. *Scalable Distributed Stream Join Processing*; Association for Computing Machinery: New York, NY, USA, 2015; pp. 811–825. [CrossRef]

10. Qiu, Y.; Papadias, S.; Yi, K. *Streaming HyperCube: A Massively Parallel Stream Join Algorithm*; Herschel, M., Galhardas, H., Reinwald, B.; Fundulaki, I., Binnig, C., Kaoudi, Z., Eds.; EDBT: Copenhagen, Denmark, 2019; pp. 642–645. [CrossRef]
11. Mast, K.; Chen, L.; Sirer, E. Enabling Strong Database Integrity using Trusted Execution Environments. *arXiv* **2018**, arXiv:1801.01618.
12. EdgelessDB—The SQL Database Tailor-Made for Confidential Computing. Available online: https://www.edgeless.systems/products/edgelessdb/ (accessed on 23 May 2022).
13. Han, Z.; Hu, H. ProDB: A Memory-Secure Database Using Hardware Enclave and Practical Oblivious RAM. Available online: https://www.sciencedirect.com/science/article/pii/S0306437920301332?via%3Dihub (accessed on 23 May 2022).
14. Ribeiro, P.S.; Santos, N.; Duarte, N.O. *DBStore: A TrustZone-Backed Database Management System for Mobile Applications*; ICETE: Panama City, Panama, 2018.
15. Zheng, W.; Dave, A.; Beekman, J.G.; Popa, R.A.; Gonzalez, J.E.; Stoica, I. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation, Boston, MA, USA, 27–29 March 2017; USENIX Association: Denver, CO, USA, 2017; pp. 283–298.
16. Eskandarian, S.; Zaharia, M. ObliDB: Oblivious Query Processing for Secure Databases. *arXiv* **2019**, arXiv:1710.00458.
17. Nilsson, A.; Bideh, P.N.; Brorsson, J. A Survey of Published Attacks on Intel SGX. *arXiv* **2020**, arXiv:2006.13598.
18. Jauernig, P.; Sadeghi, A.R.; Stapf, E. Trusted Execution Environments: Properties, Applications, and Challenges. *IEEE Secur. Priv.* **2020**, *18*, 56–60. [CrossRef]
19. Brasser, F.; Müller, U.; Dmitrienko, A.; Kostiainen, K.; Capkun, S.; Sadeghi, A.R. Software Grand Exposure: SGX Cache Attacks Are Practical. In Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT 17), Vancouver, BC, Canada, 14–15 August 2017; USENIX Association: Vancouver, BC, Canada, 2017.
20. Canella, C.; Bulck, J.V.; Schwarz, M.; Lipp, M.; von Berg, B.; Ortner, P.; Piessens, F.; Evtyushkin, D.; Gruss, D. A Systematic Evaluation of Transient Execution Attacks and Defenses. In Proceedings of the 28th USENIX Security Symposium (USENIX Security 19), Santa Clara, CA, USA, 14–16 August 2019; USENIX Association: Santa Clara, CA, USA, 2019; pp. 249–266.
21. Costan, V.; Devadas, S. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086. 2016. Available online: https://eprint.iacr.org/2016/086 (accessed on 3 October 2021).
22. Halevi, S.; Shoup, V. Algorithms in HElib. In *Advances in Cryptology—CRYPTO 2014*; Garay, J.A., Gennaro, R., Eds.; Springer: Berlin/Heidelberg, Germany, 2014; pp. 554–571.
23. Meftah, S.; Tan, B.H.M.; Mun, C.F.; Aung, K.M.M.; Veeravalli, B.; Chandrasekhar, V. DOReN: Toward Efficient Deep Convolutional Neural Networks with Fully Homomorphic Encryption. *IEEE Trans. Inf. Forensics Secur.* **2021**, *16*, 3740–3752. [CrossRef]
24. Meftah, S.; Tan, B.H.M.; Aung, K.M.M.; Yuxiao, L.; Jie, L.; Veeravalli, B. Towards high performance homomorphic encryption for inference tasks on CPU: An MPI approach. *Future Gener. Comput. Syst.* **2022**, *134*, 13–21. [CrossRef]
25. Castro Fernandez, R.; Migliavacca, M.; Kalyvianaki, E.; Pietzuch, P. Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, New York, NY, USA, 22–27 June 2013; ACM: New York, NY, USA, 2013; pp. 725–736. [CrossRef]
26. Elseidy, M.; Elguindy, A.; Vitorovic, A.; Koch, C. Scalable and Adaptive Online Joins. *Proc. VLDB Endow.* **2014**, *7*, 441–452. [CrossRef]
27. Wilschut, A.N.; Apers, P.M.G. Dataflow query execution in a parallel main-memory environment. In Proceedings of the First International Conference on Parallel and Distributed Information Systems, Miami Beach, FL, USA, 4–6 December 1991; pp. 68–77. [CrossRef]
28. Dittrich, J.P.; Seeger, B.; Taylor, D.S.; Widmayer, P. Progressive Merge Join: A Generic and Non-blocking Sort-based Join Algorithm. In Proceedings of the 28th International Conference on Very Large Data Bases, VLDB Endowment, Hong Kong, China, 20–23 August 2002; pp. 299–310.
29. Urhan, T.; Franklin, M.J. *Xjoin: A Reactively-Scheduled Pipelined Join Operatorë*; IEEE: Piscataway, NJ, USA, 2000; p. 27.
30. Mokbel, M.F.; Lu, M.; Aref, W.G. Hash-Merge Join: A Non-blocking Join Algorithm for Producing Fast and Early Join Results. In Proceedings of the 20th International Conference on Data Engineering, Boston, MA, USA, 2 April 2004; IEEE Computer Society: Washington, DC, USA, 2004; pp. 251–262. [CrossRef]
31. Lawrence, R. Early Hash Join: A Configurable Algorithm for the Efficient and Early Production of Join Results. In Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, 30 August–2 September 2005; pp. 841–852.
32. Tao, Y.; Yiu, M.L.; Papadias, D.; Hadjieleftheriou, M.; Mamoulis, N. RPJ: Producing Fast Join Results on Streams Through Rate-based Optimization. In Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, Baltimore, MD, USA, 14–16 June 2005; ACM: New York, NY, USA, 2005; pp. 371–382. [CrossRef]
33. Haas, P.J.; Hellerstein, J.M. Ripple Joins for Online Aggregation. *SIGMOD Rec.* **1999**, *28*, 287–298. [CrossRef]
34. Li, F.; Wu, B.; Yi, K.; Zhao, Z. Wander Join: Online Aggregation via Random Walks. In Proceedings of the 2016 International Conference on Management of Data, San Francisco, CA, USA, 26 June–1 July 2016; ACM: New York, NY, USA, 2016; pp. 615–629. [CrossRef]
35. Kaplan, D. *AMD x86 Memory Encryption Technologies*; USENIX Association: Austin, TX, USA, 2016.
36. Alves, T.; Felton, D. Trustzone: Integrated Hardware and Software Security. 2004. Available online: https://www.techonline.com/tech-papers/trustzone-integrated-hardware-and-software-security/ (accessed on 3 October 2021)
37. Hill, M.D.; Masters, J.; Ranganathan, P.; Turner, P.; Hennessy, J.L. On the Spectre and Meltdown Processor Security Vulnerabilities. *IEEE Micro* **2019**, *39*, 9–19. [CrossRef]

38. Barber, R.; Lohman, G.; Pandis, I.; Raman, V.; Sidle, R.; Attaluri, G.; Chainani, N.; Lightstone, S.; Sharpe, D. Memory-Efficient Hash Joins. *Proc. VLDB Endow.* **2014**, *8*, 353–364. [CrossRef]
39. Blanas, S.; Li, Y.; Patel, J.M. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs. In Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, Athens, Greece, 12–16 June 2011; Association for Computing Machinery: New York, NY, USA, 2011; pp. 37–48. [CrossRef]
40. Taassori, M.; Shafiee, A.; Balasubramonian, R. VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures. In Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, Williamsburg, VA, USA, 24–28 March 2018; Association for Computing Machinery: New York, NY, USA, 2018; pp. 665–678. [CrossRef]
41. Graefe, G. Sort-merge-join: An idea whose time has(h) passed? In Proceedings of the 1994 IEEE 10th International Conference on Data Engineering, Washington, DC, USA, 14–18 February 1994; pp. 406–417. [CrossRef]
42. Brenner, S.; Wulf, C.; Goltzsche, D.; Weichbrodt, N.; Lorenz, M.; Fetzer, C.; Pietzuch, P.; Kapitza, R. SecureKeeper: Confidential ZooKeeper Using Intel SGX. In Proceedings of the 17th International Middleware Conference, Trento, Italy, 12–16 December 2016; Association for Computing Machinery: New York, NY, USA, 2016. [CrossRef]
43. Zhao, C.; Saifuding, D.; Tian, H.; Zhang, Y.; Xing, C. On the Performance of Intel SGX. In Proceedings of the 13th Web Information Systems and Applications Conference (WISA), Wuhan, China, 23–25 September 2016; pp. 184–187. [CrossRef]
44. Dinh Ngoc, T.; Bui, B.; Bitchebe, S.; Tchana, A.; Schiavoni, V.; Felber, P.; Hagimont, D. Everything You Should Know About Intel SGX Performance on Virtualized Systems. *Proc. ACM Meas. Anal. Comput. Syst.* **2019**, *3*, 1–21. [CrossRef]
45. pmbw—Parallel Memory Bandwidth Benchmark/Measure. Available online: https://github.com/bingmann/pmbw (accessed on 3 October 2021).
46. Che Tsai, C.; Porter, D.E.; Vij, M. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 17), Santa Clara, CA, USA, 12–14 July 2017; USENIX Association: Santa Clara, CA, USA, 2017; pp. 645–658.
47. Kim, C.; Kaldewey, T.; Lee, V.W.; Sedlar, E.; Nguyen, A.D.; Satish, N.; Chhugani, J.; Di Blas, A.; Dubey, P. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *Proc. VLDB Endow.* **2009**, *2*, 1378–1389. [CrossRef]
48. Gropp, W.; Lusk, E.; Doss, N.; Skjellum, A. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Comput.* **1996**, *22*, 789–828. [CrossRef]
49. Peng, I.B.; Markidis, S.; Gioiosa, R.; Kestor, G.; Laure, E. MPI Streams for HPC Applications. *arXiv* **2017**, arXiv:1708.01306.
50. Karimov, J.; Rabl, T.; Katsifodimos, A.; Samarev, R.; Heiskanen, H.; Markl, V. Benchmarking distributed stream data processing systems. In Proceedings of the 2018 IEEE 34th International Conference on Data Engineering, Paris, France, 16–19 April 201; IEEE: Piscataway, NJ, USA, 2018; pp. 1507–1518.
51. Chintapalli, S.; Dagit, D.; Evans, B.; Farivar, R.; Graves, T.; Holderbaugh, M.; Liu, Z.; Nusbaum, K.; Patil, K.; Peng, B.J.; et al. Benchmarking streaming computation engines: Storm, flink and spark streaming. In Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Chicago, IL, USA, 23–27 May 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 1789–1792.
52. Gulisano, V.; Jerzak, Z.; Voulgaris, S.; Ziekow, H. The DEBS 2016 grand challenge. In Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems, Irvine, CA, USA, 20–24 June 2016; ACM: New York, NY, USA, 2016; pp. 289–292.
53. McKeen, F.; Alexandrovich, I.; Anati, I.; Caspi, D.; Johnson, S.; Leslie-Hurd, R.; Rozas, C. Intel® Software Guard Extensions (Intel® SGX) Support for Dynamic Memory Management Inside an Enclave. In Proceedings of the Hardware and Architectural Support for Security and Privacy 2016, Seoul, Korea, 18 June 2016; Association for Computing Machinery: New York, NY, USA, 2016. [CrossRef]
54. Peng, I.B.; Markidis, S.; Laure, E.; Holmes, D.; Bull, M. A Data Streaming Model in MPI. In Proceedings of the 3rd Workshop on Exascale MPI, Texas, AX, USA, 15 November 2015; Association for Computing Machinery: New York, NY, USA, 2015. [CrossRef]