UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Computer Science Curriculum

Kert Tali

Parallel and Cloud-Native Secure Multi-Party Computation

Master's Thesis (30 ECTS)

Supervisors: Riivo Talviste, Ph.D
Pelle Jakovits, Ph.D

Tartu 2022

Parallel and Cloud-Native Secure Multi-Party Computation

Abstract:
Secure multi-party computation (MPC) enables analysis based on sensitive data from multiple data owners, applying distributed cryptographic protocols to ensure privacy. Such protocols introduce distinct communication requirements, causing the computation to run significantly longer than its counterpart, conventional computing. General MPC frameworks are available that make it simple to develop such privacy-preserving applications, but running said applications assumes multiple non-colluding computing parties that host the protocol runtimes, having rigorously set up the required infrastructure. Utilising cloud resources for this occasion is a good alternative to on-premises deployments. First, it allows for a larger degree of automation in the infrastructure set-up. Secondly, cloud datacenters enjoy superior network characteristics, detrimental for MPC performance, and offer elastic compute resources at competitive price models. This thesis presents a cloud-native deployment of the SHAREMIND MPC framework on Kubernetes. It further proposes methods for parallel programming, with which MPC applications could be scaled over clusters. Familiar programming models, MapReduce and bulk-synchronous parallel, are adapted to MPC, and benchmarked in commodity clouds, showing near-linear speedup.
Keywords: secure multi-party computation, parallel computation, cloud-native applications
CERCS: P170 Computer science, numerical analysis, systems, control

Paralleelne ja pilvepõhine turvaline ühisarvutus

Lühikokkuvõte:
Turvaline ühisarvutus (MPC) võimaldab andmeanalüüsi mitme osapoole salajastel sisenditel, rakendades hajusaid krüptograafilisi protokolle säilitamaks andmete privaatsus. Erinevalt konventsionaalsest arvutusest vajavad taolised protokollid oma töö võimaldamiseks pidevat protsessidevahelist suhtlust. Üldotstarbelised MPC raamistikud tagavad programmeerijale hõlpsa viisi privaatsust säilitavate rakenduste loomiseks, kuid rakenduste endi jooksutamine eeldab mitme sõltmatu osapoole olemasolu, kes juurutaks vastava protokolli käitamise jaoks tarviliku taristu. Rakendades pilvteenuseid, on võimalik juurutamist automatiseerida enam, kui tundmatu võrgu ja riistvaraga keskkondades. Lisaks on pilve andmekeskustel MPC jõudlusele esmatähtis kiire võrguühendus ning võimalus maksta vaid kasutatud resursside eest. See töö sisaldab arhitektuuri SHAREMIND MPC pilvepõhiseks juurutuseks Kubernetese klastritel. Lisaks esitatakse viisid paralleelsetele MPC rakenduste loomiseks, mis neid kiirendaks, jaotades arvutused üle klastri. Implementeeritakse MapReduce ja *bulk-synchronous parallel* paradigmadel põhinevad rakendused, näitamaks nendega pea lineaarset kiirendust.
Võtmesõnad: turvaline ühisarvutus, paralleelarvutused, pilvepõhised rakendused
CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Contents

1 Introduction

How we process data depends on its characteristics. It would not be wise to operate on large scale data sets without careful planning to minimise cost and time. In this specific case, the techniques of big data optimise the means to extract useful information from huge pools of plain data within reasonable time, often employing parallel processing. For big data and in general, much of today's processing needs are covered by cloud providers, however data protected under policy or legislation often calls for on-premises computing. It is rightfully so, as the good faith and security of the providers is difficult to verify.

Data is an impactful asset that holds power over the success and well-being of businesses and individuals. Hence the need to protect delicate information from competitors or people with ill intent. This characteristic may impose a variety of restrictions on data usage to preserve privacy. Secure multi-party computation (MPC) presents a solution to private data processing. It involves cryptographic protocols and multiple non-colluding computing parties to securely operate on values without learning their actual content. For example, MPC schemes based on secret sharing do it by splitting cleartext inputs into seemingly random shares which can be operated with if all share holders simultaneously take part in the protocol, and recombined to reveal their true value. MPC can mean privacy preserving computation offloading to potentially devious parties, but also analysing private inputs of multiple data owners.

The cryptographic backbone of MPC does not come without significant cost in complexity and performance. Comparing MPC to conventional computing, one of the main differences is its distinct communication requirement. Certain operations, like checking the equality of two shared values, may require multiple rounds of communication per invocation between computing parties. The consequent blow-up in algorithm running time is attributed to why deployments of MPC in practice are few and far between.

MPC has found little adoption in large-scale enterprise applications. One reason for this can be speculated to be the absence of commercial offerings that support an enterprise deployment model, which in turn may be due to underwhelming demonstrated performance. So far, available general-purpose MPC frameworks can mostly be characterized as academic prototypes or platforms for cryptographers to evaluate new protocols on. Nevertheless, as the state-of-the-art in MPC gradually advances, the need for a platform feasible for today's enterprise landscape becomes more apparent. Such a platform should manifest cloud nativity along with making use of the elastic nature of cloud infrastructure.

With respect to these two issues, this thesis provides the following contributions:

1. Architecture and implementation for reproducible cloud deployment of MPC. The devised solution follows the current industry incline towards cloud-native technologies and presents a working prototype for automated deployment of the SHAREMIND MPC framework on Kubernetes clusters.

2. Employ the elasticity enjoyed from contribution 1 to horizontally scale MPC in effort to speed up execution of privacy-preserving data analysis. Proposed parallel programming models for MapReduce and bulk-synchronous parallel are demonstrated to exploit the new environment, following up with benchmarks of example applications on tabular and graph-based data respectively.

The ensuing work is laid out as follows. Chapter 2 provides an introduction to the concepts relevant for this work. Chapter 3 explores and reasons about the work of other authors that is akin. In chapter 4, a high level overview is given of the requirements for the planned deployment and parallel MPC programs. The architecture's design, implementation details, and evaluation are detailed in chapter 5. The final conclusion and suggestions for future work are presented in chapter 6.

2 Preliminaries

This section outlines the background of concepts that support the practical area of this thesis. Section 2.1 introduces foundations of MPC along with the SHAREMIND MPC framework which is the MPC backend of choice for this work. In section 2.2, the focus is on explaining general parallel programming paradigms and the motivation of their usage with MPC. Finally, the latest practices and concepts of cloud-native applications are given an overview in section 2.3.

2.1 Secure Multi-Party Computation

Secure multi-party computation is a cryptographic problem described by Yao [37] for computing on secret inputs of multiple participants. In it, n parties come together to jointly compute a known function f on their private inputs $x_1, \dots, x_n$ to find the output $y_1, \dots, y_n$.

$$(y_1, \dots, y_n) = f(x_1, \dots, x_n)$$

In the process, a party learns nothing about the inputs of its neighbours and only the output they are supposed to receive. The function f captures the functionality that is required of the private computation, which is generally such that all output values are dependent on all inputs. Given this, mutually distrusting participants are able to extract useful (aggregated) information from each other's knowledge while avoiding leakage.

The formal definition distinctly specifies $n \geq 2$ mutually distrusting parties that take the roles of input provider, computing party, and output receiver simultaneously. The author uses the millionaires' problem to illustrate such a scenario – two individuals who are adamant in hiding their net worth from each other want to know which one of them is richer; they engage in MPC, each inputting their cumulative wealth, calculating the comparison in private, and receiving the result [37]. It is accepted that practical MPC deployments models can vary in attributing those roles in different manners for supporting a wider range of business cases [21]. For example, input parties may be different from the parties receiving results, as was the case with a study by Bogdanov, et al [12], in which separate government institutions acting as data owners and computation parties aggregated their records to be output to the statistics bureau interested in the outcome. In addition, decoupling input parties from computation allows for relieving computational load from clients. Such a scenario was exercised by the Danish sugar beet auction [13], where MPC was used to process private bids in remote servers.

MPC relies on specific cryptographic protocols for computing f in which all participants engage in. In essence of a general MPC protocol, parties can be thought to hold encrypted representations of values they operate on. During the execution they may engage in protocol communication over network channels they have established between each other to securely evaluate the necessary expressions. Most prominent general purpose MPC protocol implementations are based on the *garbled circuits* or *secret sharing* techniques.

The garbled circuits technique as originally proposed by Yao [37] is a two-party asymmetric protocol for computing the result of a binary circuit encoding of f. Asymmetry is due to one party taking the role of the circuit garbler and the other being the evaluator. The garbler's task is to encrypt the truth tables of each of the circuit's gates and provide keys in such a manner that allows the evaluator to decrypt subsequent gate keys one by one, learning the result at the final *output layer*. Both parties can input values to the circuit – in the case of garbler-side values, the seemingly random starting keys corresponding to 0 or 1 are provided to the evaluator, while the evaluator obtains keys of their own inputs using an oblivious transfer protocol. Oblivious transfer is a construction which in this scenario allows the evaluator to query one of the two keys without revealing the corresponding bit. Garbled circuits is an attractive method for two-party computation which can represent any kind of computation, as it requires only a few communication rounds, however it can result in huge circuits, which are cumbersome to transfer over a network [6].

2.1.1 Secret Sharing

Secret sharing is another general primitive used as a building block for MPC protocols. Originally proposed by Shamir [31], the idea of a (k, n)-threshold sharing scheme was to split an element $v \in \mathbb{Z}_p$ into shares $v_1, \ldots, v_n$ to distribute among n parties so that

$$q(x) = v + a_1 x + \ldots + a_{k-1} x^{k-1}$$

and $v_i = q(i)$, where the coefficients of q are chosen uniformly at random. The shares can be reconstructed to v by at least k parties by first discovering the secret polynomial using Lagrange interpolation and evaluating it on $q(0) = v$. Since interpolation of a $k-1$ degree polynomial only works if k different evaluations are known, then it holds that at least the threshold of k parties have to come together to reveal it.

The feasibility of Shamir's secret sharing for MPC in terms of operations on the values is shown by Cramer, et al. [16]. The scheme conveniently supports linear operations to be calculated locally – adding and subtracting shares with constants or other shares requires no communication due to the properties of polynomials. Multiplication of two shares requires communication however, and can be accomplished as described by the authors.

A simpler method for secret sharing, although losing the threshold property, is to construct the shares to add up to v, called additive sharing. Further material will use the notation $[\![v]\!]$ to represent an additively shared value, in addition v_i is the corresponding additive share known by party $\mathcal{P}_i$. The construction of secret shares of a value $v \in \mathbb{Z}_m$ known to an input party is created by generating uniformly random values $r_1, \ldots, r_{n-1} \in$

$\mathbb{Z}_m$ to act as the shares for the first $n-1$ parties, and calculating the last share as

$$r_n = v - r_1 - \ldots - r_{n-1} \mod m.$$

Share r_i can then be distributed to $\mathcal{P}_i$ and reconstruction of v can only occur when all n parties collaborate. In fact, no coalition of parties of size less than n can infer anything about v since the values are chosen randomly and the derivation takes place in a ring. Linear operations are still free in terms of communication, meaning that they can be calculated locally by parties.

Algorithm 1: SHAREMIND protocol for secure multiplication [9]

```
   Input: Parties $\mathcal{P}_1$,$\mathcal{P}_2$,$\mathcal{P}_3$ hold $[\![u]\!]$ and $[\![v]\!]$
   Result: Parties $\mathcal{P}_1$,$\mathcal{P}_2$,$\mathcal{P}_3$ hold $[\![w]\!]$, the product of $[\![u]\!]$ and $[\![v]\!]$
1  $[\![u]\!]' \leftarrow$ Reshare($[\![u]\!]$)
2  $[\![v]\!]' \leftarrow$ Reshare($[\![v]\!]$)
3  foreach $i, j \in \{(1,2),(2,3),(3,1)\}$ do
4      send $u'_i$, $v'_i$ to $\mathcal{P}_j$;                          // Done by $\mathcal{P}_i$
5  foreach $i, j \in \{(1,3),(2,1),(3,2)\}$ do
6      $w'_i \leftarrow u'_i v'_i + u'_i v'_j + u'_j v'_i$;                // Done by $\mathcal{P}_i$
7  $[\![w]\!] \leftarrow$ Reshare($[\![w]\!]'$)
```

Procedure Reshare($[\![v]\!]$)

```
   Input: Additively shared value $[\![v]\!]$.
   Result: New additively shared value $[\![v]\!]'$, with new random $v'_i$ such that
           $\sum_i v'_i = v$
1  foreach $i, j \in \{(1,2),(2,3),(3,1)\}$ do
2      $r_{ij} \xleftarrow{\$} \mathbb{Z}_{2^m}$ ;                                  // Done by $\mathcal{P}_i$
3      send $r_{ij}$ to $P_j$ ;                                // Done by $\mathcal{P}_i$
4  foreach $i, j, k \in \{(1,2,3),(2,3,1),(3,1,2)\}$ do
5      $v'_i \leftarrow v_i + r_{ij} - r_{ki}$ ;                         // Done by $\mathcal{P}_i$
6  return $[\![v]\!]'$
```

The latter secret sharing scheme is used in the main three-party protocol of the SHAREMIND MPC framework. Communication requirements can be exemplified using its shared value multiplication procedure as shown in algorithm 1 [9]. Transmissions that do not have data dependencies can be batched together to optimise communication rounds. A single n-bit integer multiplication can be boiled down to a single round of communication of $15n$ bits [9]. In the additive scheme there are other operations that require even more communication, for example division, checking equality, inequalities, and bitwise operations on shared values.

2.1.2 SHAREMIND MPC

SHAREMIND MPC (hereinafter called SHAREMIND for brevity) is a framework for doing secure data analysis and storage with MPC [8]. The framework comprises an extensive set of facilities for developing and running MPC applications. This includes the SECREC 2 domain-specific language (DSL) and compiler for writing programs, server runtime for the computation, and client libraries for interacting with the servers. In this thesis, components of the SHAREMIND framework are used and built around to develop a system capable of scalable parallel computation.

SECREC is an imperative C-style language for developing privacy preserving data analysis applications [30]. Its standard library and programming facilities aim to make it straight forward to implement various business logic while not requiring extensive cryptographic prowess. Abstractions enable programmers to make distinctions between public information and data supposed to be kept secret. During the analysis task, they may for example choose to *declassify* some intermediate values to the parties to use them as control flow conditions, or *publish* end results to the client without the server ever learning the data in clear.

The SECREC language heavily relies on vectorised operations to facilitate performance optimizations, most prominently Single-Instruction Multiple-Data (SIMD) to batch together communication rounds that would have otherwise been executed one after another [30]. Early benchmarks show a four orders of magnitude increase in the multiplication and equality operations per second over using loops [9]. Therefore it is the programmers responsibility to employ this practice, otherwise their program will be heavily bottlenecked by synchronous back-and-forth protocol communication, the speed of which is dependent on the latency between parties.

The framework is modular in the types of protocols it supports – these are called *protection domains* within SHAREMIND. This work considers the SHARED3P protection domain, a protocol set for additively shared three-party computation with passive security, because it is the most mature and fully featured of the existing ones. Passive security means that the protocol does not protect against an active adversary – a corrupted party that works unfaithfully against the protocol by sending falsified data to its peers or alike. Nonetheless, the party is not able to infer any useful information about the private values by itself, due to additive secret sharing.

The SHARED3P protocol requires a set of three SHAREMIND servers to perform the computation on behalf of the client(s). The server instances run within the infrastructure of the logical party which exposes a public port, enabling incoming protocol communication from peers. Servers are configured to authenticate the neighbouring SHARED3P parties with mutual TLS and encrypt all following communication, with the public key of the recipient, with keys and network endpoints exchanged securely beforehand. On start-up, servers will seek to establish a connection with their protocol counterparts and will serve client's requests as long as all peers are responding. For clarity, in this thesis a single interconnected set of three sharemind servers is referred to as a *3-clique* or *clique*.

Clients are persons or programs with authorization to invoke a specific computation, upload, or retrieve (private) data from the servers. A client may be one of the logical parties or a separate entity, depending on the business case. Multi-party computation has to be initiated concurrently in all of the clique's servers, providing private arguments as the data shares if required by the program. This process is automated by the provided client libraries which can generate additive shares locally and negotiate processes with the clique. Over the years, various client applications have been developed to fulfil various requirements, including a CSV data uploader, statistical analysis tool Rmind, and gateway libraries that intermediate HTTP communication from browser-based clients.

2.2 Parallel Programming Models

Motivation behind parallelisation of algorithms is the achievable speedup from allocating more resources towards a single goal. The speedup of a parallel program is a performance metric for measuring its improvement in running time, calculated as $S = \frac{T_S}{T_P}$, where T_S is the time spent running in serial, and T_P in parallel. Two kinds of speedup are generally distinguished: relative and absolute speedup – relative speedup defines T_S as the time taken by the same parallel program running as a single process, while for absolute speedup, T_S is taken as the running time of the best known serial algorithm counterpart.

Classically, the doctrine for designing parallel algorithms is to maximise the amount of useful CPU cycles per round of communication and synchronisation points. Network communication is considered as a bottleneck that can be minimised, batched, or otherwise optimised with clever parallel algorithm design.

In the case of MPC, we may find ourselves with a certain cryptographic protocol that unavoidably requires frequent and synchronous communication between parties. Despite the cryptographic community actively working on new protocols with improved round efficiency, practice shows that they are always hindered by network bandwidth and latency. This prompts viewing the network channel as a resource – parallel applications distributed across several channels may combine bandwidth allowances of a single node and run several protocols concurrently, resulting in less cumulative waiting and faster completion.

There are no practical limitations to the kinds of parallel programming models that could be adapted to MPC. A clique could be thought of as a regular processor or thread running a task in a parallel program, only spanning over multiple parties. That is as long as the program is deterministic, ensuring identical control flow and ordering of data

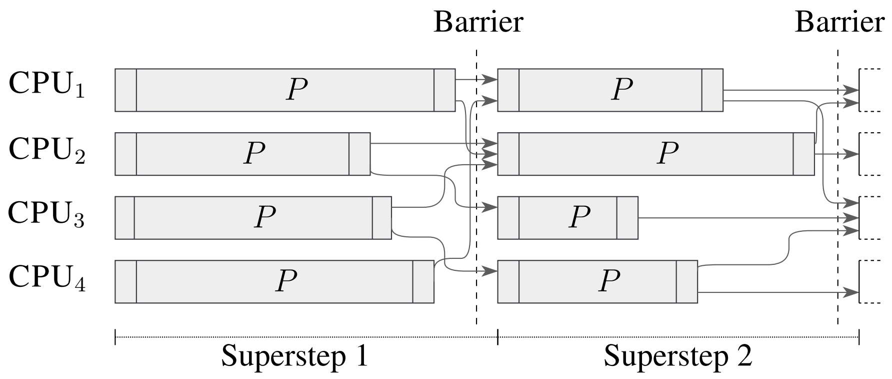


Figure 1. A visual representation of bulk-synchronous parallel supersteps.

during execution. In practice, some programming models are more straightforward to implement, depending on the MPC framework and its flexibility.

In this thesis, we will test our solution based on two parallel programming models: bulk-synchronous parallel and MapReduce. Both models offer data-oriented parallelism and support algorithms for graph and geometric data decomposition respectively. This choice was made due to their simplicity and clear synchronisation barriers.

2.2.1 Bulk-Synchronous Parallel

Bulk-synchronous parallel (BSP), as described by Valiant [33], is an early attempt to describe a model for a parallel computer, much like the von Neumann model is in the sequential context. Contrary to the ubiquitous parallel random-access machine (PRAM) concept, BSP does not share a global address space between processors but describes a distinctive manner of communication and synchronisation.

The high-level operation of a BSP program is shown on figure 1. A set of processors is given a task each to perform independently that within itself has no data dependencies. The execution of the task, indicated on the figure as P, may take varying amounts of time, so processors are synchronised by a global barrier until the last P finishes. During this time, message exchange occurs between processors as defined by the task. The two phases of computation and communication form a so-called *superstep*, several of which may be run one after another to build useful parallel algorithms.

The BSP model's global synchronisation and a clear separation of computation and communication stages alleviate some key stumbling blocks of parallel programming: deadlocks and race-conditions, which usually have to be mitigated by the programmer [27]. Conversely, load balancing task sizes between processors has a pivotal role in the efficiency of algorithms implemented on BSP due to barriers.

The notion of chaining identical *supersteps* makes BSP a useful tool for iterative algorithms. Particularly, big data graph processing systems exercising vertex-centric computation are often built on the principles of BSP. Google's Pregel [24], Apache's Giraph[1], Hama[2], and Spark GraphX[3] are some examples of the kind of graph parallel frameworks.

A vertex-centric (or *think-like-a-vertex*) programming model is computation from the view of each individual vertex – its state and the messages it receives from neighbouring vertices [27]. Each vertex is usually adorned with some local data and the targets of its outgoing edges. A user-defined function (UDF) is invoked for each vertex, first processing incoming messages from the previous iteration to update its state, then emitting messages to other vertices. In practice, vertices may even alter the topology of the graph by altering their outgoing edges and mark themselves as inactive to abstain from further participation or to indicate completion of the iterative algorithm, making the paradigm a powerful tool for certain tasks [24].

In the iterative lockstep nature of the BSP, the method of parallelisation is evident. Vertices are partitioned among processors, with each processor responsible for handling the computational needs of their subset of vertices – state of a vertex is kept in memory throughout the whole run. Clever partitioning of the more connected components of the graph can be seen as an optimisation to exploit locality of messages.

2.2.2 MapReduce

Google shared their success story with the MapReduce model originally in 2004 [17]. It is described as a programmer friendly abstraction to large scale data processing inside a cluster. The programmer's responsibility is to provide implementations of at least two functions:

- *Map* takes as input a single key-value pair $\{k, v\}$, emits zero or more key-value pairs $\{k', v'\}$. The goal of the *map* stage is to prepare raw data into key-value pairs specific to the task.

- *Reduce*, reminiscent of FOLD from functional programming, takes as input a pair of key and all of the emitted values with that key $\{k', v'_1, \ldots, v'_n\}$, and outputs $\{k'', v''\}$. The goal of this function is to aggregate values into meaningful results.

This creates natural parallelisation of the *map* phase, which in turn should refine the raw data into much smaller key-value pairs. Prior to the *reduce* phase, which is also executed in parallel, keys k' need to be sorted, grouped by key, and shuffled among the workers such that each reduce worker receives the full set of values emitted under the same key.

In addition, MapReduce allows for some optional refinement functions to be specified in the pipeline as they may be needed [17]. The first is the *combiner* function, which is

[1]https://giraph.apache.org

[2]https://hama.apache.org

[3]https://spark.apache.org/docs/latest/graphx-programming-guide.html

executed right after map within the same worker. Its general role is to act as an early reducer, aggregating the $\{k, v\}$ pairs that an individual worker has emitted – minimising the amount of emitted values leaves less keys to transfer, sort and distribute, which can become a bottleneck in the system otherwise. The second user-defined function is the *partitioner* that can be used to specify the concrete reduce instance that must process a given key. Left unimplemented, the default behaviour partitions the keys based on their hash and the number of reducers equally. A custom partitioner can be used to manage workload imbalances or to ensure systematic structure of outputs (eg. ordering).

While the API may seem primitive, the virtue of it lies in the supporting facilities and orientedness towards commodity hardware. MapReduce is performant with big data mainly due to its approach of moving the computation to the data in a distributed setting, rather than the other way around. A prerequisite for this is a distributed filesystem running alongside the workers, used for locally storing shards of inputs, results, and the intermediate emissions. A master node, aware of the locations of input shards, may then schedule map tasks on the workers enjoying data locality, and point reducers to retrieve relevant intermediate data from other nodes. Visual reference of the whole MapReduce pipeline is given in figure 2.

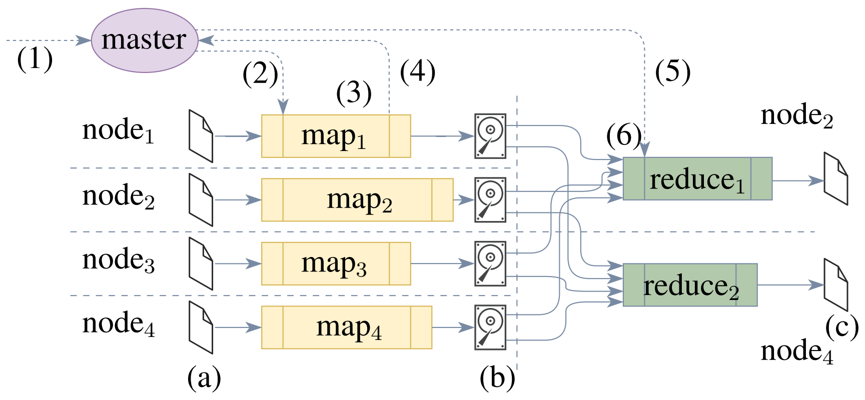


Figure 2. Simplified MapReduce data and task pipeline [17]. (1) The master node receives a job from a client. (2) Four map tasks are scheduled on four nodes. (3) A map task takes its input shard from the distributed filesystem (a), runs the mapper, combiner, and partitioner functions and writes the partitions to disk (b). (4) Map task reports back the written partitions and where to find them. (5) Master schedules reduce tasks on two nodes when all map tasks have finished. (6) Reduce starts by pulling its relevant partitions from other nodes, merging and aggregating them by key, writing the results back into the filesystem (c) in the end.

Comparing MapReduce with the BSP model, it can be noted that they share similarities such as barriers between stages followed by an exchange of messages. Indeed, Pace [29] has shown in his analysis that BSP can be modelled in MapReduce without any asymptotic penalties. He points out that the reverse may come with its hardships, as an iterated MapReduce algorithm couldn't access any other data that isn't communicated by the previous round. Generally, iterative algorithms in MapReduce is also frowned upon, as intermediate and iteration results are frequently written to disk, as opposed to the more efficient in-memory state keeping.

2.3 Cloud-Native Applications

Cloud computing is without a doubt the standard for Internet service deployments in the foreseeable future. Various service models transfer management of resources, like storage and networking, to the responsibility of the cloud provider [7]. This has proven useful not only for streamlining deployment, but also optimising cost by only paying for what is used. An appealing aspect of cloud computing is the seemingly endless resources on demand, providing scaling opportunities with no upfront cost.

Multiple design patterns have been adopted for developing cloud applications with scalability in mind [15]. Microservices and serverless computing are employed to split traditional monolithic services into self-contained and independently scaling components, responsible for a specific subtask. Coincidentally, microservices are central to a newer concept called *cloud-native applications*. The term *cloud-native* has many interpretations, but Kratzke's and Quint's survey on the subject defines cloud-native applications as distributed, elastic and horizontal scalable systems of microservices with the amount of stateful components kept to a minimum [20]. It goes further to specify that each deployment unit (eg. microservice) exhibits cloud-focused design patterns and is operated using a self-service platform.

Popularising this concept is the mission of the Cloud Native Computing Foundation (CNCF) [18]. It collates and promotes open-source projects – purpose specific components that build on the ideology and aid adoption of cloud-native best practices. Virtues that the CNCF seeks in its fostered projects include support for public, private, and hybrid clouds and cloud vendor agnosticity, meaning that cloud-native infrastructures should be easily deployed or migrated to any kind of environment.

The Kubernetes container management platform[4] is tightly associated with the CNCF, and is central to many of its projects. A container management platform by itself can be thought of as a middle ground between *infrastructure-* and *platform-as-a-service* cloud deployment models. On one hand, it provides users the means to host containerised workloads on clusters without any assumptions for their inner workings. On the other hand, it provides a vast API for common interfacing and administrative tasks generally offered by PaaS, like scaling, load-balancing, storage, and stateful components among others. The API itself is declarative, meaning that the Kubernetes control plane continuously

[4]`https://kubernetes.io/`

tries to maintain a *desired state* of each cluster resource. The Helm[5] package manager is a good example of managing complex Kubernetes deployments as sets of configuration declarations, sharing similarities with Infrastructure-as-Code.

[5]https://helm.sh/

3 Related Work

In this section we give an overview of relevant literature and previously made solutions. First, in section 3.1, a recently open-sourced project that brings cloud-native MPC to market is dissected. Following in sections 3.2 and 3.3 are previous efforts to marry parallel programming with MPC using MapReduce and graph parallel algorithms respectively. Finally, two solutions that have previously employed parallel programming with SHAREMIND, and shown its effectiveness, are analysed in 3.4 and 3.5.

3.1 Carbyne Stack

In a recent effort to pioneer cloud-native MPC, engineers at Bosch Research have launched its open-source platform named Carbyne Stack [14]. The project contributes scaffolding and utility services to run two-party MPC effectively across two virtual cloud providers acting as parties. The MP-SPDZ framework [19] is used internally as the MPC protocol runtime, specifically its two-party protocol set.

The services are laid out as follows:

- *Amphora* service provides an interface to store and receive secret shared input and output data. Data is kept in *object storage* (similar to Amazon's S3) to be consumed by the MPC program or client.
- *Castor* facilitates storage for correlated randomness – a necessity for the chosen protocol to execute certain operations, like multiplication, efficiently. The randomness is pre-generated by the client ahead of time.
- *Ephemeral* is the compute service itself, further split into three components:
    - *Ephemeral serverless* fetches inputs from *Amphora*, correlated randomness from *Castor*, and invokes MP-SPDZ on the requested MPC program. Implemented as a serverless service, it is the entry point for the client to instantiate the computation.
    - *Discovery* service keeps track of the state of running computations and coordinates the creation and teardown of inter-cluster networking.
    - *Network controller* is defined as a custom Kubernetes operator, responsible for monitoring the cluster for networking requests of Ephemeral instances. It dynamically provisions service mesh routing for clique instances to communicate to its counterparts in the other cluster as per the instructions from the discovery service.

Carbyne Stack provides valuable insight to architecting cloud native MPC deployments. Multi-party computation aligns well with the nature of serverless computing as long as intra-clique discovery and communication is made possible. Carbyne Stack resolves this by networking parties with the aid of a discovery service and a multicluster service mesh. A similar solution is implemented for this thesis.

Currently, the Carbyne Stack provides no access control for MPC programs or for reading the secret shared data from Amphora, however the latter is planned to be implemented[6]. For the former, to be practically viable, parties need to be able to enforce which programs can run and process private inputs to prevent arbitrary scripts from leaking information. Both are measures the SHAREMIND platform implements by design [8][7].

The platform is not an effort to parallelise MPC computation, however it may be easily adapted. This could mean writing an alternative client application, that invokes multiple Ephemeral calls at once and synchronises the execution as per the task dependency graph of the parallel MPC program. The cluster's inner workings would continue working in the same way as before, however due to the serverless function not being able to retain state over multiple runs, the client may need to do more work keeping track of the intermediate data and supply each clique with the object identifiers that contain their next inputs.

3.2 MapReduce with secure multi-party computation

Volgushev, et al. dedicate a line of research towards practical means to enable secure cross-organisation data analysis with MPC on big data volumes [34–36]. They make the observation that no matter the cyptographic advancements, overhead of MPC is still making it unviable for real-world data processing tasks. The authors propose an intelligent analytics engine, interleaving local cleartext data processing in MapReduce with consequently smaller MPC tasks. The principle of their Conclave engine is well rounded: "do as little as possible and as much as necessary under MPC" [36:4].

Conclave is centred around its DSL and compiler, allowing the user to specify relational queries in an environment where multiple parties hold the different relations. Rather than employing an MPC framework directly, it formulates a task dependency graph of the specified queries, detecting the steps that can be locally pre-processed and reduced. The compiler *pushes* the so-called *MPC frontier* to a minimum, meaning that it delegates any possible processing to the parties owning the data both before and after the inevitable MPC region.

The cleartext processing tasks are executed in the PySpark implementation of MapReduce, exploiting data-parallelism. Tasks left for secure computation will be generated

[6]`https://github.com/carbynestack/amphora/blob/23fb76829bb366ce37fbdde482854be6f394c546/amphora-service/README.md#authentication-and-authorization`

[7]Access control of MPC programs: `https://docs.sharemind.cyber.ee/2022.03/installation/release-notes#encrypted-computing-engine-4`

code in the MPC backend of choice, such as Obliv-C or SHAREMIND, while not compromising privacy guarantees. In reported metrics, the combination of MapReduce and SHAREMIND shows convincing results – with their example queries, Conclave was able to process a billion record relation in 20 minutes, whereas running the same example in full in MPC could just finish a maximum of 10k records at 10 minutes.

Even though the MPC regions in the work are not run in parallel, there are takeaways in the context of this thesis. First, it is important to note that MapReduce-like processing pipelines also make sense within secure computation. This may be the case when the details of the analysis are not yet known when the shares are collected, when the stored shares are expected to be used for multiple different analyses, or simply when the devices submitting shares are not suitable to handle heavy preprocessing. Secondly, the Conclave system may benefit from the main outcome of this thesis, parallelising MPC in addition to its already parallel cleartext processing.

3.3 Privacy-Preserving Graph-Parallel Computation

Efficient secure computation on graph-structured data is a coveted milestone due to its possible applications in privacy-preserving machine learning and data mining [28]. Multiple approaches have been proposed in recent years to speed up graph processing in MPC, utilising a variety of specialised protocols, parallelism, and general optimizations.

The GraphSC framework was devised by Nayak, et al. [28], exposing a vertex-centric programming abstraction that is compiled and executed as a garbled circuit. Circuits are generated such that their garbling and evaluating can be well parallelized – both parties contribute n processors for the task, each acting as the garbler (party a) or evaluator (party b) of a subcircuit. Secret shares representing the graph and its augmentations are kept in shared memory of each party (one multi-core processor or a cluster, communicating via a message passing interface), with the circuit dictating oblivious RAM operations (masking of access patterns). With this method of creating circuits encoding Pregel-like GATHER, SCATTER, and APPLY operations, the authors show a near linear speedup in relation to the number of processors on implementations of PageRank, matrix factorization, and histogram calculation.

Mazloom, et al. [25, 26] demonstrate two approaches similar to GraphSC for also parallelising vertex-centric graph algorithms. In their first instalment [25], they introduce the OblivGraph protocol: sharing the general construction of GraphSC, but with differential privacy instead of oblivious RAM access, relaxing the security to better efficiency. With differential privacy, the structure of the graph is obscured with dummy edges and oblivious shuffling of edges to hide the real structure of the graph. Followed up by the second article [26], they move away from garbled circuits to propose a custom four-party protocol exercising differential privacy. All in all, the final version performs a matrix vectorisation 1872 times faster than GraphSC and 288 times faster than its predecessor, OblivGraph.

Anagreh, et al. [2–5] have pursued a more pragmatic avenue of efficient MPC graph algorithms. They have proposed privacy-preserving algorithms for well-known graph problems such as finding Minimum Spanning Trees [4], Single-Source Shortest Paths [3], Single-Source Shortest Distances (SSSD) and All-Pairs Shortest Distance [2]. The works target SHAREMIND and design the algorithms around vectorised operations to deliver a type of low granularity communication parallelism that SIMD offers in the framework. Authors present thorough benchmarks for their implementations, which provides a reference for comparison for the vertex-centric SSSD also implemented in this thesis on the vertex-centric model.

3.4 Tax Fraud Detection System Implementation

In 2014, a privacy preserving system based on SHAREMIND for automated anomaly detection in value-added tax declarations was proposed to the Estonian Tax and Customs Board (MTA) [10]. Employing MPC, the primary value proposition was to withhold plain transaction data from the MTA, while still enabling discrepancy analysis of sale and purchase transactions between local companies. The interest for this was sparked by a law requiring companies to file their monthly transactions, but it was promptly vetoed by the president as unconstitutional under the concern of privacy.

Non-colluding parties, envisioned hosting the computation, all had some interest in the functioning of such a system. For one, MTA as the main stakeholder partakes in the computation. So does the traders association, who represents the taxpayers in effort to maintain privacy of their tax filings. Finally, the Information Systems and Registers Centre, which reports to the Ministry of Justice, and is responsible for safety of critical information systems in Estonia.

Among its requirements, the system had to perform analysis on a large number of filed declarations within reasonable time. This influenced the system's design process to consider parallel processing. The structure of the system is as follows:

1. During the *upload* phase, companies upload their secret shares of fields in their declaration forms to each of the MPC parties' servers. The receiving end runs a simple MPC upload program separating the sales' and purchases' amounts and persists them, along with the identifiers of the associated parties, into secret share databases. At this stage, parallelism is duly considered by distributing the companies' data between n aggregation queues.

2. The *aggregation* phase runs over n parallel cliques, with each clique processing the data partition prepared for it in the previous phase. For each company, the sum of its transaction totals with each partner is appended to a unified aggregation results table.

3. *Aggregation finalisation* phase pairs up the sales of one company to the purchases from that company by each counterpart.

4. Final *risk calculation* phase runs a comparison of those pairs and outputs companies with any significant deviations from the expected balance as deceitful.

It can be seen how the phases follow a MapReduce pipeline pattern. The upload phase operates on individual declarations, much like the operation of the map stage of MapReduce. Initial aggregation takes the role of the combiner, summing up the sale and purchase totals of company pairs before passing them on. Final aggregation and risk calculation make up the reduce phase – taking the partner pair as key and their balances as values to then compare them. Even though the reduce stage in this case runs on a single process, a modification would be possible to make it parallel as well. This could be done by sorting and partitioning the keys so that each reduce instance gets a subset and all the values belonging to them.

The initial paper [10] benchmarked the system in a local environment – three physical machines connected by a 1 Gbit/s link and sub-millisecond latencies. Speedup, albeit insignificant, is reported in the parallel phase even if all the processes communicate over the same channel. The same system is revisited in a later report focusing on its cloud deployment and further efficiency improvements [11]. There, 80 parallel SHAREMIND cliques were split over four Amazon EC2 virtual machines per party. Although there are no benchmarks for assessing the number of instances' effect on the system performance, which is why it is not immediately apparent as to how much parallel aggregation improves execution time. Rather the work expresses the importance of communication between cloud regions, showing dramatic increases in runtime and monetary cost as parties are separated into different geographic regions. Results on the cluster do show improvements when compared to the original article nonetheless, confirming the motivation of this thesis of parallelising MPC programs.

Relevant to this thesis, the latter report expresses the need for further effort in bringing MPC closer to cloud computing. This is justified with the cloud's elasticity in terms of resource provisioning, and no up-front costs as opposed to expensive on-premises hardware. For this, authors note that automatic service provisioning tools need to be developed to aid adoption and reduce the administrative know-how of future parties that want to partake in some MPC application.

3.5 Anonymous Messaging Implementation

The second parallelisation effort of an application on the SHAREMIND platform has been an MPC supported anonymous messaging service by Alexopoulos, et al. [1] named MCMix. Its underlying protocol achieves anonymity with simulation-based security guarantees against a global adversary. That is to say that any single entity in control of the network at each point of the protocol's interaction with its components can not learn any metadata of the occurring conversations other than that a specific client is participating in the system.

The gist of the proposed functionality involves two sub-protocols that are executed at regular intervals: *dialling* and *conversation*. Dialling allows an user to establish a random rendezvouz point with the intended conversation partner and notify the recipient of the request. The two users will then commence in exchanging short messages by submitting them to be left on the set point as dead drops, which the next iteration of the conversation program will securely pick up and deliver to the recipient. MPC programs for either of the steps are constructed to hide the conversating partner pairs and their messages.

Benchmarking results of the protocols' implementation on SHAREMIND show that the system can accommodate even realistic amounts of concurrent users within reasonable time. With a 1 Gbit/s bandwidth between each MPC party, the dialling and conversation protocols exhibit near identical completion times, being able process 500k users in 300 seconds. With this in mind, the authors express their concern regarding performance of just the conversation protocol, as it is the more latency critical component, to which they propose a parallel approach. This was a non-trivial task as the messages of users had to be partitioned over n cliques, however due to implementation details, the message had to be able to reach an output index of some other clique. The *ad hoc* solution was to split the conversation program into three sub-programs, in between which data is exchanged among cliques. As a result of the parallelisation, the authors report a small overhead with just two cliques, but project the speedup to allow for servicing significantly more clients with four or more parallel instances.

Note, that the parallel conversation protocol could be depicted as a BSP program. Three distinct sub-programs of it are run in parallel and synchronised for message exchange, effectively forming three supersteps. After it has run, a new iteration will begin; reading in the new diallings and incoming messages.

Internally, synchronisation and intermediate data exchange was coordinated by a custom controller application working as a gateway in front of each SHAREMIND instance[8]. From a client's perspective, an application specific controller allows for transparent execution of multiple pipelined MPC tasks. This execution model is considered, but not necessarily reused for the practical solution of this thesis as will be discussed shortly.

[8]`https://github.com/druid/mcmix-benchmark`

4 High Level Requirements

In this section we address the requirements for components in the parallel MPC framework. Starting by introducing relevant assumptions for the underlying MPC platform to better understand the system incorporating it. The assumptions are based on SHAREMIND MPC, as it is the chosen framework for this thesis. Then we lay out how parallelism is to be achieved and how it should be orchestrated. Finally, we detail the specifics of running MPC in a cluster.

4.1 Assumptions

General purpose MPC systems like SHAREMIND execute special programs containing business logic written by the programmer as some privacy-preserving computation task. The program defines the sequence of instructions as usual, but makes a distinction between public and private values. Public being the variables for which the values are known to the computation party in clear, used to determine the control flow or support the algorithm in other ways. Private values are protected from the view of individual parties with cryptographic means – the procedure for operating on these values is specified by the protocol they adhere to. Protocols may need the parties to engage in communication to compute on the values and it is expected that in practice, network constraints quickly become the bottleneck of performance. As the program's code is ready for deployment, each party should examine it for any privacy-compromising logic, compile it, and upload it to their system for it to be accessed and executed by the servers when called.

We view MPC as following a client-server model of operation, in which there are multiple servers making up the clique. In order to start a multi-party computation, the servers need to have previously established communication channels and be listening for incoming tasks. The client orders the execution of a program by its name from each of the computing parties simultaneously, providing public and private inputs to each party, and receiving published results at its completion. Computation is a non-interruptible process that happens in unison within a clique – sharing the same control-flow to not fall out of sync w.r.t. the computation on private values.

Some network configuration is expected to enable the communication of the components. Intra-clique networking depends on all servers being able to reach their counterparts over TCP/IP. Due to the non-collusion of parties, the protocol, in most cases, operates over the public Internet, meaning that the corresponding routing and firewall entries have to be in place. The same applies for client-clique interaction, which in case of SHAREMIND utilises the same endpoint on servers.

For decomposing tasks, we make the assumption that a task can feed intermediate data to subsequent tasks. This can be approached in multiple ways, but at the very least the MPC framework has to be capable of holding state outside of a single task's lifecycle. In the case of SHAREMIND, persisting secret shares and public values between program invocations within the cluster is possible – e.g. supporting use case models such as analysis of multiple data owners' inputs or running different queries on the same data would not be possible otherwise. The current implementation accomplishes this by exposing two different interfaces for storage of secret-shared values: storing tabular data in the HDF5 format on the filesystem or in-memory key-value stores backed by Redis. Either is a fit for carrying over specific data to subsequent stages. Using disk based storage to communicate intermediate data should not affect the performance of the overall task, as the intra-clique protocol communication remains as the dominant source of overhead.

Although not extensively explored in this thesis, the authentication, authorization, and access control for programs and data is important to note nonetheless. Control over the pre approved programs that the clique supports is handled as mentioned in the first paragraph, meaning that clients can not run arbitrary computation. All parties have to agree on the clients and their public keys that are given authorization to execute certain MPC programs, and additionally, which potentially private databases the programs executed by them can read or modify. This is to deter clients in a multiple role scenario that are not acting in good faith from trying to declassify information not meant for them.

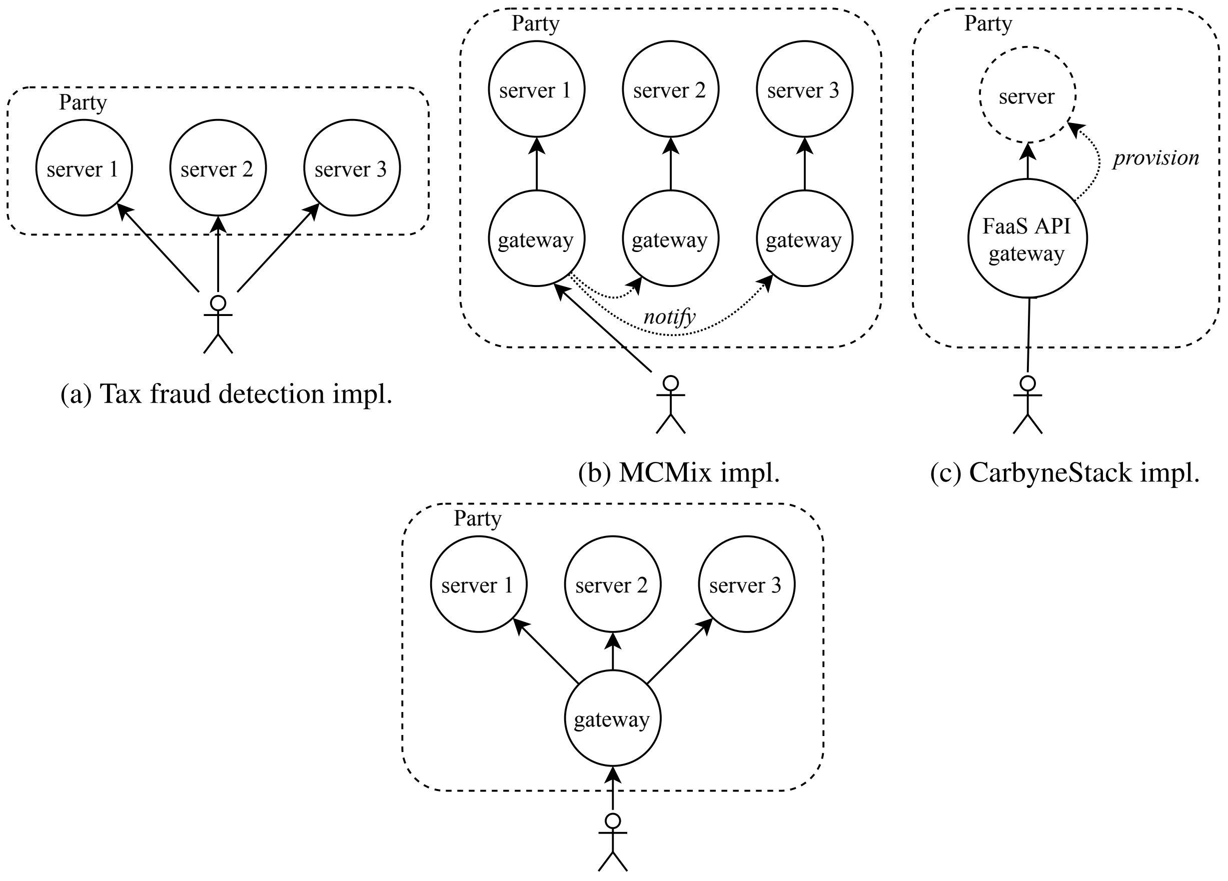
4.2 Parallelisation

Several alterations to a regular MPC workflow and deployment model have to be made when considering a parallelism-offering system. This involves establishing requirements regarding how parallel regions are defined, client's interaction with the parallel system, and gathering-scattering of intermediate data.

In this work, the execution model and protocols of the MPC platform are to remain untouched. Rather, parallelism is approached in a similar fashion to the previous work by Alexopoulos, et. al [1] and Bogdanov, et. al [10]. The measure of granularity is determined by a single MPC program – this can be a program that is expected to be run as a data-parallel region, contributing a section of a larger privacy preserving computation task, henceforth called a *sub-task*.

Having multiple MPC servers per party presents the issue that a client program[9] would need to keep track of each individual instance by their public network address, authenticate to each one separately, maintain multiple open connections, and synchronise the execution of the sub-tasks. This kind of client-clique communication was used in the tax fraud detection prototype [11] as illustrated on figure 3a, but is undesirable in our case as it limits exploiting some elasticity properties further discussed in section 4.3. Two alternative approaches could be considered: either employ serverless (FaaS) methodologies to run MPC servers on demand (similar to Carbyne Stack [14] as shown on

[9]Here and onwards, the application that communicates with the MPC system is referred to as just a *client* – not to be confused with a person interacting with the system.



(a) Tax fraud detection impl.

(b) MCMix impl.

(c) CarbyneStack impl.

(d) Proposed solution

Figure 3. Different methods for clients to address parallel MPC cliques: solutions from previous work and the planned architecture.

figure 3c), or develop a single point of entry, in the form of a gateway, to the cluster (as was implemented in MCMix [1], figure 3b). To minimise development costs, the latter option is preferred.

Such a gateway relieves some burden on the client. First it hides individual MPC servers from the client – this is good as a co-located service can have a better overview of the health of the servers and can route requests locally, along with requiring just a single authentication from the client. Secondly, it provides a synchronisation point for parallel regions of tasks. As can be seen by comparing figures 3b and 3d, the difference with the MCMix solution is the 1-to-n as opposed to n-to-n relation between cliques and gateways. This additional complexity of multiple gateways is able to be left out without sacrificing any of the functionality.

The gateway application should not be specific to the parallel algorithm for minimising the amount of custom components, i.e the gateway does not execute sub-tasks or scatter data among cliques on its own initiative. Ideally, only the MPC program and client should be aware of the task and its composing sub-tasks. This shifts keeping track of the execution flow to the client, but in turn offers more flexibility and saves deployment effort for different parallel models. Parallel regions must be accommodated by prior provisioning of as many parallel cliques as the computation requires. The degree of parallelism of a sub-task is to be decided by the client and passed along as an argument of the process negotiation to each party's gateway.

The client should not mediate data in between parallel regions, instead a distributed storage solution should be used for communication of intermediate shares or public values between one party's cliques. This is to save bandwidth and minimise data transfer time, but also support MPC use cases where the client initiating the computation should not be able to learn the data being analysed. An alternative to the storage based solution would be to delegate the transport of communicated values to the gateways. This was the approach of MCMix [1], in which all intermediate values were sent back as computation results to a task specific gateway so that they could be gathered and given to the following sub-task as arguments. As discussed earlier, a general parallel gateway implementation is preferred to one that is purpose-made to handle a specific application. Choosing the proposed solution also comes with an accompanying benefit, that of failure tolerance in multiple-stage tasks – if a sub-task fails, then it can be restarted without data loss, as the communicated values from the preceding sub-task are persisted on disk.

The gateway has to allow the client to invoke parallel MPC sub-tasks with inputs and receive outputs when applicable. An exception of passing input to sub-tasks can be meta-arguments. As the MPC programs have to be written in consideration of their parallel environment, it may need to know from which files or index ranges to consume data from, and how to prepare its intermediate results for the next sub-task. Good examples of this are data-parallel algorithms and the gather-scatter behaviour of the message-passing paradigm. Parallel regions spawned to handle a partition of a large input dataset have to know at minimum their index and the amount of siblings in the parallel region.

To reiterate,

1. data owner(s) upload their dataset(s) to the clusters;

2. a client orders the execution of a sub-task from the gateway, specifying the degree of parallelism n;

3. the gateway delegates the order to its servers, supplementing their arguments with the unique instance's index $i \in \{0, \ldots, n-1\}$ that the program uses to select the partition of data to process (in the case of a data-parallel algorithm);

4. if results are published from any of the instances, they are concatenated and sent to the client;

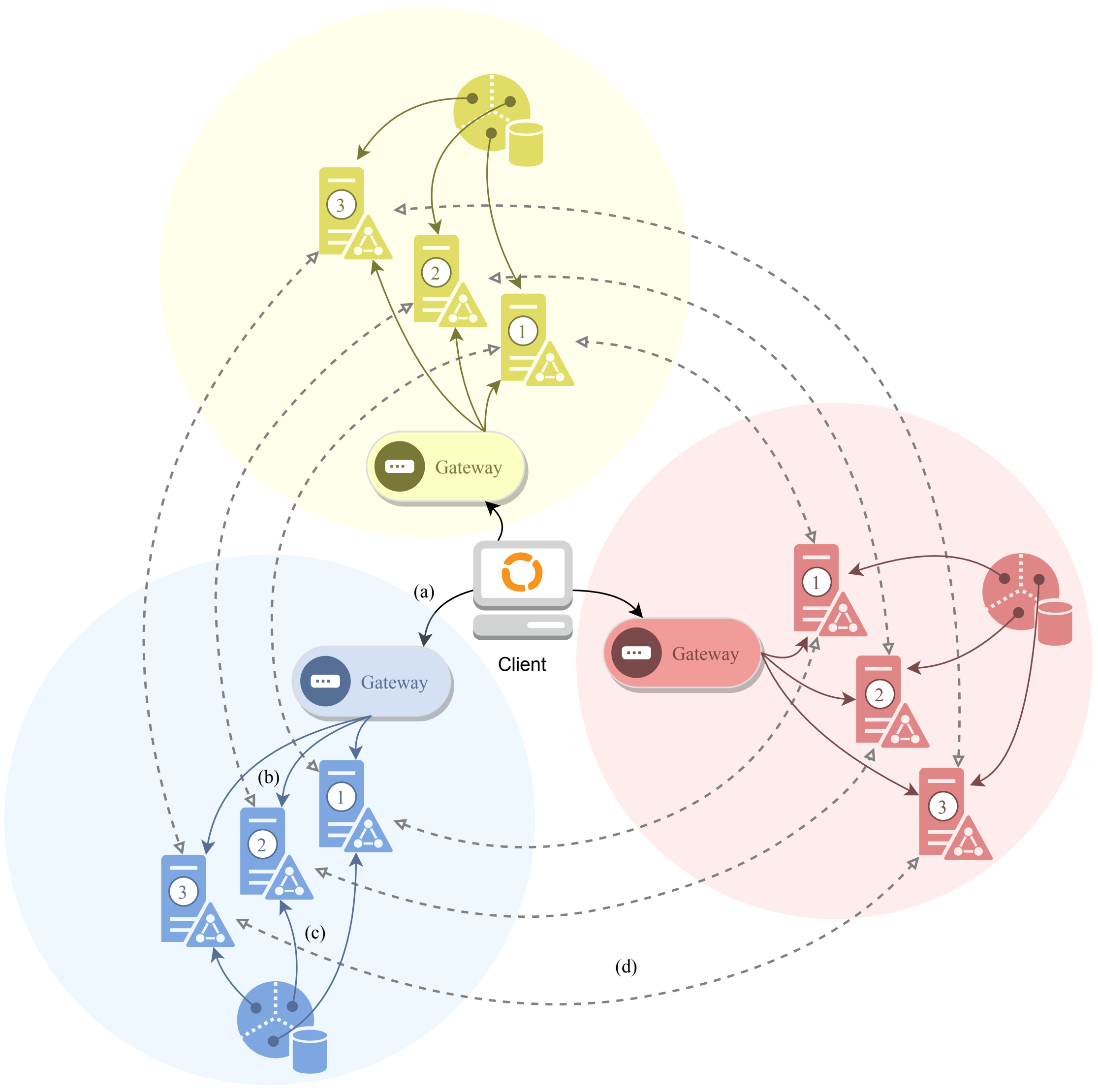


Figure 4. Visual representation of a parallel three-party MPC computation with three cliques, with colours showing the domain of each party. (a) The client negotiates a parallel MPC sub-task, requesting it to be run on three instances. (b) The gateway relays the requests to the individual MPC servers which are numbered by their clique ID. (c) The servers fetch their specific partition of data from the shared filesystem. (d) All three cliques engage in multi-party computation and run the distributed protocol over all parties.

5. if there are any more sub-tasks to perform, repeat item 2.

A simplified diagram of steps 2 and 3 is shown on figure 4.

4.2.1 Goals

Motivation behind parallelisation remains the same as is the case for parallel computing in general: solely to accelerate execution of tasks. The technical solution presented in this thesis has to provide speedup of a task as more cliques are concurrently engaged in computation. The ultimate goal of parallelisation of MPC is spreading of network workload over multiple nodes. In practice, this should offer improved throughput of multi-party protocol communication as it would 1) combine bandwidth quotas enforced on individual nodes and distribute load over several network interface controllers and 2) reduce collective waiting time of blocking protocol steps that stems from latency.

Quality assurance is performed by analysis of metrics extracted from clusters during the benchmarks in section 5.3. By observing the network and filesystem usage, it is assessed whether the devised system and programming models exploit the proposed environment to their benefit. Processor utilisation is additionally monitored to verify that the computation is not CPU-bound.

The concept and usage of the devised system on any subsequent MPC applications needing to be parallelised should be easy to grasp for a programmer that is familiar with the SECREC language based on the given examples. It should be clear how to implement new privacy-preserving parallel algorithms while being aware of the privacy implications caused by those design decisions – pointers in both of these areas are discussed in the following sections based on the implementations of MapReduce and BSP. While not mutually exclusive, the SIMD execution model used in SHAREMIND is unrelated to the much larger granularity parallelism considered here. It is encouraged to still practise SIMD inside the parallel programs as its performance benefits remain relevant.

4.2.2 Programming Models

The prospect for employing MPC on large datasets is still problematic due to the overhead of protocols, however this thesis argues that it could be remedied with appropriate parallel algorithm design. To an extent, inspiration for programming models can be taken from big data solutions that share resemblance with the style of data analysis often needed from MPC. Further, those models are easy to grasp or already familiar to many, not to mention the root problem of increased data volumes that these solve are reminiscent to what is hindering MPC.

In this thesis, provided examples focus on data-centric decomposition of tasks – a simple approach for utilising parallelism where a significant part of the computation can be done in a secluded manner before requiring inter-process communication. Programming models of MapReduce and bulk-synchronous parallel are taken as well-known illustrative examples and their general operation is mimicked to evaluate the system.

MapReduce is characterised by a highly synchronous three-step data processing flow that allows for a high degree of parallelism and can be expressed with tools provided by SHAREMIND. Two of its three stages, *map* and *reduce*, can be spread across virtually any number of processes. Either can accommodate processing that ultimately makes up the significant part of computation.

User defined combiner and partitioner is not explored explicitly, but the map and partition programs can be extended accordingly to include this logic if needed. By default, partitions are split equally between the reducers, giving the first instance the smallest keys w.r.t their ordering. Keys throughout the processing pipeline should be integer values in the interest of efficiency – values on the other hand can be integers, floating point numbers, strings, or vectors. Tokenization can be used as local pre and post processing to accommodate string keys, if so desired.

Throughout the privacy-preserving MapReduce application, no input, intermediate, or output keys and values should leak. Due to partitioning it is however difficult to hide the amount of key-value pairs at each stage. As it requires sorting and grouping emitted values by keys, a curious party profiling the process could learn how many key-value pairs were emitted for certain keys by assuming the keys based on their order.

Proposed general operation of the stages are given as MPC specific pseudocode in algorithms 2, 3, and 4. *Map* operates only on individual data entries, thus its parallelisation is trivial. Note that the `map` user-defined function (UDF) and step 4 of algorithm 2 would preferably be applied to vectors instead of individual values if SIMD optimizations are in place. Partitioning follows (algorithm 3), which is a single-process sub-task in order to ensure that the *reduce* stage process receives all key-value pairs emitted by *map* with the same key. While the partitioning in cleartext MapReduce is done within the map task in parallel, the same is not trivially accomplished here if keys are to be kept private throughout the process – partitions should be prepared in such manner that the subsequent parallel reduce instances would be able to differentiate partitions' keys that only they should process. One possibility would be declassifying the result of an oblivious hashing [32] of the partition's keys so that equal keys could be identified, but due to added complexity this is not considered.

Notable steps in partition are 6 and 7, the purpose of which is to find the indices on which it is possible to split partitions. Step 6 sorts the emitted key-value pairs using privacy-preserving sorting, so as to not reveal the keys. Step 7 checks non-equality of each key with the next, only revealing the indices at which there is a change in the key value, saving the array of truth-values into variable *splits*. This is consequently the only necessary leakage. Split values are used to partition reduce databases at the correct

Algorithm 2: MPC MapReduce *map* program

```
Input: UDF map($[\![k]\!]$, $[\![v]\!]$); Clique index $i$; database $db_{in,i}$ with two columns,
       $\overrightarrow{[\![k]\!]}$ and $\overrightarrow{[\![v]\!]}$, corresponding to keys and values for the $i$-th map clique.
Result: Database $db_{emit,i}$ contains columns $\overrightarrow{[\![k']\!]}$ and $\overrightarrow{[\![v']\!]}$.
1 $\overrightarrow{[\![k]\!]} \leftarrow$ readColumn$(db_{in,i}, 0)$; $\overrightarrow{[\![v]\!]} \leftarrow$ readColumn$(db_{in,i}, 1)$
2 $\overrightarrow{[\![k']\!]} \leftarrow \emptyset$; $\overrightarrow{[\![v']\!]} \leftarrow \emptyset$
3 foreach $[\![k]\!], [\![v]\!]$ in $\overrightarrow{[\![k]\!]}, \overrightarrow{[\![v]\!]}$ do
4     $[\![k']\!], [\![v']\!] \leftarrow$ map($[\![k]\!]$, $[\![v]\!]$)
5     $\overrightarrow{[\![k']\!]}$.push($[\![k']\!]$); $\overrightarrow{[\![v']\!]}$.push($[\![v']\!]$)
6 writeColumn$(db_{emit,i}, 0, \overrightarrow{[\![k']\!]})$; writeColumn$(db_{emit,i}, 1, \overrightarrow{[\![v']\!]})$
```

Algorithm 3: MPC MapReduce *partition* program

```
Input: Amount of map cliques $n$; amount of reduce cliques $m$; databases of map
       outputs $db_{emit,i}$ for all $0 \leq i < n$.
Result: Database $db_{part,i}$ contains columns $\overrightarrow{[\![k']\!]}$ and $\overrightarrow{[\![v']\!]}$ sorted by $[\![k']\!]$, such that
        for all $i$ and $j$ ($0 \leq i, j < m$ and $i \neq j$), $db_{part,i}$ shares no keys with
        $db_{part,j}$.
1 $\overrightarrow{[\![k']\!]} \leftarrow \emptyset$; $\overrightarrow{[\![v']\!]} \leftarrow \emptyset$
2 foreach $i \leftarrow 0 \ldots n-1$ do
3     $\overrightarrow{[\![k']\!]}$.extend(readColumn($db_{emit,i}, 0$))
4     $\overrightarrow{[\![v']\!]}$.extend(readColumn($db_{emit,i}, 1$))
5 $l \leftarrow \overrightarrow{[\![k']\!]}$.size()
6 securely sort vectors as matrix $\left[\overrightarrow{[\![k']\!]}, \overrightarrow{[\![v']\!]}\right]$ by the first column
7 $split \leftarrow$ declassify($\overrightarrow{[\![k']\!]} \neq$ concat($\overrightarrow{[\![k']\!]}[0], \overrightarrow{[\![k']\!]}[0:l-1]$))
8 $step \leftarrow \lfloor l/m \rfloor$
9 foreach $i \leftarrow 0 \ldots m-1$ do
10    if $i = m-1$ then
11        $t \leftarrow \overrightarrow{[\![k']\!]}$.size()
12    else
13        $t \leftarrow$ index($split[step:]$, true)
14    writeColumn($db_{part,i}$, $\overrightarrow{[\![k']\!]}[0:t]$, 0); $\overrightarrow{[\![k']\!]} \leftarrow \overrightarrow{[\![k']\!]}[t:]$
15    writeColumn($db_{part,i}$, $\overrightarrow{[\![v']\!]}[0:t]$, 1); $\overrightarrow{[\![v']\!]} \leftarrow \overrightarrow{[\![v']\!]}[t:]$
```

Algorithm 4: MPC MapReduce *reduce* program

```
Input: UDF reduce($[\![k]\!]$, $\overrightarrow{[\![v]\!]}$); clique index $i$; sorted database $db_{part,i}$.
Result: Database $db_{out,i}$ contains columns $\overrightarrow{[\![k'']\!]}$ and $\overrightarrow{[\![v'']\!]}$.
1  $\overrightarrow{[\![k']\!]} \leftarrow$ readColumn$(db_{part,i}, 0)$; $\overrightarrow{[\![v']\!]} \leftarrow$ readColumn$(db_{part,i}, 1)$
2  $\overrightarrow{[\![k'']\!]} \leftarrow \emptyset$; $\overrightarrow{[\![v'']\!]} \leftarrow \emptyset$
3  $l \leftarrow \overrightarrow{[\![k']\!]}$.size()
4  $split \leftarrow$ declassify($\overrightarrow{[\![k']\!]} \neq$ concat($\overrightarrow{[\![k']\!]}[0], \overrightarrow{[\![k']\!]}[0 : l - 1]$))
5  $unique \leftarrow split$.count(true)
6  foreach $i \leftarrow 0 \ldots unique - 1$ do
7      if $i = unique - 1$ then
8          $t \leftarrow \overrightarrow{[\![k']\!]}$.size()
9      else
10         $t \leftarrow$ index($split$, true)
11     $[\![k'']\!], [\![v'']\!] \leftarrow$ reduce($\overrightarrow{[\![k']\!]}[0]$, $\overrightarrow{[\![v']\!]}[: t]$)
12     $\overrightarrow{[\![k']\!]} \leftarrow \overrightarrow{[\![k']\!]}[t :]$; $\overrightarrow{[\![v']\!]} \leftarrow \overrightarrow{[\![v']\!]}[t :]$
13     $\overrightarrow{[\![k'']\!]}$.push($[\![k'']\!]$); $\overrightarrow{[\![v'']\!]}$.push($[\![v'']\!]$)
14 writeColumn$(db_{out,i}, 0, \overrightarrow{[\![k'']\!]})$; writeColumn$(db_{out,i}, 1, \overrightarrow{[\![v'']\!]})$
```

indices.

The *reduce* stage can then run aggregational operations on its partitions' values per key and output the results. This is shown in algorithm 4. The same splitting logic is employed as in algorithm 3, now to find the slices on which to run the reduce UDF; sorting is not required as the partition sub-task has already sorted the partitions by key.

For clarification, the only virtues the MPC example lends from the original MapReduce implementation is the general programming model. By conjecture, the distributed in-place data processing as exercised by real MapReduce implementations would not benefit this system by much, since the data volumes are expected to be inherently smaller.

Bulk-synchronous parallel model is used to show iterative parallel computation with MPC. It allows for interleaving computation and communication with clear synchronisation barriers. As is apparent from the name of the model, a set of computations takes place in isolation before a collective synchronised communication. This naturally fits the communication requirements set in section 4.2.

Proposed general construction of BSP in MPC is given as *computation* and *communication* abstractions of programs – the steps for the former can be seen in algorithm 5, algorithm for the latter is omitted as it is analogous to the *partition* program of MPC

Algorithm 5: MPC BSP *computation* program

```
Input: UDFs $\mathsf{newState}([\![k]\!], [\![state]\!], \overrightarrow{[\![msg_k]\!]})$, $\mathsf{newMessages}([\![k]\!], [\![state]\!])$;
       clique index $i$; database of messages $db_{msg,i}$; database of states $db_{state,i}$.
Result: Database $db_{out,i}$ contains messages by key for the next iteration;
        database $db_{state,i}$ contains updated states.
1  $\overrightarrow{[\![k]\!]} \leftarrow \mathsf{readColumn}(db_{state,i}, 0)$; $\overrightarrow{[\![state]\!]} \leftarrow \mathsf{readColumn}(db_{state,i}, 1)$
2  $\overrightarrow{[\![k_{msg}]\!]} \leftarrow \mathsf{readColumn}(db_{msg,i}, 0)$; $\overrightarrow{[\![msg]\!]} \leftarrow \mathsf{readColumn}(db_{msg,i}, 1)$
3  $\left[\overrightarrow{[\![k]\!]}, \overrightarrow{[\![state]\!]}, \overrightarrow{[\![msgs_k]\!]}\right] \leftarrow$ left join $\left[\overrightarrow{[\![k]\!]}, \overrightarrow{[\![state]\!]}\right]$ and $\left[\overrightarrow{[\![k_{msg}]\!]}, \overrightarrow{[\![msg]\!]}\right]$ on the
   1st column, collecting $msg$ to list $msgs_k$
4  foreach $idx, [\![k]\!], [\![state]\!], \overrightarrow{[\![msg_k]\!]}$ as row in $\left[\overrightarrow{[\![k]\!]}, \overrightarrow{[\![state]\!]}, \overrightarrow{[\![msgs_k]\!]}\right]$ do
5      $\overrightarrow{[\![state]\!]}[idx] \leftarrow \mathsf{newState}([\![k]\!], [\![state]\!], \overrightarrow{[\![msg_k]\!]})$
6  $\mathsf{clear}(db_{state,i})$
7  $\mathsf{writeColumn}(db_{state,i}, 0, \overrightarrow{[\![k]\!]})$; $\mathsf{writeColumn}(db_{state,i}, 1, \overrightarrow{[\![state]\!]})$
8  $\overrightarrow{[\![k_{out}]\!]} \leftarrow \emptyset$; $\overrightarrow{[\![msg_{out}]\!]} \leftarrow \emptyset$
9  foreach $[\![k]\!], [\![state]\!]$ as row in $\left[\overrightarrow{[\![k]\!]}, \overrightarrow{[\![state]\!]}\right]$ do
10     $\overrightarrow{[\![k_{temp}]\!]}, \overrightarrow{[\![msg_{temp}]\!]} \leftarrow \mathsf{newMessages}([\![k]\!], [\![state]\!])$
11     $\overrightarrow{[\![k_{out}]\!]}.\mathsf{extend}(k_{temp})$
12     $\overrightarrow{[\![msg_{out}]\!]}.\mathsf{extend}(msg_{temp})$
13 $\mathsf{writeColumn}(db_{out,i}, 0, \overrightarrow{[\![k_{out}]\!]})$; $\mathsf{writeColumn}(db_{out,i}, 1, \overrightarrow{[\![msg_{out}]\!]})$
```

MapReduce already shown in algorithm 3. Computation forms the parallel region of BSP. Realistically, the programs that run in the parallel region do not have to be the same, but may implement different logic; this line of thought may be used to implement task-centric parallelism.

The *computation* sub-task operates on the state database individual to the clique, that is kept in storage over supersteps. A state database is essentially a collection of private key-value pairs, represented in algorithm 5 as vector pair $\overrightarrow{[\![k]\!]}$ and $\overrightarrow{[\![state]\!]}$. Keys being the abstraction for the data identifier, and state being the data that is being processed in iterations – in vertex-centric graph processing for example, keys could be vertex id-s, and a state could contain its edges, edge weights, and vertex augmentations. Each clique is responsible for a specific set of keys over the duration of the whole task.

Based on algorithm 5, a computation process goes through the following general process:

(steps 1-2) A clique reads in the states of its allotted keys (private); also reads in the (private) messages from the previous superstep.

(step 3) Messages are grouped by their recipient key into variable length vectors and joined to the clique's data keys. This has the privacy implication of revealing the distribution of messages, as essentially, vector lengths can not be hidden.

(step 5) The newState UDF is applied to each state with the received messages, calculating the new state, which are updated on disk at step 7.

(steps 10-12) New messages are created based on the state and newMessages UDF, persisted on disk to the messages database at step 13.

The communication program is triggered between computation iterations. It works similarly to the partition step of MapReduce, sorting the messages by the target vertex and partitions them to prepare for ingestion by the following iteration. The single divergence is the fact how messages need to be partitioned – no longer can the keys be distributed evenly w.r.t. the amount of messages, but the partitioning has to be aware of the cliques' key ranges for which they are prepared for. This can be achieved with a separate supplementary database that holds the clique-key relation; as these have to be kept in clear, it is advisable to not adorn any significant meaning to the keys. To hide the movement of messages between cliques, a privacy preserving shuffle, introduced by Laur, et al. [22], can be applied by the partitioner on the total set of messages before sorting.

Focusing on a specific type of computation may help better understand the privacy considerations. Privacy goal of vertex-centric iterative graph processing in this model is to hide the structure of the graph – edge relations, edge weights, and augmentations kept in the state. Not hidden is the distribution of outbound degrees, due to the graph being internally encoded as variable length adjacency lists. Another privacy relaxation is due to the fact that specific computation instances require messages only from the vertices allotted to them. This means that the partitioner has to be aware of the vertex identifier to know which clique to prepare its messages to. It does not mean that this would lead to leakage of the graph structure, for example by tracing which clique sent messages to which other. This is because all vertex identifiers are kept hidden as they are shuffled and sorted using privacy preserving methods by the partitioner before declassification, not revealing the source of the message.

4.3 Clustered Deployment and Scaling

One aim of this thesis is to propose a suitable and modern environment in which parties could host and scale their MPC resources with cost efficiency and ease. As discussed by Bogdanov, et al [11], MPC has great potential to make use of elastic cloud infrastructure, but further work is needed to develop the necessary provisioning tools. The Carbyne Stack project [14] achieved this to a large extent, but due to shortcomings mentioned in section 3.1, is not yet ready to take on real data analysis tasks. Contribution 1 of this thesis shares a common goal with the project, to create a cloud-native deployment of MPC running in clusters, however due to its late emergence, coming to attention of the author during ongoing development, it does not deliberately aim to improve on it. Rather, this work proposes an alternative, building it around the production-ready SHAREMIND framework.

The new deployment should relieve the work and cost usually arising from setting up bare metal servers, networking, and configuration. Instead, an MPC party should be able to bring up, tear down, and configure the system from a single point with provided automation scripts. This enhances accessibility of MPC to a wider user space by eliminating the need for most of the technical know-how required to set up parties' infrastructure today. The solution should also be cost effective compared to on-premises deployments. This can naturally be achieved on the elastic cloud due to the ubiquitous pay-as-you-go model and short-term commitments. All in all, successfully fulfilling these requirements should encourage adoption of MPC in new privacy preserving applications.

The system requires each computing party to host the relevant components in their own cluster. A party alone manages their own configuration and necessary keys, either of which have to be agreed upon in secure means prior to setting it up – public keys of clients and neighbouring parties can be communicated via email as digitally signed documents. To minimise unequal influence over the execution from any single party, the system should not exercise disproportional orchestration patterns, ruling out master-worker or leader designation across parties.

Seamless interconnection for protocol communication has to be set in place. This means that clique servers should automatically be able to discover and reach their counterparts in other clusters without any low level networking configuration. In addition, this should even work for any number of cliques running in parallel over the clusters. Any possible added components that enable this behaviour should not impose excessive latency or bandwidth constraints that would inhibit performance of the protocol.

Horizontally scaling up the cliques should occur on demand of the client. This is to be integrated with the gateway discussed in section 4.2, which should have rights to manage the scaling factor of MPC servers in its cluster if more are requested than currently available. Pruning unused cliques to save costs should happen automatically by checking whether the instance is idle. The latter should not be the responsibility of the gateway, but rather delegate it to standard cluster auto-scaling facilities.

5 Cluster Deployment of SHAREMIND MPC

This chapter documents the devised system based on SHAREMIND. Section 5.1 outlines the architecture and technological choices and specifies how they satisfy requirements set in chapter 4. In section 5.2, the more intricate implementation details of contributed components are given along with examples of applications for the two proposed programming models. Finally, section 5.3 presents benchmarking results and quality assurance analysis.

5.1 Architecture

Components of the system are deployed in a Kubernetes cluster, with each computation party managing their own environment at a cloud provider of their choice, or on-premises. A diagram of the resulting architecture explained in this section is illustrated on figure 5. Kubernetes is opted for due to its design features, most importantly making it easy to scale application containers over a large set of physical machines, fulfilling the requirement of spreading the network workload. It is also the most widespread container management platform, with many cloud providers offering managed control planes and amenities like networked storage, meanwhile useful supporting applications are in large supply thanks to the thriving communities like CNCF. The Docker Swarm platform was considered as an alternative, however Kubernetes was deemed more flexible and extensible for certain tasks. Developing the Kubernetes deployment, vendor agnosticism is kept in mind and it is further demonstrated to run on multiple cloud provider environments in the section 5.3.

5.1.1 SHAREMIND Server StatefulSet

SHAREMIND servers are deployed in the cluster as a StatefulSet[10] resource. StatefulSets provide means to scale application instances in such a way that each replica is adorned with a stable, unique and ordered network identifier. Depending on its scaling factor, it maintains a desired state of a specific amount of containers serving a SHAREMIND instance in the cluster. It provisions the servers, following an anti-affinity rule, to spread evenly among the available nodes, so as to maximize bandwidth quotas of each node. Other cluster services can reach the instances using their local DNS name.

All server instances mount a single network filesystem for communicating intermediate results of sub-tasks. This is backed by a Kubernetes PersistentVolume that allows concurrent write access by multiple Pods (i.e. containerised workloads), which in turn is using a storage solution of the cloud provider. A filesystem approach for data gathering and scattering is chosen due to the tabular data storage interface of SHAREMIND. Other methods such as Redis key-value data stores could also be used by running it in the same

[10] `https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/`

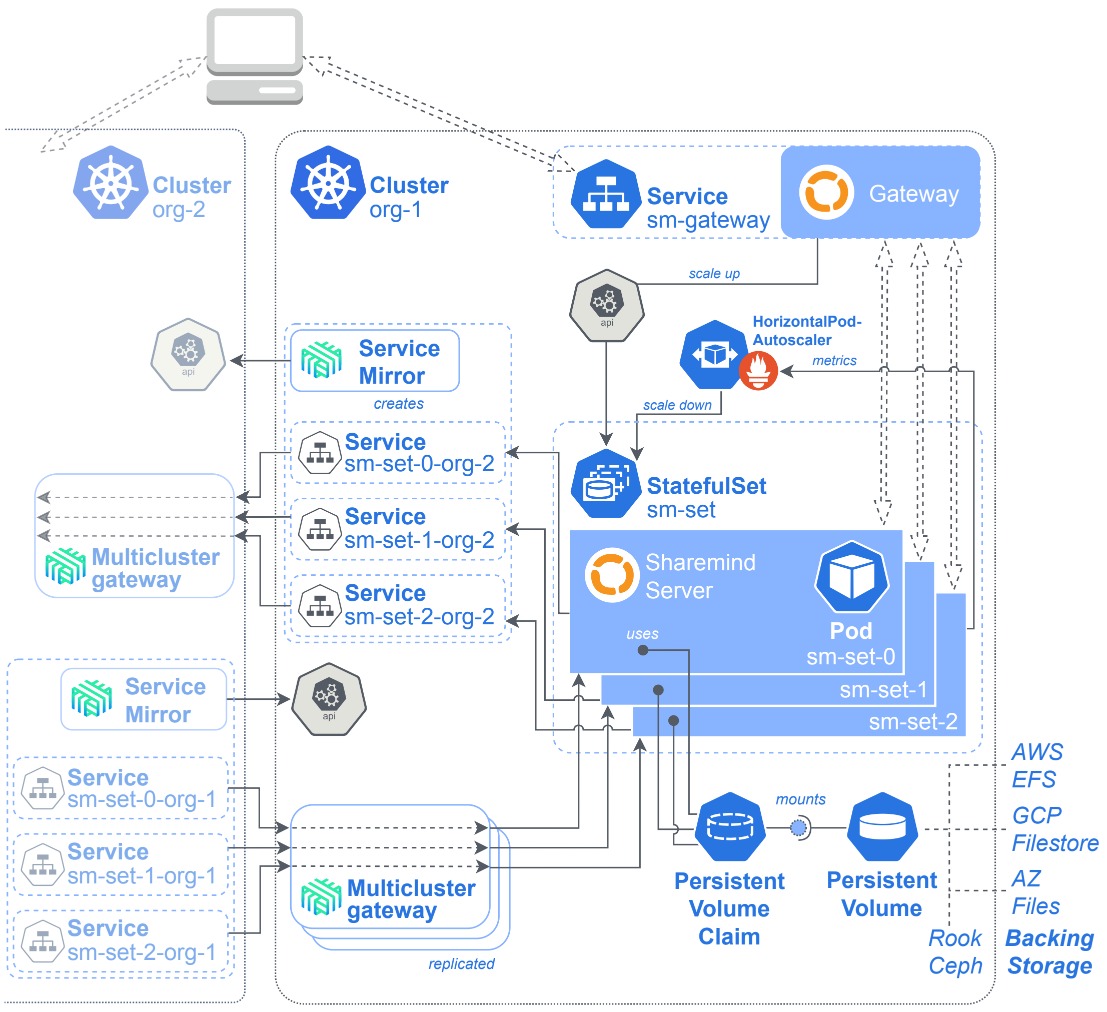


Figure 5. Components and their interactions in the cluster architecture from the view of a single MPC party.

cluster. Redis specifically has utility only in cases where keys can be public values, as it can not index secret shares.

5.1.2 Gateway

The single cluster entry point for clients to initiate multi-party computation is a Java web server, or gateway. Its core is a modified version of a pre-existing library for creating web-based applications supporting SHAREMIND. The original library is a Java analogue of the Node.js Sharemind Web Gateway module[11], containing the native interface of

[11]`https://docs.sharemind.cyber.ee/2022.03/development/sharemind-web-application-tutorial`

SHAREMIND clients, meaning it acts as a client by nature, only translating and proxying the HTTP based process negotiations.

Two main changes are necessary to support parallel deployments of SHAREMIND servers. First, the gateway is required to handle multiple MPC process negotiations, distributing users' inputs to and returning outputs from multiple servers. Parallel process negotiations are always distributed sequentially to each StatefulSet instance, starting from zero. Only data-parallel sub-tasks are considered in the scope of this thesis, aligning with the MapReduce and vertex-centric graph processing usage examples presented in section 5.3, where identical programs are being run in each parallel region. This said, further modifications to the gateway could be done to also support task-based parallelism.

Secondly, it processes clients' parallelisation requests and dictates the desired scale of the server StatefulSet in the cluster. This can be done by granting the gateway resource the sufficient rights to make the respective calls to the Kubernetes control plane's API.

As scaling up is done on demand, scaling down the servers is automated. The Kubernetes HorizontalPodAutoscaler resource is configured to observe the CPU utilisation of the cliques w.r.t. the last server, and initiate a gradual scale-down in the reverse order. This means that any parallel computation taking place with a lower amount of cliques than currently provisioned can continue.

In real-world use, the gateway should itself implement authentication and granular access control, as currently, the gateway library can not carry any client-provided credentials over to the servers. The current architecture has not prioritised this, but if such requirements arise in the future, then it would be straight-forward to implement.

5.1.3 Multi-cluster

Enabling discovery of dynamically provisioned cliques is accomplished by linking the parties' clusters into a multi-cluster. How it works is that parties mirror certain services running in their cluster to the clusters of other parties. For example, if the first party runs a service `sm-set-2` that is flagged for export, then mirroring facilities in clusters of parties 2 and 3 get notified of it, subsequently spawning a local endpoint with a descriptive DNS name like `sm-set-2-org-1.default.svc.cluster.local` that tunnels outgoing requests to the cluster service of party 1. Tunneling happens through a replicated multi-cluster gateway as pictured on figure 5.

This functionality is offered by Linkerd and its multi-cluster extension[12]. Similar solutions for routing between cluster services can be achieved with Istio[13] and Consul Connect[14]. The central use case for each of these technologies is federating several clusters of a single organisation for seamless interoperability. In this work, federation should take place across non-colluding parties and assume no further influence of one cluster

[12]`https://linkerd.io/2.11/features/multicluster/`

[13]`https://istio.io/latest/docs/ops/deployment/deployment-models/`

[14]`https://www.consul.io/docs/k8s/installation/multi-cluster`

over its neighbour other than proxying traffic of appropriately marked services. Linkerd's multi-cluster stands out in this regard, as its requirements[15] prioritise decentralisation of state and control plane, aligning with the isolation and equality requirements for MPC parties.

As a side note, circumventing exotic methods like service mirroring would be possible to an extent while still providing discovery between SHAREMIND servers. For example, the provisioned server instances could be exposed outside of the cluster with unique IPs. This way, the external cloud load balancer could route requests to specific nodes which host the server instance and minimise hops – node hopping being the phenomenon where requests are proxied by intermediate routing facilities between cluster nodes. However, more often than not, every additional public IP comes with extra cost, not to mention that the exposed IPs of parallel instances have to be communicated and kept track of as every cluster scales their deployment. A managed kubernetes platform may allow for multiple LoadBalancer-type services to listen on different ports of the same public IP, in which case parallel instances could allocate consecutive port numbers. Popular cloud providers have different behaviours in this regard – Azure Kubernetes Service allows it, whereas Google Kubernetes Engine does not[16] – although it would be a good fit for the current use case, it is a nonstandard one. Another possibility would be to use an ingress controller, which manages OSI layer 7 routing from within the cluster based on the HTTP-path requested. It is not applicable for SHAREMIND as it currently relies on TCP for intra-clique communication. Linkerd multi-cluster is not susceptible to any provider-based limitations, but it is still important to cover the throughput and latency overhead that may be caused by its proxying nature, which is shown in section 5.3.

5.2 Implementation

This work's tangible output, including deployment scripts and applications can be found alongside its documentation under appendix 6. The following section may reference certain files and directories to aid the reader in navigating its content, but covers the implementation on a general level, reserving the more technical details and instructions for the accompanying *README.md* files.

Deployment of the devised system in existing Kubernetes clusters is automated by Helm charts. Two charts present in the *helm* top-level directory are *multicluster* and *sharemind* for provisioning the Linkerd multi-cluster facilities and MPC-related components, respectively. Setting up the multi-cluster requires some involvement among all MPC parties that employ this system, as a simple public key infrastructure has to be set up for the clusters to securely be able to query each other's API to mirror their services and provide mutual TLS for these channels. This means creating a shared trust anchor,

[15]`https://linkerd.io/2020/02/17/architecting-for-multicluster-kubernetes/`

[16]`https://cloud.google.com/kubernetes-engine/docs/how-to/service-parameters#lb_ip`

or root certificate authority, to sign each Linkerd instance's certificate. The specific possession of the trust anchor's private key is not of significance to the security of MPC, as SHAREMIND employs its own security measures underneath, and of small significance to the clusters – even though Linkerd limits the API interaction to specific calls that do not reveal much, leaking the key might open opportunities for denial-of-service or other attacks. For example, one of the more proactive parties could be chosen to distribute the required certificates. After that matter, the computing parties should select a short DNS suffix, identifying the neighbouring clusters, and configure multi-cluster to differentiate the mirrored services accordingly.

The */helm/sharemind* directory contains a chart for deploying the gateway and server StatefulSet using a unified configuration file, *values.yaml*. As prerequisites, the cluster administrator must first follow the standard procedure of generating public and private keys for the SHAREMIND servers and the gateway, and distribute the server's public key to its neighbouring parties. Further, they must ensure the Kubernetes cluster access to the container registry hosting the gateway and server images. This registry might be provided by the MPC framework's developer in a real scenario, or self-hosted as during the testing of this thesis, but is not publicly served by the author. Cluster administrators are also expected to choose the backing storage medium for server's databases, provision the corresponding cloud resources, and apply possible configurations it may need; section 5.3 and file */helm/sharemind/cluster-storage.md* contain further discussions regarding possible options on popular cloud providers. Finally, the configuration has to be filled in with identities of the neighbouring parties' servers and the SECREC programs that the servers support.

Project files for the parallel gateway implementation reside in */gateway/*. It serves a dual-protocol WebSocket/HTTP API based on Socket.IO, that the SHAREMIND Web Client library interfaces with. Originally, with a single gateway per server, it relayed the client's negotiation messages to a single instance of the gateway library. In turn, parallel deployments required those messages to be terminated and scattered among multiple instantiations. Minimal modifications to the underlying library were required to accommodate this design, however these are not included with this thesis due to involving proprietary code.

Demonstrating the usage of proposed parallel programming models and benchmarking the system, examples of applications are implemented for both MapReduce and BSP. Example client applications for running the MPC tasks can be found in */client/-coocurrence* and */client/sssd*. For the MapReduce application, a program for computing a term co-occurrence matrix is implemented. In addition to being a classic demonstrative MapReduce use case, term co-occurrence is a counting problem, in which the goal is to find the total amount co-occurring value pairs in a set of input value vectors – or more generally, calculating the distribution of pairs of discrete events occurring in a large set of observations [23]. Given the values are words, and vectors being sentences from a corpus, this kind of approach is commonly used for analysing lexical semantics in natural language processing. A more motivating use-case for MPC would be commerce analytics of products or categories of products bought by users over multiple merchants who don't want to reveal their exact sales.

The implementation follows the *stripes* method of keeping intermediate values [23]. Contrary to the so called *pairs* method, which is arguably the intuitive way to approach this problem by having the map program emit the counts of each term-pair occurring in a vector, the stripes method calls for emitting a vector of counts instead. To illustrate, if an input term vector is `[a, b, a, c]`, then rather than emitting `[{(b, a), 2}, {(b, c), 1}, ...]` resulting in a large amount of intermediate values, the stripes method preemptively combines them into a more compact form of `{b: {a: 2, c: 1}}`. In the MPC implementation, the maximum amount of values that can be put in this stripe is bounded by the *stripe length* parameter. This allows putting the emitted values into a rectangular tabular database, and enabling some vectorized operations to be used in the reduce stage – stripes shorter than the bounded length would be padded to the according size with null-terms. All terms and counts, in input, intermediate, and output values are kept private.

For the BSP application, single-source shortest distances (SSSD) with an iterative vertex-centric approach is chosen. In this example, SSSD finds the walk lengths of least weight from a selected vertex to every other vertex on a weighted directed graph. As messages, the vertices send their currently known best length from the source to each neighbour, adding to it the weight of the edge that connects them. Message aggregation is a function for computing the minimum of the messages to acquire the new best length to the vertex. The implementation can be easily adapted to output the sequence of vertices on each shortest path also known as single-source shortest paths.

In the implementation, the state of a vertex contains its outgoing edge targets, the edges' weights, and the currently known shortest path from the source. The state is provided for the first iteration by the client or clients who want to perform data analysis on a graph, each providing a subgraph with agreed upon vertex identifiers. Selecting the source vertex involves setting the specific vertex path length to zero at the start, and the lengths of all other vertices to some sufficiently large value, representing infinity. Careful to avoid overflows, the implementation specifies this as $2^{32}/2$, or half of the 32-bit unsigned integer that lengths are represented as internally. In light of privacy and the fact that this algorithm does not alter the graph's topology, each iteration does the same amount of work in order to hide the graph's structure. This means that each vertex always emits messages, even if the value is a representation of infinity – a party must not know this fact as it would compromise the structure of the graph. As the iteration count of this algorithm can not be known ahead of time, all cliques check whether the message aggregation modified the state in the given iteration, publishing a vote to abort otherwise. If the client receives aborts from all cliques, it refrains from negotiating further processes and considers the computation concluded.

5.3 Evaluation

To get a better understanding regarding the viability and realistic performance benefits of the system, the example programs are ran over three different cloud's managed Kubernetes clusters. This is to mimic a real MPC deployment, where parties are free to use a cloud provider of their choosing, preferably diversifying their choices to mitigate colluding providers. This evaluation makes use of the services offered by Google Cloud Platform (GCP), Amazon Web Services (AWS), and Microsoft Azure to act as cloud platforms of the three parties. All mentioned platforms offer services required by the described architecture:

- managed Kubernetes control planes;

  GCP – Google Kubernetes Engine (GKE)
  EKS – Elastic Kubernetes Service (EKS)
  Azure – Azure Kubernetes Service (AKS)

- provisioned compute node pools;

  GCP – Google Compute Engine (GCE)
  EKS – Elastic Compute Cloud (EC2)
  Azure – Azure Virtual Machines

- network file storage with support for concurrent writing by multiple nodes;

  GCP – Filestore
  EKS – Elastic File System (EFS)
  Azure – Azure Files

Clusters were deployed in close proximity to each other and the client. This meant provisioning the cloud assets in regions within Northern Europe of each provider. AWS and Azure offer all necessary services in Central Sweden, whereas the closest region under GCP is in Southern Finland. All availability zones were utilised to spread any resources within the regions. Node instance sizes were chosen to closely match their counterparts in neighbouring clusters. The common parameters were two virtual CPUs with four GB of RAM per node. Maximum number of provisioned nodes was limited to six due to regional quotas set by Azure. A summary of the cluster details is given in table 1.

Table 1. Details of the cloud environments.

Cloud Platform	**Region**	**Nodes**	**Network Storage**
GCP	`europe-north1`	6×`e2-medium`	1 TiB *Google Filestore* Basic HDD
AWS	`eu-north-1`	6×`t3.medium`	5 GiB *Elastic File System* Standard
Azure	`swedencentral`	6×`B2s`	100 GiB *Azure Files* Standard

Table 2. Storage characteristics

	GCP		**AWS**		**Azure**	
	read	**write**	**read**	**write**	**read**	**write**
Sequential	1776 MiB/s	129 MiB/s	302 MiB/s	102 MiB/s	270 MiB/s	66.6 MiB/s
Random	1097 MiB/s	126 MiB/s	305 MiB/s	99.1 MiB/s	255 MiB/s	83.5 MiB/s
Avg. Latency	172 μs	2539 μs	3031 μs	9780 μs	8750 μs	2497 μs

Each cloud provider required unique approaches for PersistentVolumes supporting concurrent reading and writing by multiple nodes. Such functionality is generally offered by network file systems (NFS), which was settled on for each of the clusters. Azure offered the most convenient workflow with dynamic provisioning of *Azure Files* instances, but minimum size of 100GiB. AWS allowed for mounting dynamically sized Elastic File System (EFS) instances. GCP's Filestore service required pre-provisioning expensive storage devices – for HDD storage, the minimum size being 1TiB, and quotas for new accounts restricted creation of SSD-backed storage. All storage solutions exercised a bursting or throughput allowance system based on the chosen storage type and size, effectively limiting I/O operations if they are used exuberantly. This could be circumvented by deploying an in-cluster network file system on block storage attached directly to some nodes with tools like Rook Ceph[17]. Even though this has the potential to outperform cloud file storage solutions, it incurs significant processing requirements in the cluster and consequently requires more, or more capable nodes.

Since storage throughput is central to the performance of the chosen method of communication, then making note of the metrics is important. The throughput of selected storage solutions were tested with DBENCH[18] from within the cluster and shown in table 2. Tests were performed on freshly provisioned clusters and accounts, so are not expected to reflect any throttling from bursting allowances. Used for prolonged periods, proposed storage solutions may however affect the performance of communication in the parallel applications, so in-cluster storage might be preferred in that case. The observed

[17]`https://rook.io/`
[18]`https://dbench.samba.org/`

Table 3. Network characteristics of Pods communicating via Linkerd's multicluster gateways. Single instance achieved bandwidth is reported in both directions: horizontal axis represents the uploading cluster, whereas vertical is for download throughput.

		GCP	**AWS**	**Azure**
GCP	**throughput**		1.53 Gbits/sec	996 Mbits/s
	latency		15 ms	15 ms
AWS	**throughput**	807 Mbits/sec		1.04 Gbits/sec
	latency	15 ms		11 ms
Azure	**throughput**	933 Mbits/sec	1.03 Gbits/sec	
	latency	15 ms	11 ms	

measurements are not expected to impact the running time of the programs by much, since data volumes in the upcoming benchmarks are at maximum a few megabytes in size and written sequentially.

The larger impact for MPC is due to network performance, most notably latency. Table 3 gives insight about the network characteristics between the clusters. Measurements are taken over the Linkerd multi-cluster and therefore contain the processing overhead that it may add.The metrics were collected with iPerf3 and Linkerd's monitoring utility Viz.It can be seen that Linkerd's multi-cluster offers high performance even when running on relatively small nodes. The observed effective bandwidth of around 1 Gbits/sec and latencies under 20 ms provides a favourable environment for benchmarking MPC in.

Performance metrics of the implemented MPC applications were gathered from client logs, SHAREMIND's built-in profiling, and Kubernetes cAdvisor reporting of cluster resource usage. All reported runtimes are measured from the start of the process negotiation by the client to when the gateway has reported the completion of the sub-task. Parallel regions were run on one through six servers, distributed equally over every cluster node.

The term co-occurrence program was run on 1000 term vectors, each containing 4 to 11 terms represented as integers from 1 to 128; distributions of terms and the vector lengths were uniformly random. The intermediate vector stripes had a fixed length of ten.

The effect of parallelisation on the term co-occurrence implementation can be seen on figure 6. A near linear relative speedup can be observed: two instances finished 1.98 times faster than a single instance, while six instances performed the task 5.21 times quicker. As the number of parallel instances grows, the partition sub-task will take longer to complete, taking around 11 to 17 seconds in the smallest and largest tests, respectively. Since the partition task does not do any operations on shared values that is dependent on the number of the preceding and succeeding parallel instances and the input size stays

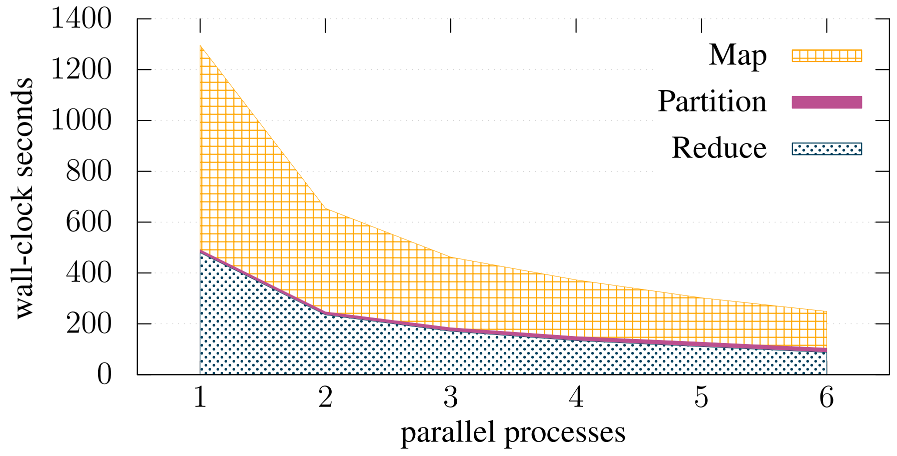


Figure 6. Speedup of term co-occurrence MapReduce application over one to six parallel map and reduce cliques.

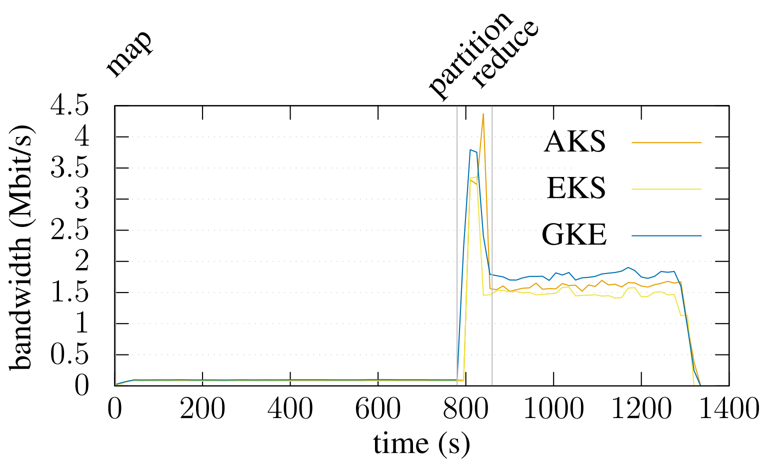
the same, then this runtime difference could only be explained by the increase in work for preparing data for the additional reduce instances.

Network efficiency optimisations of SHAREMIND programs depend heavily on the programmer using vectorized operations on shared values. Naïvely porting a MapReduce solution of a problem to MPC most probably leaves much to be desired in terms of efficiency. This is partly the case with the stripes implementation of term co-occurrence – even though addition in additive secret sharing is free, the map stage requires many equality checks to construct the count *stripes* with SIMD only possible to utilise within the individual input term vectors, but not on a larger scale. Shared integer equality checks in the used protocol set involve $\log_2 n + 2$ communication rounds for n-bit integers [8] – the given term co-occurrence implementation uses 32-bit terms, requiring seven rounds of synchronous communication.

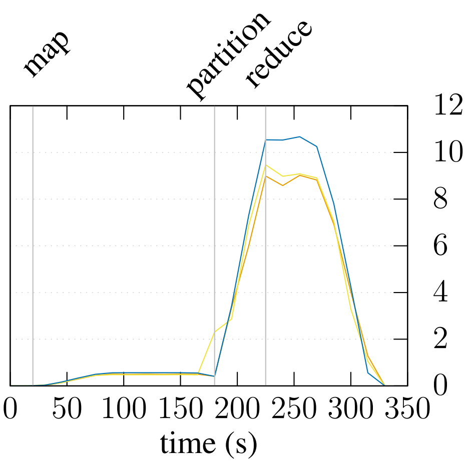
Indeed, by figure 7a, bandwidth utilisation stays low around 0.1 Mbits/s for the duration of the map stage. The partition task runs naturally quickly, as its only non-free operations are fully vectorized. Reducing has some better opportunities for SIMD optimizations than map, utilising 1.5–2 Mbit/s of bandwidth throughout.

Figure 7b illustrates the effect that the non-ideal programs running on parallel instances have on total used bandwidth. With six instances of mappers and reducers, both stages enjoy a linear increase in cumulative throughput. This observation has positive connotations for the practicality of running MPC applications that are difficult to optimise in conventional means, but are easily parallelised otherwise.

The SSSD implementation was tested with a random sparse graph of 10k vertices and 30k edges. The vertex-centric model took 21 iterations to reach a fixpoint and abort. Runtimes reported in figure 8 are extrapolated from a single iteration of computation



(a) Single clique map and reduce



(b) Six clique map and reduce

Figure 7. Term co-occurrence bandwidth consumption by cluster.

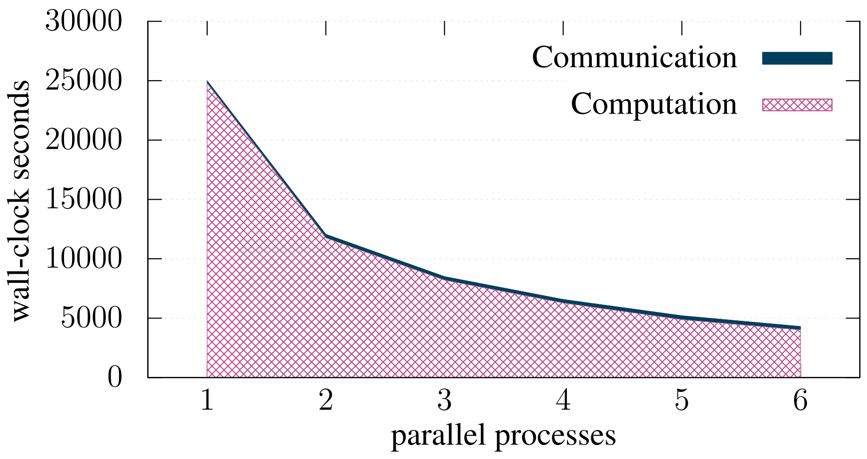
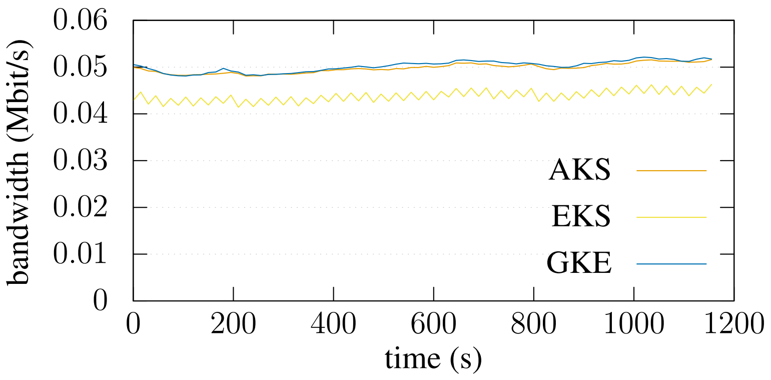


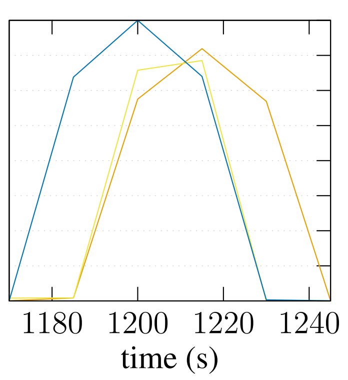
Figure 8. Speedup of SSSD BSP application over one to six parallel cliques.

and communication, supported by the fact that the privacy preserving implementation does the same amount of work in every iteration. Speedup characteristics are identical to those of term co-occurrence, linear w.r.t number of parallel processes. As computation processes were scaled, no evident growth of the communication sub-task's running time, like could be observed from the term co-occurrence's partition program, was noted.

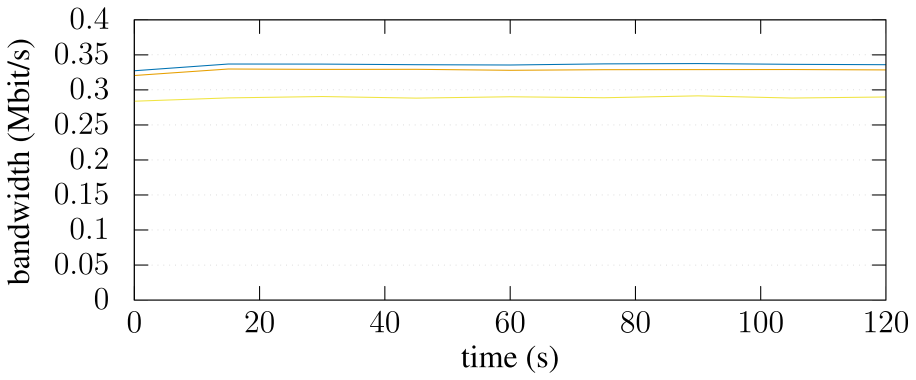
Computing SSSD with MPC was also exercised by Anagreh, et al. [2] with algorithms optimised for vectorized execution in SHAREMIND. Results can not be compared directly due to variations in network characteristics experienced by the different environments – the authors artificially introduce delay of 40 ms for their high-latency benchmarks,



(a) Single clique computation.



(b) Single clique communication.



(c) Six clique computation sub-task summed.

Figure 9. SSSD bandwidth consumption by cluster.

low-latency benchmarks use sub-millisecond latencies of a local network, meanwhile the current setup lies in between the two with 15 ms. Their sparse graph oriented Bellman-Ford implementation finished a relatively similar task size of 9.5k vertices and 28.5k edges within 19.4k seconds in a low-latency environment, while in a high latency environment, a 3k vertex and 9k edge graph took 39.8k seconds. Their dense graph oriented Djikstra's algorithm implementation was not affected by the number of edges, and finished a 10k-vertex graph with 6k seconds in low-latency, with high-latency, only a 1k-vertex graph could be processed during the same amount of time. This shows that performance wise, the vertex-centric solution is comparable to a highly optimised algorithm even in a single clique case, where it demonstrated a 25k second runtime with 10k vertices and 30k edges. Nevertheless, it is important to keep in mind that the current implementation reveals the distribution of edges, and therefore has weaker privacy than algorithms that are compared against.

Similarly to the term co-occurrence map phase, the computation sub-task leaves the bandwidth heavily unutilized. Single clique network throughput over time is shown on figure 9a and 9b for the computation and communication, respectively. Computation results in low bandwidth usage of around 0.05 Mbit/s, much due to its iterative nature in processing messages. With a sparse graph as the one used for this experiment, the vectors of messages moved between iterations are short but plentiful, therefore there are less opportunities to employ SIMD in aggregating them. Figure 9c shows that, as was the case with term co-occurrence, a parallel deployment enjoys linearly scaling bandwidth as more servers are used.

To rule out any other bottlenecks, CPU and RAM usage were monitored in the cluster. The server's container was witnessed utilising a peak of 20% of the two virtual cores of its host during the partition stage of term co-occurrence and communication stage of SSSD. The processors were never throttled by the Kubernetes' scheduler. Working set size (RAM) did not exceed 100 MB during any execution. These observations confirm that MPC is fitting to use in a cluster environment, utilising many small and affordable machines and parallel algorithms to spread the workload, but also enjoy natural relative speedup that results from parallel protocol execution.

The architecture, proposed method of parallelisation, and chosen cloud resources worked together well. Chosen storage solutions were successful in accommodating the storage-based communication between cliques efficiently compared to the performance of MPC. This said, in-cluster storage may be the better choice if data volumes are expected to grow, as it could potentially speed up the communication procedure. The multi-cluster architecture fulfilled its function of routing requests between individual cliques, while not imposing significant overhead in latency. It automatically managed the discovery through DNS, reducing complexity of setting up the MPC environment.

This deployment should also be tested with better optimised MPC programs. Current example applications were bound by latency due to not utilising enough SIMD operations. It would be insightful to see MPC programs saturating the bandwidth, becoming bound by throughput instead. In that case, parallelism could be employed to hypothetically achieve more throughput than of a single node or virtual machine, which may be limited otherwise due to undisclosed quotas or physical network interface restrictions.

6 Conclusion

This thesis' goals were two-fold. First, architecting an environment for secure multi-party computation that is up to date with the latest cloud-computing practices. This was delivered on as a Kubernetes deployment, with the main focus on automatic discovery and enablement of protocol communication between MPC servers. Furthermore, it improves the accessibility of MPC for a wider audience, most notably due to cost savings of *pay for what you use* cloud pricing models as opposed to on-premises hardware, but also by demanding less expertise than manually setting up the infrastructure. The second goal was to exploit the elasticity of the devised cluster deployment to speed up MPC with coarse-grain parallel algorithms over multiple compute instances. This warranted analysis of parallel programming models that are suitable for MPC, both in terms of efficiency and privacy preservation characteristics, lending many ideas from big data processing. MPC constructions and example programs of MapReduce and bulk-synchronous parallel models were presented and benchmarked to show near linear speedup, confirming the results of previous work.

A prospect for further work is apparent for both of these directions. Cloud native MPC could be the next step in the proliferation of privacy-preserving data analysis. For this, MPC frameworks should adapt to, or be designed around cloud and cluster deployments. This work designed its solution for one specific framework, SHAREMIND MPC, requiring it to conform to its specific needs. Better architectural choices could be made if frameworks natively support standard cloud storage options and serverless deployment, as demonstrated by CarbyneStack. The utility of parallel programming as shown in this work should be evaluated on real business cases. This is to verify whether it could sustain its benefits outside of simulated scenarios, with realistic analysis requirements, data models and volumes. An immediate subject for further experiments could be to parallelise subroutines of the Rmind statistical analysis tool.

As for concrete functionalities, the gateway application created in this thesis could be further developed to support configurable client authentication and authorization to run specific tasks, without which, the system is not yet feasible to be deployed in production. Use cases for task-oriented MPC parallelism may also turn out to be desirable, prompting further modifications to the currently data-oriented parallel gateway.

Acknowledgements

The author of this thesis wishes to thank his colleagues from Cybernetica for their engagement in technical discussions, and for providing access to the SHAREMIND MPC framework. Further gratitude is expressed towards STACC for their support throughout this work's development.

References

[1] Nikolaos Alexopoulos, Aggelos Kiayias, Riivo Talviste, and Thomas Zacharias. {MCMix}: Anonymous Messaging via Secure Multiparty Computation. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1217–1234, 2017. ISBN 978-1-931971-40-9. URL `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/alexopoulos`.

[2] Mohammad Anagreh, Peeter Laud, and Eero Vainikko. Parallel Privacy-Preserving Shortest Path Algorithms. *Cryptography*, 5(4):27, December 2021. ISSN 2410-387X. doi: 10.3390/cryptography5040027. URL `https://www.mdpi.com/2410-387X/5/4/27`. Number: 4 Publisher: Multidisciplinary Digital Publishing Institute.

[3] Mohammad Anagreh, Eero Vainikko, and Peeter Laud. Parallel Privacy-Preserving Shortest Paths by Radius-Stepping. In *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 276–280, March 2021. doi: 10.1109/PDP52278.2021.00051. ISSN: 2377-5750.

[4] Mohammad Anagreh., Eero Vainikko., and Peeter Laud. Parallel privacy-preserving computation of minimum spanning trees. In *Proceedings of the 7th International Conference on Information Systems Security and Privacy - ICISSP,*, pages 181–190. INSTICC, SciTePress, 2021. ISBN 978-989-758-491-6. doi: 10.5220/0010255701810190.

[5] Mohammad Anagreh., Peeter Laud., and Eero Vainikko. Privacy-preserving parallel computation of shortest path algorithms with low round complexity. In *Proceedings of the 8th International Conference on Information Systems Security and Privacy - ICISSP,*, pages 37–47. INSTICC, SciTePress, 2022. ISBN 978-989-758-553-1. doi: 10.5220/0010775700003120.

[6] David W. Archer, Dan Bogdanov, Benny Pinkas, and Pille Pullonen. Maturity and Performance of Programmable Secure Computation. *IEEE Security Privacy*, 14(5): 48–56, September 2016. ISSN 1558-4046. doi: 10.1109/MSP.2016.97. Conference Name: IEEE Security Privacy.

[7] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, Ariel Rabkin, Ion Stoica, et al. Above the clouds: A berkeley view of cloud computing. Technical report, Technical Report UCB/EECS-2009-28, EECS Department, University of California ..., 2009.

[8] Dan Bogdanov. *Sharemind: programmable secure computations with practical applications*. PhD thesis, University of Tartu, 2013. URL `http://hdl.handle.net/10062/29041`.

[9] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security*, 11(6):403–418, November 2012. ISSN 1615-5270. doi: 10.1007/s10207-012-0177-2. URL `https://doi.org/10.1007/s10207-012-0177-2`.

[10] Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. How the Estonian Tax and Customs Board Evaluated a Tax Fraud Detection System Based on Secure Multi-party Computation. In Rainer Böhme and Tatsuaki Okamoto, editors, *Financial Cryptography and Data Security*, Lecture Notes in Computer Science, pages 227–234, Berlin, Heidelberg, 2015. Springer. ISBN 978-3-662-47854-7. doi: 10.1007/978-3-662-47854-7_14.

[11] Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. Privacy-preserving tax fraud detection in the cloud with realistic data volumes. *T-4-24, Cybernetica AS*, 2016.

[12] Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, and Riivo Talviste. Students and taxes: a privacy-preserving study using secure computation. *Proc. Priv. Enhancing Technol.*, 2016(3):117–135, 2016.

[13] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael Schwartzbach, and Tomas Toft. Secure Multi-party Computation Goes Live. In Roger Dingledine and Philippe Golle, editors, *Financial Cryptography and Data Security*, Lecture Notes in Computer Science, pages 325–343, Berlin, Heidelberg, 2009. Springer. ISBN 978-3-642-03549-4. doi: 10.1007/978-3-642-03549-4_20.

[14] Bosch. Carbyne stack. `https://carbynestack.io/`, 2022. Accessed: 2022-03-24.

[15] Rajkumar Buyya, Satish Narayana Srirama, Giuliano Casale, Rodrigo Calheiros, Yogesh Simmhan, Blesson Varghese, Erol Gelenbe, Bahman Javadi, Luis Miguel Vaquero, Marco A. S. Netto, Adel Nadjaran Toosi, Maria Alejandra Rodriguez, Ignacio M. Llorente, Sabrina De Capitani Di Vimercati, Pierangela Samarati, Dejan Milojicic, Carlos Varela, Rami Bahsoon, Marcos Dias De Assuncao, Omer Rana, Wanlei Zhou, Hai Jin, Wolfgang Gentzsch, Albert Y. Zomaya, and Haiying Shen. A Manifesto for Future Generation Cloud Computing: Research Directions for the Next Decade. *ACM Computing Surveys*, 51(5):105:1–105:38, November 2018. ISSN 0360-0300. doi: 10.1145/3241737. URL `https://doi.org/10.1145/3241737`.

[16] Ronald Cramer, Ivan Damgård, and Ueli Maurer. General Secure Multi-party Computation from any Linear Secret-Sharing Scheme. In Bart Preneel, editor, *Advances in Cryptology — EUROCRYPT 2000*, Lecture Notes in Computer Science, pages 316–334, Berlin, Heidelberg, 2000. Springer. ISBN 978-3-540-45539-4. doi: 10.1007/3-540-45539-6_22.

[17] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, San Francisco, CA, December 2004. USENIX Association. URL `https://www.usenix.org/conference/osdi-04/mapreduce-simplified-data-processing-large-clusters`.

[18] Cloud Native Computing Foundation. Cloud native computing foundation. `https://www.cncf.io/`, 2022. Accessed: 2022-03-24.

[19] Marcel Keller. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1575–1590. Association for Computing Machinery, New York, NY, USA, October 2020. ISBN 978-1-4503-7089-9. URL `https://doi.org/10.1145/3372297.3417872`.

[20] Nane Kratzke and Peter-Christian Quint. Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study. *Journal of Systems and Software*, 126:1–16, April 2017. ISSN 0164-1212. doi: 10.1016/j.jss.2017.01.001. URL `https://www.sciencedirect.com/science/article/pii/S0164121217300018`.

[21] Peeter Laud and Liina Kamm. *Applications of secure multiparty computation*, volume 13. Ios Press, 2015.

[22] Sven Laur, Jan Willemson, and Bingsheng Zhang. Round-Efficient Oblivious Database Manipulation. In Xuejia Lai, Jianying Zhou, and Hui Li, editors, *Information Security*, Lecture Notes in Computer Science, pages 262–277, Berlin, Heidelberg, 2011. Springer. ISBN 978-3-642-24861-0. doi: 10.1007/978-3-642-24861-0_18.

[23] Jimmy Lin and Chris Dyer. *Data-intensive text processing with MapReduce*, volume 3. Morgan & Claypool Publishers, 2010.

[24] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, June

2010. Association for Computing Machinery. ISBN 978-1-4503-0032-2. doi: 10.1145/1807167.1807184. URL `https://doi.org/10.1145/1807167.1807184`.

[25] Sahar Mazloom and S. Dov Gordon. Secure Computation with Differentially Private Access Patterns. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 490–507, New York, NY, USA, October 2018. Association for Computing Machinery. ISBN 978-1-4503-5693-0. doi: 10.1145/3243734.3243851. URL `https://doi.org/10.1145/3243734.3243851`.

[26] Sahar Mazloom, Phi Hung Le, Samuel Ranellucci, and S. Dov Gordon. Secure parallel computation on national scale volumes of data. pages 2487–2504, 2020. ISBN 978-1-939133-17-5. URL `https://www.usenix.org/conference/usenixsecurity20/presentation/mazloom`.

[27] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing. *ACM Computing Surveys*, 48(2):25:1–25:39, October 2015. ISSN 0360-0300. doi: 10.1145/2818185. URL `https://doi.org/10.1145/2818185`.

[28] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. GraphSC: Parallel Secure Computation Made Easy. In *2015 IEEE Symposium on Security and Privacy*, pages 377–394, May 2015. doi: 10.1109/SP.2015.30. ISSN: 2375-1207.

[29] Matthew Felice Pace. BSP vs MapReduce. *Procedia Computer Science*, 9:246–255, January 2012. ISSN 1877-0509. doi: 10.1016/j.procs.2012.04.026. URL `https://www.sciencedirect.com/science/article/pii/S1877050912001470`.

[30] Jaak Randmets. *Programming Languages for Secure Multi-party Computation Application Development*. PhD thesis, University of Tartu, 2017. URL `http://hdl.handle.net/10062/56298`.

[31] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, November 1979. ISSN 0001-0782. doi: 10.1145/359168.359176. URL `https://doi.org/10.1145/359168.359176`.

[32] Riivo Talviste. *Applying Secure Multi-party Computation in Practice*. PhD thesis, University of Tartu, 2016. URL `http://dspace.ut.ee/handle/10062/50510`.

[33] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990. ISSN 0001-0782. doi: 10.1145/79173.79181. URL `https://doi.org/10.1145/79173.79181`.

[34] Nikolaj Volgushev, Andrei Lapets, and Azer Bestavros. Scather: programming with multi-party computation and mapreduce, 2015. URL `https://open.bu.edu/handle/2144/21774`.

[35] Nikolaj Volgushev, Andrei Lapets, and Azer Bestavros. Programming Support for an Integrated Multi-Party Computation and MapReduce Infrastructure. In *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, pages 60–66, November 2015. doi: 10.1109/HotWeb.2015.21.

[36] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 1–18, New York, NY, USA, March 2019. Association for Computing Machinery. ISBN 978-1-4503-6281-8. doi: 10.1145/3302424.3303982. URL `https://doi.org/10.1145/3302424.3303982`.

[37] Andrew C. Yao. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 160–164, November 1982. doi: 10.1109/SFCS.1982.38. ISSN: 0272-5428.

Appendix

I. Source Code Repository

Programmes and deployment scripts developed for this thesis are openly accessible with supplementing documentation at the following code repository:
`https://github.com/ktali/clustered-parallel-mpc`.