DFI: The Data Flow Interface for High-Speed Networks

Lasse Thostrup
TU Darmstadt

Jan Skrzypczak*
Zuse Institute, Berlin

Matthias Jasny
TU Darmstadt

Tobias Ziegler
TU Darmstadt

Carsten Binnig
TU Darmstadt

ABSTRACT

In this paper, we propose the Data Flow Interface (DFI) as a way to make it easier for data processing systems to exploit high-speed networks without the need to deal with the complexity of RDMA. By lifting the level of abstraction, DFI factors out much of the complexity of network communication and makes it easier for developers to declaratively express how data should be efficiently routed to accomplish a given distributed data processing task. As we show in our experiments, DFI is able to support a wide variety of data-centric applications with high performance at a low complexity for the applications.

1 Introduction

Motivation: Scale-out data processing systems are the typical architecture used today by many systems to process large data volumes since they allow applications to increase compute and memory capacities by simply adding further processing nodes. However, a typical bottleneck in scale-out systems is the network which often slows down the speed of data processing if communication is in the critical path. For distributed in-memory systems this might lead to degraded performance when adding more nodes.

However, this changed with the advent of high-speed networks such as InfiniBand. Network bandwidth increased almost up to the speed of main memory and latencies dropped by orders of magnitude [4], making scale-out solutions more competitive. However, blindly upgrading to faster networks does often not directly translate into performance gains, as there is a plenitude of aspects to consider to achieve a good performance for distributed data processing systems.

One particular important aspect to efficiently use high-speed networks is to redesign data processing systems to leverage remote direct memory access (RDMA) as a low overhead communication protocol. RDMA provides kernel bypass and zero-copy making data transfers less expensive than classical network stacks such as TCP/IP [9]. In recent years, industry and academia have thus started to adapt scale-out data processing systems in order to make use of RDMA. As a result, significant speed-ups have been shown for a wide range of data processing systems ranging from key-value stores [11, 15, 17], over distributed DBMSs (for OLTP and OLAP) [4, 12, 23] to Big Data systems and Distributed Machine Learning [22].

However, using RDMA is complicated because it provides only low-level abstractions (called RDMA verbs) for data processing. Hence, redesigning data processing systems for RDMA often requires significant efforts to take care of many low-level detail choices [4, 3, 12, 25] regarding remote memory and connection management as well as other decisions such as which RDMA verbs to use for which type of workload.

Contribution: In this paper, we propose the Data Flow Interface (DFI) as a way to make it easier for data processing systems to exploit high-speed networks. Accordingly, DFI defines abstractions and interfaces suited to a broad class of data-intensive applications, yet simple enough for practical implementation with predictable performance and low overhead relative to "hand-tuned", ad hoc alternatives. In designing a high-level interface tailored to data processing, we adopt the approach taken by the high-performance community for MPI [10] to provide a simple yet effective interface for high-speed networks. However, since MPI has been designed for computation-intensive workloads such as large-scale simulations, it comes with many design choices that are not optimal for data-intensive workloads [13]. Consequently, MPI has seen only very limited adoption for data processing systems [2].

In brief, the main idea of DFI is that data movements are represented as *flows*. DFI flows are an abstraction providing primitives for efficient network communication. These primitives are intended to be used as a foundation for building data-intensive systems and provide many benefits over MPI (e.g., thread-centricity and pipelined communication). By lifting the level of abstraction, DFI flows not only hide much of the low-level complexity of network communication but also allow developers to declaratively express how data should be efficiently routed to accomplish a given distributed data processing task. Moreover, DFI flows allow developers to specify *optimization hints*; e.g., to maximize bandwidth-utilization or minimize network latency of transfers. By using flows as the main abstraction, DFI supports a wide variety of data-centric applications ranging from bandwidth-sensitive distributed OLAP to more latency-sensitive work-

*Work done while at Zuse Institute, Berlin

loads such as distributed OLTP or replication with consensus protocols.

Recently, the need for better interfaces to high-speed networks has also been discussed in a vision paper [1]. We, however, are the first paper that provide a concrete suggestion and a full implementation for an interface that can enable a broad class of data-centric applications to make efficient use of modern networks. Moreover, there have also been several other attempts to build libraries for data processing over high-speed networks [7, 8, 5]. For example, FaRM [7] and GAM [5] provide a programming model based on a shared address space which focuses on supporting latency-sensitive workloads (e.g., such as distributed transactions). Another example is L5 [8], which target the communication between clients and servers to replace traditional client-centric communication libraries such as ODBC. Different from those libraries, as mentioned before, DFI flows aim to be a much more general abstraction that can support a broader class of data-centric applications.

In summary, this paper makes the following contributions:

- First, we present the design of DFI based on the general abstraction of flows that allow developers to declaratively specify the communication behavior of distributed systems by defining its topology (1:1, N:1, 1:N and N:M) as well as providing other properties for execution.
- Second, we provide a first implementation of DFI[1] for an InfiniBand-based networking stack and discuss how the high-level abstractions of DFI are being mapped to the low-level implementations using RDMA.
- Third, we provide an evaluation of our DFI implementation and demonstrate that DFI is able to efficiently utilize the network but also showcase that DFI can provide high performance for different data-centric applications.

Outline: The remainder of this paper is structured as follows: In Section 2, we first give an overview of two existing interfaces, RDMA verbs and MPI. Afterwards, in Section 3 we present an overview of DFI before we discuss details of the programming model in Section 4 as well as our implementation for InfiniBand in Section 5. Finally, we conclude with our evaluation in Section 6, details on how to integrate DFI in Section 7 and a summary in Section 8.

2 Existing Interfaces

In this section we give a short overview of existing interfaces namely the standard RDMA verbs interface native to the InfiniBand network stack and the Message Passing Interface (MPI), the de facto standard in the HPC community.

2.1 RDMA Verbs

The InfiniBand RDMA verb interface is a low-level interface providing low latency and high bandwidth communication. The interface exposes one-sided verbs (*write*, *read* & *atomics*) and two-sided verbs (*send* & *receive*) which refer to the involvement of end-points (i.e., one-sided verbs only involves the CPU of the sender). The high performance of RDMA is in general achieved by the asynchronous nature of RDMA, making it possible to pipeline computation and communication such that the CPU is not busy idling during network communication. To issue RDMA verbs (one- or two-sided), the application has to register a memory region in which the RNIC can directly access memory, leaving communication related memory-management a responsibility of the application. Moreover, due to the RDMA verb interface's very low abstraction level it provides also a huge design space. This requires applications need to carefully explore this design space and to optimally make use the available low-level options [8, 25, 11, 24].

[1] `https://github.com/DataManagementLab/DFI-public`

2.2 Message Passing Interface

The Message Passing Interface (MPI) is widely used by the HPC community as a high-level abstraction for high-speed networks, and has through many years of development reached a mature and industrial-strength quality. It has however seen limited adoption in data management systems due to its synchronization heavy block-synchronous-parallel processing model and poor multi-threading performance [20].

3 DFI Overview

In this section, we first highlight the central design goals of DFI before we discuss the flow-based programming model, as well as the high-level idea of the execution model behind flows.

3.1 Key Design Principles

The aim of DFI is to provide a high-level abstraction that provides efficient support for a broad set of data processing systems. In the following, we present the key design principles of DFI to ideally support the needs of these systems:

(1) Pipelining: Different from MPI, which targets compute-centric applications such as distributed simulations, many data-centric applications are often dominated by data transfers (i.e., data shuffling). For this reason, it is shown to be crucial that computation and communication can be overlapped [3].

(2) Thread-centricity: Multi-threading is essential not only in achieving high degrees of parallelism in modern data-centric architectures but also to saturate the network as mentioned before. Hence, different from MPI, DFI should be designed from ground up to enable a thread-centric execution and communication model.

(3) Low-overhead synchronization: Another important aspect that goes along with thread-centricity is that DFI aims to provide low-overhead synchronization between sender and receiver threads as well as between sender threads that target the same receiver. By providing low-overhead synchronization, DFI thus should enable scalability to a high number of sender and receiver threads.

(4) Declarative optimization: A last important goal is that DFI exposes parameters as a handle for applications to declare what optimizations are desired. Examples of such optimizations are whether applications are bandwidth or latency sensitive, but also other guarantees such as global ordering of messages when data is send across flows (which is important, for example, for data replication protocols).

3.2 Flow-based Programming Model

At the center of the abstraction are DFI's flows. Flows encapsulate the movement of data between end-points in a

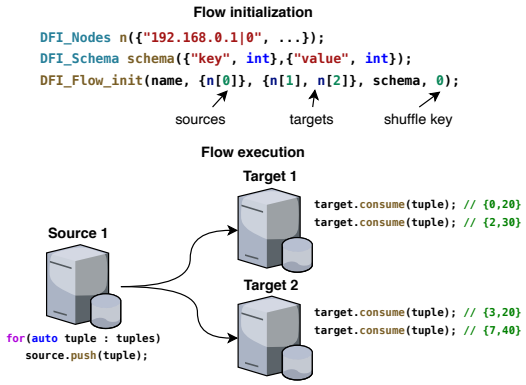


Figure 1: DFI's Programming and Execution Model. Example of flow initialization for setting up a shuffle based flow. The flow execution exemplifies the tuple-based push and consume primitives on DFI.

distributed application, by exposing *sources* and *targets* as data entry and exit points on a per thread-level. This simple abstraction allows applications to compose potentially complex communication topologies, including both point-to-point, one-to-many, many-to-one and many-to-many communications between worker threads of multiple nodes. As we show later in this section, the flow abstraction is powerful enough to support a wide range of data processing use-cases such as distributed join algorithms, but also consensus protocols.

In the following, we provide an example of a concrete many-to-many flow type in DFI, which is one out of multiple other flow types as we discuss later. The most common many-to-many communication in data processing systems is arguably key-based shuffling of data across multiple sources and targets. An example of such a shuffle flow in DFI is illustrated in Figure 1.

As we see in the example, before a flow can be used it first has to be initialized by specifying a unique flow name identifier, location of source and target threads identified with the node address and a thread ID in DFI_Nodes, the schema of tuples that are transferred and on which key the tuples should be shuffled (see upper part of Figure 1). Note, it is also possible for applications to specify application-specific partition functions, but as default a simple key-based hash function is used to partition the tuples across receivers.

To make the flow available for other nodes, its metadata is published in a central registry upon initialization (e.g., a master node in a distributed system). For using a flow, sources and targets first need to retrieve the flow metadata from the central registry. The source nodes can then use a flow by pushing tuples into the flow and the target to consume tuples out of the flow by pulling from the flow (see lower part of Figure 1).

In addition to shuffle flows between N senders and M receivers, DFI provides other flow types (i.e., a combiner and a replicate flow) and topologies (i.e., 1:1, N:1, 1:N and N:M) to support various data processing applications. More details about the full programming model of DFI will be explained in Section 4.

3.3 High-level Flow Execution

Key to the execution model of DFI's flows are the design principles discussed above. We achieve these design principles by implementing an execution model where each thread with a source or target has a private send/receive buffer that not only decouples sender from receiver threads but also uses a new memory layout for remote data transfer between sender/receiver threads with only minimal synchronization overhead as we discuss next.

Flow type	*Communication topology*	*Flow options*
Shuffle flow	1:1, N:1, 1:N, N:M	Bandwidth/latency
Replicate flow	1:N, N:M	Bandwidth/latency + ordering guarantees
Combiner flow	N:1	Bandwidth/latency + various aggregations

Table 1: DFI flow types for a wide range of data-centric applications. Communication topologies and flow options further allow applications to adjust the behavior of flows based on application requirements.

In the following, we present the high level execution of flows by following the example of shuffling tuples shown in Figure 1. The push primitive on sources is asynchronous and returns immediately after the tuple to be transferred is copied into the internal send buffer. This non-blocking behavior allows applications to interleave the computation and communication, i.e., pipeline, and thus utilize both CPU and network resources. Moreover, internally the flow execution heavily uses the available one-sided RDMA primitives to reduce the CPU involvement of the targets, and thus decouples the sources and targets as much as possible. To enable one-sided network communication, as mentioned before, a receive buffer must be in place in which the tuples of one or multiple sending threads are written to. Details about the buffer design and their low-overhead synchronization model are discussed further in Section 5.

Once a tuple has been pushed into the flow, a routing decision will be made by the flow based on the provided shuffling key. Depending on the chosen optimization goal (bandwidth or latency), the execution of the flow will transport tuples across the network. For bandwidth optimization, flows batch tuples together destined for the same target in order to achieve a better bandwidth utilization through larger messages. On the other hand if a latency optimization is chosen, the flow execution will prioritize transferring the tuple as soon as possible.

4 Programming Model

In the following, we present a more detailed view on the programming model of DFI and its main abstractions by detailing the opportunities for setting up various communication flow types. In addition, the programming model will be demonstrated through a set of concrete use cases.

4.1 DFI Flows

So far we have only presented a concrete example of constructing a flow for shuffling tuples between a set of source and target threads. However, DFI defines flows with different characteristics to support the wide demands of data processing systems. Table 1 shows the three flow types in DFI, the communication topologies supported by the corresponding flows, as well as their declarative flow options.

The flow abstraction also offers easy adaptability of application algorithms, since different types of flows can be triv-

ially exchanged to offer different behaviors. For instance, to change a symmetric re-partition join algorithm into a fragment-and-replicate join, instead of using a shuffle flow that routes tuples based on the join key, use a replicate flow to replicate the inner table. Performing such algorithmic changes on typical solutions leveraging the RDMA verb interface would infer a significant rewrite of the communication relevant parts of the solution.

In the following, we discuss the different flow types and their potential use in data processing systems.

Shuffle Flow: The shuffle flow is a central abstraction of DFI, where various different communication patterns and routing options can be specified. The communication pattern is indirectly defined by declaring the participating sources and targets in the flow initialization, and can therefore follow 1:1, N:1, 1:N and N:M communication patterns between sending and receiving threads.

The routing of tuples from sources to targets can be defined in three ways in a shuffle flow: (1) The application specifies the shuffle key and let DFI handle the routing. (2) A routing function can be supplied for more control, e.g., to realize different partition functions such as range-partitioning or radix hash partitioning. (3) Lastly, it is also possible to directly specify the node identifier of a target thread on each push into the flow.

Replicate Flow: Another flow type that DFI provides is a so called replicate flow, which targets data processing tasks involving data duplication, such as replicated state machines, fragment-and-replicate join operators or data duplication for stream processing.

The performance of a naïve replication of tuples which uses multiple RDMA operations (i.e., one for each target), will quickly become limited by the outgoing link-speed of the source node; e.g., a replicate flow with 1 source and 8 targets, will have to divide the available network bandwidth at the source, if messages are replicated to all 8 targets on the source node. In DFI, we instead make use of RDMA multicast such that when enabled, messages are replicated in the network as to prevent the outgoing link of the source(s) from becoming a bottleneck.

For some applications using replication, ordering of messages plays an important role. An example of this is state machine replication, where the correctness depends on all replicas processing the incoming operations in the same order. Since many networks (including InfiniBand) do not provide this guarantee if multiple receivers are involved [16] (even not for simple networks with only one switch), replicate flows can be initialized to provide global ordering guarantees, such that all targets consume tuples out of the flow in the same order.

Combiner Flow: The third flow type supported by DFI is the combiner flow. The focus of the combiner flow is many-to-one communication patterns which is typically used in aggregation scenarios, such as a SQL aggregation or a parameter server for distributed machine learning. The combiner flow supports different aggregations (e.g., SUM, COUNT, MIN, MAX) to be performed on the tuples.

Again while a naïve implementation would implement the reduction at the target node, the network can be used to accelerate the reduction. For example, InfiniBand offers the SHARP protocol, that enables in-network aggregations for high-speed InfiniBand networks and thus could help to mitigate when the in-bound network of the receiver becomes a bottleneck.

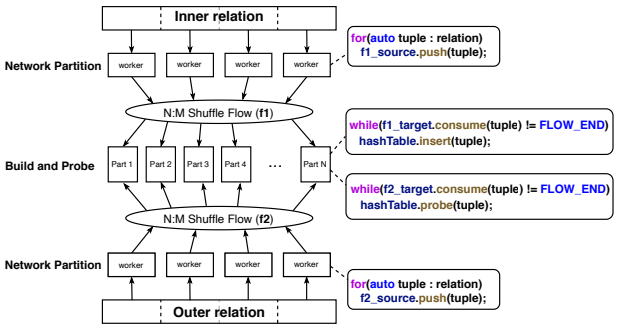


Figure 2: Distributed Radix Hash Join with DFI flows. Two shuffle flows are used to partition tuples across network, one for each relation.

4.2 Use Cases

In the following we present two distributed data processing use cases and how they are realized through DFI: First, we discuss distributed joins for OLAP where the aim is to reduce the runtime by making efficient use of the available network bandwidth. Second, we present a distributed consensus use case where the performance criteria is low latency and high message throughput.

Distributed Radix Join: The distributed radix hash join is a popular join operator due to its dominating performance [3, 2]. The idea behind the radix hash join is to partition the input relations into such small partitions that the resulting hash tables fit into the CPU caches to reduce cache-misses.

In its original form the distributed radix join has a high level of complexity since multiple senders need to coordinate when writing to the same receivers. For example, in [3, 2], histograms of buckets are pre-computed in a first pass on each input table to allocate private memory buffers for each thread on the receiver node and then use coordination-free one-sided communication in a second pass to shuffle the data of each input table.

We argue that with DFI, the design of a distributed radix join is simpler while the performance is on par (and sometimes even better) with the latest distributed radix join implementations (as will be shown in Section 6.2). To realize the join with DFI, two bandwidth optimized shuffle flows are used as shown in Figure 2, one for shuffling each relation. Figure 2 also shows the pseudo-code how tuples can be pushed into the flows during network partitioning, and consumed at the target (i.e., receiver node) out of the flows for the relations to either build the hash table (for the inner relation) or probe the hash table (for the outer relation).

The shuffle flows for the join are initialized with one source per sender thread and one target per output partition. That way the flow can used for achieving the desired partition fan-out. The routing of tuples to the partition-specific *targets* is done on a per thread level by passing a radix hash function to DFI as the routing function. This also leads to a reduction of complexity of the DFI join compared to the original RDMA-based distributed radix join since the histogram computation can be omitted. Moreover, the memory management of local and remote buffers is handled in DFI.

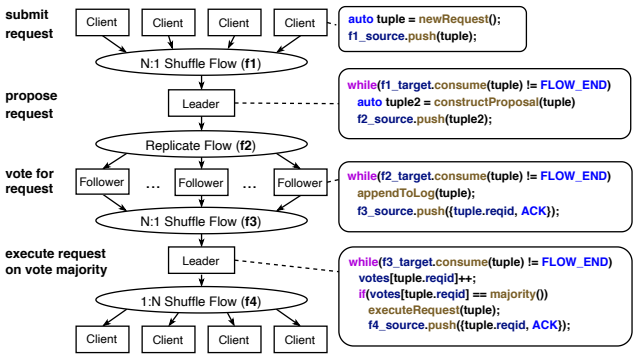


Figure 3: Leader-based consensus with DFI flows. Four flows are used to realize consensus between replicas.

Distributed Consensus: Consensus in a distributed system describes the agreement of multiple (often asynchronous) participants on a single value, or a sequence of values, while tolerating the presence of faulty participants. It is a fundamental primitive in distributed computing which is needed, for example, for the reliable implementation of replicated state machines, leader election, or system reconfiguration.

Classical consensus protocols [14, 18] are centered around a centralized coordinator, called leader. The leader orders concurrently arriving requests of participants (i.e., clients) and forwards them to a set of so called followers. The followers vote for requests that they receive from the leader. Once the leader has received a majority of votes (itself included), the leader can notify the corresponding client that its request was agreed-upon. The high-level message flow of a leader-based consensus implementation using DFI can be modeled directly with the flows provided by DFI and is depicted in Figure 3. Figure 3 additionally shows pseudo-code of how these flows are used for the communication which we explain in the following.

Clients initially send their vote with an N:1 shuffle flow to the leader. The replicate flow is ideal to handle the communication from the leader to its followers, as all followers receive identical messages. The use of the RDMA multicast verbs built into DFI alleviates load placed on the leader compared to the naïve replication of messages. This is an interesting optimization, as the leader is typically a major bottleneck in consensus-based systems. Once followers received the request and voted for a result, they send the outcome back to the leader, again using a shuffle flow. In a last step the leader distributes the consensus-outcome to the client using the client IDs as the shuffle key.

An interesting optimization that DFI provides is to use the optimization option for global ordered multicast (also referred to as ordered unreliable multicast - OUM). In particular, Li et al. [16] propose a single round-trip consensus protocol based on OUM. While this work focuses on Ethernet-based systems, to our knowledge, DFI is the first system that can provide these semantics in the context of InfiniBand.

As we show in Section 6.2.2, using the ordered multicast significantly improves both throughput and latency compared to conventional consensus protocol designs using native RDMA that follow more classical consensus designs.

5 Flow Implementation

In this section, we briefly present how we realized the key design principles discussed in Section 3 through an implementation of DFI. A more extensive discussion of the implementation along with details on the different flow implementations can be found in the original publication [20].

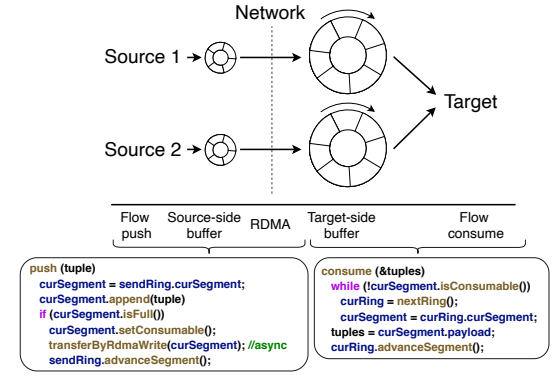


Figure 4: DFI flow implementation using ring buffers. In DFI flows, each source allocates a private target-side ring buffer to minimize coordination overhead.

5.1 Flow Execution

The key design principles listed in Section 3 impose challenges for how the data transfer between the sources and targets is realized which are pivotal for distributed data processing. In the following, we give a brief overview of the flow execution paired with the design principles.

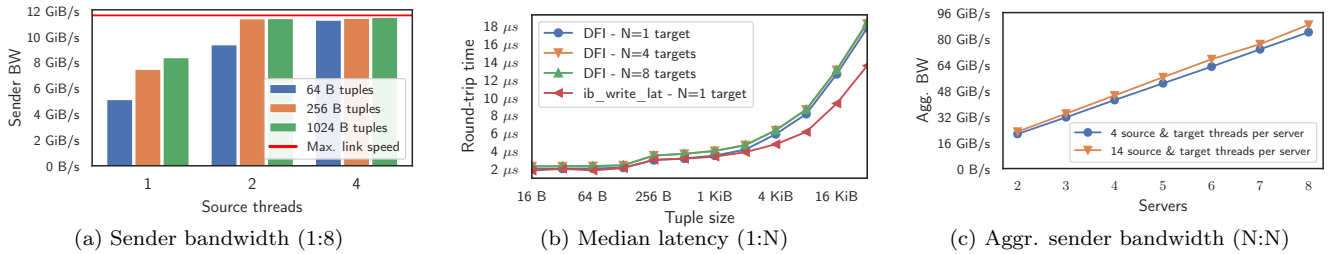
On a high-level, DFI uses a private send/receive buffer for each pair of source and target threads as illustrated in Figure 4. The design of source- and target-side buffers follows a ring-based design where each ring is composed of a configurable number of segments and is allocated as one consecutive RDMA-enabled region in memory. The segment itself can be sized to contain a single tuple up to a batch of tuples. Therefore, the segment size is a tuning parameter that allows DFI to either optimize for bandwidth or latency independent of the tuple sizes used by the application.

One key question is how such a segmented ring design enables pipelining of tuples with low-overhead synchronization. By having private send/receive buffers for each pair of sources and targets, no synchronization or coordination is needed between, e.g., multiple sources writing to the same target. As such DFI can effectively scale-out to many sources and targets without negatively impacting the performance. In order to achieve pipelined data transfer between buffers (i.e., a decoupling of senders and receivers), one-sided RDMA writes are used to copy data asynchronously from sources to targets. This asynchronous data transfer using RDMA writes is implemented by the `transferByRdmaWrite` call in Figure 4. This method also implements the synchronization with the target buffer to not overwrite any segments that has not been consumed yet. The synchronization is based on a credit approach where sources only have to synchronize with target buffers once the credit is used up.

6 Experimental Evaluation

We now evaluate DFI by first looking at the efficiency of DFI in terms of how well the high-level interface utilizes the network compared to low-level RDMA verbs. Subsequently we look at two typical use cases in data processing systems and compare the implementations to existing state-of-the-art solutions.

In all experiments we use the notation (N:M) to indicate the number of servers involved in a flow topology. The num-



(a) Sender bandwidth (1:8) (b) Median latency (1:N) (c) Aggr. sender bandwidth (N:N)

Figure 5: Shuffle flow performance. DFI achieves max. bandwidth and low latency for various scenarios.

ber of threads per server is reported separately per experiment.

Evaluation Environment: All experiments were conducted on an 8 node cluster where 6 of the nodes are equipped with two Intel(R) Xeon(R) Gold 5120 CPUs (14 cores) and 512 GB main-memory, and 2 nodes equipped with two Intel(R) Xeon(R) Gold 5220 CPUs (18 cores). Hyper-threading is disabled for all nodes. Each node is equipped with two Mellanox ConnectX-5 MT27800 NICs (InfiniBand EDR 4x NICs, 100 Gbps), connected to one SB7890 InfiniBand switch. The operating system is Ubuntu 18.04.1 LTS, with Linux 4.15.0-47 kernel on all nodes. DFI is implemented with C++17 and compiled with gcc-7.3.0.

6.1 Experiment 1: Efficiency of DFI

The first experiment shows the efficiency of DFI compared to low-level RDMA verbs. In the following, we evaluate the shuffle flows with bandwidth and latency optimization and lastly, a scale-out experiment is presented. For an extensive evaluation of the flows available in DFI, see full paper version [20].

Bandwidth-Optimized: Our first experiment evaluates performance for the shuffle flow from 1 server to 8 servers with varying tuple sizes. Further, we vary the number of sources (threads) pushing tuples into the flow. The batch size for the bandwidth optimized version in our experiments is 8 KiB. We choose a batch size of 8 KiB as this offers a good trade-off between network bandwidth and time until the batch is filled.

Figure 5a reports results for the bandwidth-optimized flow. As we see, in most settings we achieve the full network bandwidth. Only, the single-threaded scenario shows some overhead since batches must first be filled on the source side with individual tuples before they can be transferred to the target. This overhead can, however, be amortized by using more threads per server as shown in Figure 5a. Due to the efficient multi-threading support of DFI, we see that from two source threads on, the bandwidth is limited by the speed of the outgoing link (100 Gbps / 11,64 GiB/s - red line) for tuple sizes larger than 128 B. Moreover, when using 4 threads the maximal bandwidth is achieved independent of tuple sizes.

Latency-Optimized: We additionally evaluated the shuffle flow that implements latency optimizations. For measuring latency, two shuffle flows are used to implement a request and response pattern to measure the round-trip time between two nodes. To show that DFI's buffer design only adds minimal latency overhead, we compare the latency of DFI to *ib_write_lat*[2] which is a standard tool for performance testing that uses low-level verbs to implement a round-trip between a sender and a receiver node. For DFI, we additionally used a varying number of receiving servers (1, 4, and 8) to observe the effect on latency when shuffling to various destinations.

As we see in Figure 5b, the median latency of DFI for one full round-trip only adds minimal overhead when compared to *ib_write_lat* which is due to the intermediate copy to the buffer. Moreover, keep in mind that DFI provides a high-level abstraction and thus not only reduces application complexity but also provides several optimizations to applications. This includes an efficient overlapping of compute and communication as well as many other optimizations such as efficient replication and ordering guarantees. As we show in Section 6.2, this enables DFI to provide superior performance in different use cases when compared to existing approaches that are using other interfaces (low-level RDMA verbs or MPI).

Moreover, the advantage of DFI compared to plain RDMA is the encapsulated memory management, which allows applications to use RDMA transparently without hand-tuned memory management while still achieving optimal performance. The experiment shows that this abstraction hardly incurs any overhead compared to *ib_write_lat*. For multiple targets the latency of DFI is only slightly higher due to the internal routing in the shuffle flow. Multiple targets are not supported by *ib_write_lat* though (i.e., *ib_write_lat* uses only one target in this experiment).

Scale-out: Since data processing systems often need to scale out to many nodes, we conducted a scale-out experiment for the shuffle flow, increasing the number of source and target servers. Moreover, we use 14 sources and targets on all nodes which in total gives 12544 unique source/target connections for the maximal number of nodes used. As shown in Figure 5c, DFI scales linearly with the number of nodes (as indicated by the x-axis), effectively increasing the aggregated bandwidth with the link-speed of each added node.

Key Insights (Exp. 1): Our results show, that DFI flows can provide a high-level abstraction with no or only negligible overhead compared to low-level RDMA verbs.

6.2 Experiment 2: Use Cases

We now evaluate DFI by implementing the two use cases we discussed in Section 4.2.

6.2.1 *Distributed Joins*

Distributed joins are crucial operators in OLAP due to large amounts of data having to be transferred across the network, and therefore a good candidate to evaluate bandwidth-optimized flows of DFI.

[2] `https://github.com/linux-rdma/perftest`.

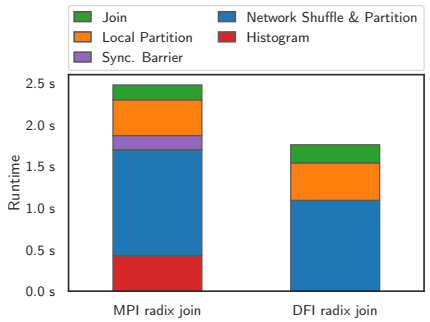


Figure 6: Dist. radix join - 8 nodes, 64 threads (DFI)/64 processes (MPI) in total. 2.56 B ⋈ 2.56 B tuples.

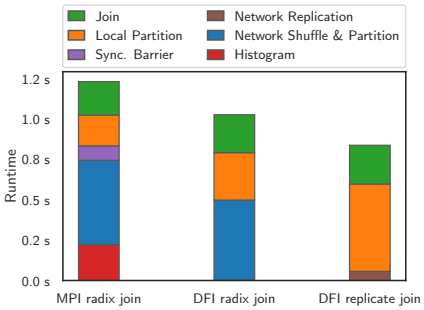


Figure 7: Dist. joins - 8 nodes, 64 threads (DFI)/64 processes (MPI) in total. 2.56 M ⋈ 2.56 B tuples.

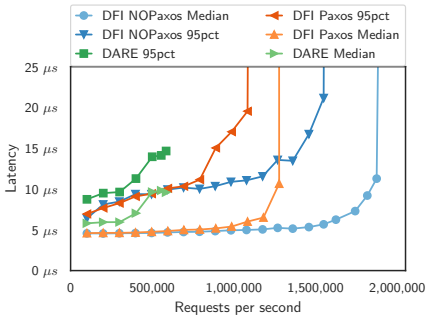


Figure 8: Performance comparison of DARE [19] with DFI-based implementations of Multi-Paxos and NOPaxos.

Radix Join: We implemented a distributed radix hash join on DFI and compared its performance to a state-of-the-art implementation for RDMA using MPI [2]. Both implementations employ the same optimizations (e.g., write-combine buffer in partitioning phase and tuple compression). However, the MPI join of [2] uses multi-process parallelism while our join uses multi-threading instead. Figure 6 shows the average runtime of the two joins for all 8 nodes.

The DFI radix join achieves the best runtime mainly due to two design choices of DFI. At first, the DFI radix join does not need to first compute a global histogram of the partition buckets. The MPI radix join in [2] makes use of one-sided *MPI_Put* primitives. In order to achieve coordination free writes, it thus has to compute exclusive writing offsets for each partition using one additional pass. Different from this, DFI encapsulates the memory management through our buffer design which makes the additional pass superfluous.

The other reason for the runtime gap is due to the synchronization barrier needed in the MPI radix join after the network partition phase. Here, the join algorithm needs to make sure that all data has arrived before starting to process the local partitioning. While the data in this experiment is uniformly distributed, some runtime variance between multiple parallel workers still exists and is more pronounced in high-speed networks. This synchronization is not needed with DFI, since incoming tuples can already be processed when they arrive in a streaming-wise fashion.

Join Adaptability: Flows in DFI offer a high-level abstraction which encapsulates the data transfer of applications. As a result, it is trivial to adapt algorithms to use a different communication pattern. To demonstrate this, we adapted our radix hash join implementation to a fragment-and-replicate join variant which uses one replicate flow that replicates the inner table on all nodes. Figure 7 shows the runtimes of the three different join implementations with a smaller inner table (1000× smaller than the outer table). The replication of the small inner table is comparably cheap compared to shuffling the big outer table over the network. Overall, for this setup this helps to further reduce the overall runtime by another 20%,

6.2.2 State Machine Replication

In this experiment, we implemented a simple key-value store that replicates data using a consensus protocol. For the experiment, we used two different consensus protocols, classical Multi-Paxos [14] and NOPaxos [16]. We modeled the normal, failure-free operation of Multi-Paxos as depicted in Figure 3. For NOPaxos, we implemented its *normal operation* protocol, which relies on the OUM primitive that can be provided by DFI's replicate flow, as well as its *gap agreement* protocol to detect lost messages. We compare both implementations with DARE [19], a state-of-the-art replicated key-value store that is based on a hand-crafted consensus protocol and heavily relies on one-sided RDMA.

We deployed all approaches with five replicas (a leader and four followers). Load was generated by six clients distributed across three separate nodes. Clients submitted 64 byte sized requests using YCSB's read-dominated workload [6] (95% reads and 5% writes). The results are shown in Figure 8.

The two DFI-based implementations consistently outperform DARE in our settings in both achieved throughput and latency. This is caused mainly by DARE's sequential design. First, each DARE client cannot submit a new request until it has received the result from its previous request, which limits its achievable throughput.

Second, DARE's write protocol serializes requests. While this limitation is mitigated by separately batching reads and writes, a mix of both request types frequently interrupts batches [21]. This is confirmed by DARE's own evaluation [19].

Our Multi-Paxos and NOPaxos implementation exhibit near-identical response latencies as long as they are not saturated. This appears counter-intuitive at first, as Multi-Paxos requires four message delays to respond to a client, whereas two messages delays suffices for NOPaxos as long as no messages are lost. However, fetching a global sequence number from the tuple sequencer of the ordered replicate flow incurs an additional two message delays.

For a load higher than 700k requests/s, we see benefits of our NOPaxos over our Multi-Paxos implementation. Under this load, the leader in Multi-Paxos becomes saturated as it has to repeatedly collect responses from a majority of replicas. In contrast, in NOPaxos the clients themselves collect these responses. This alleviates the burden placed on the leader in Multi-Paxos, which leads to stable response latencies in DFI's NOPaxos up to even higher request rates of almost 1.5M (95th percentile).

Key Insights (Exp. 2): In summary, DFI does not only achieve a better performance for distributed joins and consensus than state-of-the-art, but also offers an ease-of-use high-level abstraction to implement efficient solutions with a low code complexity.

7 Using DFI

We provide DFI as an open-source project.[3] In the repository, guides can be found detailing how to set up and include DFI, either as a git submodule or installation, along with small examples showing the setup and usage of DFI flows.

By open-sourcing our implementation, we hope to stimulate not only follow-up research but also any form of contributions and invite any interested parties to collaborate and build DFI further. Additionally we hope that commercial vendors will contribute and provide a DFI implementation also for other high-speed network stacks, similar to the collaborative effort of MPI.

8 Conclusions

In this paper, we presented DFI, a new data-centric interface for fast networks. With our implementation for InfiniBand we have shown that DFI adds only minor overhead compared to low-level abstractions such as RDMA verbs. Moreover, by implementing two use cases, we demonstrated that DFI can efficiently support data-centric applications with different requirements (high-bandwidth vs. low-latency) at high performance.

In future, we plan to integrate further useful extensions into DFI flows such as fault-tolerance as well as elasticity of flows to add/remove nodes at runtime. Additionally, we aim to extend DFI flows to support other compute architectures such as GPUs and DPUs.

9 Acknowledgments

This work was partially funded by the German Research Foundation (DFG) under the grants BI2011/1 & BI2011/2 (DFG priority program 2037), the DFG Collaborative Research Center 1053 (MAKI) as well as gifts from Mellanox and Huawei.

10 References

[1] G. Alonso, C. Binnig, I. Pandis, K. Salem, J. Skrzypczak, R. Stutsman, L. Thostrup, T. Wang, Z. Wang, and T. Ziegler. DPI: the data processing interface for modern networks. In *CIDR*, 2019.

[2] C. Barthels et al. Distributed join algorithms on thousands of cores. *PVLDB*, 10(5):517–528, 2017.

[3] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-scale in-memory join processing using rdma. In *SIGMOD*, page 14631475, 2015.

[4] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It's time for a redesign. *PVLDB*, 9(7):528–539, 2016.

[5] Q. Cai, W. Guo, H. Zhang, D. Agrawal, G. Chen, B. C. Ooi, K. Tan, Y. M. Teo, and S. Wang. Efficient distributed memory management with RDMA and caching. *PVLDB*, 11(11):1604–1617, 2018.

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.

[7] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson. Farm: Fast remote memory. In R. Mahajan and I. Stoica, editors, *NSDI*, pages 401–414, 2014.

[8] P. Fent, A. van Renen, A. Kipf, V. Leis, T. Neumann, and A. Kemper. Low-latency communication for fast DBMS using RDMA and shared memory. In *ICDE*, pages 1477–1488, 2020.

[9] P. W. Frey and G. Alonso. Minimizing the hidden cost of RDMA. In *ICDCS*, pages 553–560, 2009.

[10] W. Gropp et al. *Using Advanced MPI: Modern Features of the Message-Passing Interface*. The MIT Press, 2014.

[11] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *SIGCOMM*, pages 295–306, 2014.

[12] A. Kalia, M. Kaminsky, and D. G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *OSDI*, pages 185–201, 2016.

[13] S. J. Kang, S. Y. Lee, and K. M. Lee. Performance comparison of openmp, mpi, and mapreduce in practical problems. *Advances in Multimedia*, 2015.

[14] L. Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[15] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. Kv-direct: high-performance in-memory key-value store with programmable NIC. In *SOSP*, pages 137–152, 2017.

[16] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In *OSDI*, pages 467–483, 2016.

[17] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In *ATC*, pages 103–114, 2013.

[18] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In *ATC*, pages 305–319, 2014.

[19] M. Poke and T. Hoefler. DARE: high-performance state machine replication on RDMA networks. In *HPDC*, pages 107–118, 2015.

[20] L. Thostrup, J. Skrzypczak, M. Jasny, T. Ziegler, and C. Binnig. DFI: the data flow interface for high-speed networks. In *SIGMOD*, pages 1825–1837, 2021.

[21] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui. APUS: fast and scalable paxos on RDMA. In *SoCC*, pages 94–107, 2017.

[22] J. Xue, Y. Miao, C. Chen, M. Wu, L. Zhang, and L. Zhou. Fast distributed deep learning over RDMA. In *EuroSys*, pages 44:1–44:14, 2019.

[23] E. Zamanian, J. Shun, C. Binnig, and T. Kraska. Chiller: Contention-centric transaction execution and data partitioning for modern networks. In *SIGMOD*, page 511526, 2020.

[24] T. Ziegler, V. Leis, and C. Binnig. Rdma communciation patterns. *Datenbank-Spektrum*, 20(3):199–210, Nov 2020.

[25] T. Ziegler, S. Tumkur Vani, C. Binnig, R. Fonseca, and T. Kraska. Designing distributed tree-based index structures for fast rdma-capable networks. In *SIGMOD*, page 741758, 2019.

[3] `https://github.com/DataManagementLab/DFI-public`