FMI: Fast and Cheap Message Passing for Serverless Functions

Marcin Copik[1], Roman Böhringer[1] Alexandru Calotoiu[1] Torsten Hoefler[1]
[1]Department of Computer Science, ETH Zurich

***Abstract*—Serverless functions provide elastic scaling with a fine-grained billing model, making Function-as-a-Service (FaaS) an attractive programming model. However, for distributed jobs that benefit from large-scale and dynamic parallelism, the lack of fast and cheap communication is a major limitation of serverless computing. Individual functions cannot communicate directly, group operations do not exist, and users resort to manual implementations of storage-based communication. This results in communication times multiple orders of magnitude slower than those found in HPC systems. We overcome this limitation and present the FaaS Message Interface (FMI). FMI is an easy-to-use, high-performance framework for general-purpose point-to-point and collective communication in FaaS applications. We support different communication channels and offer a model-driven channel selection according to performance and cost expectations. We model the interface after MPI and show message passing can be integrated into serverless applications with minor changes, providing portable point-to-point and collective communication closer to that offered by high-performance systems. In our experiments, FMI can speed up communication for a distributed machine learning FaaS job by up to 162x while at the same time reducing cost by up to 941 times.**

I. Introduction

Function as a Service (FaaS) is an emerging programming paradigm popular in cloud applications. In FaaS, users focus on writing application code decomposed into a set of functions. Users are not concerned with deploying code and managing the underlying compute and storage infrastructure. Instead, function invocations are executed by the cloud provider on dynamically provisioned servers. Thus, users never allocate servers (*serverless computing*) and are charged only for computing time and memory resources used (*pay–as–you–go billing*). Small, stateless functions do not need to communicate – they simply write their results to storage, and future functions can continue from there. Thanks to the fine-grained billing and execution model, functions have become a popular programming model for irregular and unbalanced workloads.

Functions are used for distributed and stateful computations in data analytics, linear algebra, processing of multimedia, and machine learning [1–7]. These workloads benefit from the fast and cheap scalability of ephemeral function workers. However, these functions must run for a longer time and have a significant internal state. This makes storing the entire state and continuing execution later once new inputs become available inefficient — they need a cheap and fast way of exchanging data to become an efficient backend for distributed computations, but they lack a native and high-performance communication interface. Furthermore, between their reliance on accessing remote data and their statefulness such workflows rely on simultanous invocation of multiple functions and are significantly penalized for failures.

In HPC, communication in a distributed system is done using the Message Passing Interface (MPI). In contrast to virtual machines and HPC applications, functions execute in sandboxes that provide strict isolation but are prevented from accepting incoming network connections (Sec. III-B1). To communicate, functions rely on in-memory caches and storage optimized for serverless functions — these are primarily designed to improve performance [8] but introduce a user-managed and persistent component that defeats the purpose of serverless computing. Instead, users need a flexible choice between fast and cheap network communication and slower, more expensive, and more durable storage-based exchange. Unfortunately, while serverless computing is advertised as an elastic solution for computing and resource allocations, it is surprisingly inflexible when it comes to communication. The performance and price of serverless messaging is already a pressing problem, as messages exchanged over object storage come with double-digit millisecond latency and cost of $6 per million. The situation is even worse if collective communication is considered.

The importance of collective operations has been noticed very early on in the HPC community [9, 10], and they are used in virtually all MPI jobs [11]. Collectives provide a portable interface for standard parallel programming patterns [12]. Replacing send–receive messages with collectives makes applications simpler and easier to program, debug, and maintain while retaining the expressiveness and performance of direct messages [13]. From the user's point of view, collectives provide *"division of labor: the programmer thinks in terms of these primitives and the library is responsible for implementing them efficiently"* [14]. This separation is even more important in the black-box serverless world with major differences between cloud providers.

At the same time, high performance communication requires finely tuned algorithms according to network topology, number of participants, message size, application, and even the memory hierarchy [15–20]. However, the entire communication hierarchy that includes nodes, racks, sockets, processes, and caches is hidden from the user in serverless. This provides an additional motivation for the cloud provider to implement hierarchical and multi-protocol communication [21], such that serverless applications benefit from standardized message

passing operations with high-performance implementations adjusted to the system at hand. The world of collective specializations is rich and remains concealed behind the system abstractions, and serverless should benefit from it (Sec. III-C).

The community has already identified the lack of support for efficient group communication as one of the fundamental limitations of serverless computing [1, 22, 23]. The interface of collective operations should allow for incorporating system-specific optimizations without breaking the serverless abstraction layer, e.g., including point–to–point communication and adapting hierarchy to task affinity [22]. Serverless applications can benefit from the high performance and versatility of collectives, but need a framework that hides the complexity of the cloud system.

In this work, we provide the first direct general-purpose communication framework for FaaS: the **FaaS Message Interface (FMI)**. *FMI* is an easy–to–use and high-performance framework where the implementation details of point-to-point and collective operations are hidden behind a standardized interface inspired by MPI, providing portability between clouds and runtime adaption. We use MPI as our guide as it has established itself as *the* communication solution for distributed memory systems. We have implemented and extensively evaluated multiple communication channels with respect to both price and performance and they are all included in the current library implementation. Having determined that direct communication over TCP is the best solution in all scenarios, we also implement a general-purpose TCP hole punching solution to allow functions to communicate directly, even behind NAT gateways. While we implement *FMI* on AWS Lambda, the design of our library is independent of the cloud provider and can be ported to any serverless service.

Concretely, we make the following contributions:

- We introduce a library for message passing providing common and standardized abstractions for serverless point-to-point and group communication.
- We provide analytical models for communication channels in FaaS and discuss the performance-price trade-offs of serverless communication.
- We demonstrate the application of *FMI* to serverless machine learning and present a reduction of communication overhead by a factor of up to 162x and reducing cost by up to 941 times compared to existing solutions.

II. Background

Distributed FaaS applications already implement many group and collective operations patterns across concurrent functions, prominent examples being MapReduce in data analytics [3, 24–26], `reduce-scatter` in machine learning [1, 4], and `scan` in video encoding [7].

However, the inter-function communication remains the Achilles' heel of serverless. Inspired by the statement: *"Storage is not a reasonable replacement for directly-addressed networking, even with direct I/O —it is at least one order of magnitude too slow."* [27], we list different possible communication channels and in the following sections, we conduct a detailed performance (Sec. IV) and cost analysis of these cloud systems (Sec. V):

- **Object Storage.** These systems offer persistent storage for large objects with high throughput, strong consistency [28], data reliability [29–31], and a cost linear in the number of operations and size of stored data.
- **Key-Value Storage.** NoSQL databases offer low-latency and high-throughput scaled to the workload [32, 33]. However, these support only small objects (e.g., 400kB in DynamoDB [32]) and have high costs for write operations.
- **In-Memory and Hybrid Storage.** In-memory stores such as Redis [34] and memcached [35] offer higher performance at the cost of manual scalability management by the user and non-serverless resource provisioning. Serverless-optimized storage use multiple tiers of memory and disk storage [8]. The costs depend on the size of the memory store and time it remains in use.
- **Direct Communication.** A direct network connection could offer higher performance than storage solutions without incurring any costs. We discuss a prototype implementation in Sec. III

Today's serverless functions tend to communicate using cloud proxies for messaging. Functions cannot establish direct connections which would provide higher performance at lower cost.

III. FaaS Message Interface

In this section, we discuss **FMI**, the FaaS Message Interface for point–to–point and collective communication as well as the assumptions made by our approach. We then discuss communication channels we tested as possible options for FMI (Sec. III-B). We model the interface of FMI after the proven and tested interface of MPI (Sec. III-E) and implement a selection of the most common collective operations in serverless applications (Sec. III-C). We design FMI to be modular - our design does not make any assumptions on the underlying cloud system. This is especially important because we target cloud systems that change quickly and often contain proprietary components that cannot be reused across platforms. FMI can be extended with new communication channels, collective operations, and support for programming languages other than C/C++ and Python.

A. Assumptions

a) Isolation of serverless functions: Small, stateless functions work well in isolation by writing their results to storage and can be scheduled independently from each other. However, the type of functions used in more complex serverless workflows [1–7] are neither stateless nor independent of each other, requiring complex task dependencies. We therefore dispense with the assumption that FaaS functions should be considered in isolation — as the evolving nature of the serverless computing field already makes it deprecated in many practical scenarios.

b) Simultaneous scheduling: In FMI, we assume that all functions that will be part of the same communication entity, or communicator, can be scheduled simultaneously. As soon as the first function joins the communicator a timer is started. If all functions that are scheduled to join do not do so before the timer expires, the all functions exit with an appropriate error.

c) Fault tolerance: In FaaS, it is possible to retry individual functions on failure. In FMI there is no recovery mechanism for individual members of a communicator, and if a function fails or a communication times out the entire communicator will exit with an error. However, a user could implement fault tolerant policies on top of FMI, similar to approaches used in MPI [].

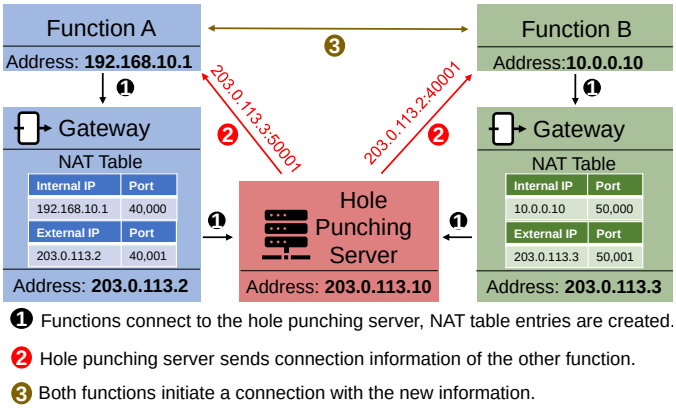


Fig. 1: **Network Address Translation (NAT) Hole Punching.**

B. Communication Channels

While communicators are responsible for data conversion and serialization, channels are the medium for data exchange and operate on raw memory. We broadly classify them in *direct* and *mediated* channels. In mediated channels, the communication between participants is done over storage or other indirect means.

We provide implementations of both types. Mediated channel examples are object storage (AWS S3), key-value database (DynamoDB), in-memory cache (Redis), and we create direct channels using TCP connections. FMI could be extended with UDP transport protocol by using QUIC to provide reliability and security [36]. Indeed, programmers can add new channels to the library with little effort and immediately benefit from other existing FMI features - such as the implementations of collective operations. However, if a channel provides more specialized mechanisms, such as support for the reduction operation, these can be added by overriding default collective algorithms.

Ideally, cloud providers would provide direct communication between functions as a service, but until such time, we provide the a method that allows direct communication using TCP in the current serverless ecosystem. We first summarize communication over Network Address Translation (NAT), highlight the obstacles towards using it in the serverless ecosystem, and suggest hole punching as a solution.

1) Network Address Translation (NAT): Function instances are placed in sandboxes behind a NAT gateway [37]. The gateway hides the endpoint by rewriting the internal address with an external one in packet headers. An outgoing communication creates an entry in the translation table. This enables replies sent to the external address to be forwarded to the intended recipient. Packets are dropped when there is no entry in the translation table, therefore the party initiating the communication can be behind a NAT gateway, but not the recipient. Thus, when both parties are behind a NAT gateway, direct communication is not possible.

2) Hole Punching: One technique to circumvent the restricted direct communication for endpoints behind a NAT is hole punching [38, 39]. This approach relies on a publicly reachable relay server to create mappings in the translation table and exchange the other party's address with each participant. Then, both participants attempt to connect roughly at the same time using the existing address mappings from the previous step (Fig. 1).

In our implementation of direct channels, we observed unpredictable delays of up to 40 ms, which we suspect are caused by Nagle's algorithm [40]. We therefore disable the algorithm on sockets used for direct communication by default, while allowing for it to be enabled if the performance improvements outweigh the delays.

C. Collective Communication

Decades of research into collective operations have led to many optimized communication protocols. Collective algorithms have different time, memory, and energy trade-offs [41]. When communication via cloud storage is considered, the cost of operations becomes another fundamental characteristic and limitation. Modern collective operations are extensively tuned: MPI collectives are specialized for network transport protocols [42–45], network topology [46–50], and even for specific needs of applications, such as bandwidth and sparsity optimizations in machine learning [19, 51–53].

Cloud providers must be the ones to apply such optimizations as the abstraction layer prevents users from understanding the system's architecture, and the opportunities for improvement are no less complex than for MPI: Serverless heterogeneity is increasing with RMA [54] as well as GPUs [55, 56], and the dynamically changing topology of workers presents additional challenges [57–59].

We implement the following collective operations from the MPI standard [60]: `broadcast`, `barrier`, `gather`, `scatter`, `reduce`, `allreduce`, and `scan`. The algorithms selected differ depending on the communication channel used While we provide a prototype implementation of algorithms, these should be modified and updated according to the user's needs and the cloud system configuration. For example, the runtime of the `scan` operation can be improved by using a depth-optimal but work-inefficient algorithm [61] but is impractical on channels with a high data movement cost. Furthermore, dedicated collective operations can be im-

plemented for highly dynamic task-based applications where the number of function workers frequently changes [62].

Mediated channels. In `broadcast`, the root process uploads the object to the object storage, and other functions download it, benefiting from the scalable bandwidth of the storage. In `barrier`, each function uploads a 1-byte object and polls until all data is available. In the context of collective operations implemented over storage, polling is implemented using the *list* operation on the storage – this counts the number of objects, and succeeds when the count equals the number of functions in the communicator. In `gather`, functions upload their buffers to storage and the root node polls for data while `scatter` follows an inverted communication pattern. Similarly, in `reduce` and `allreduce`, the root node is responsible for downloading data and applying the reduction. Finally, in `scan` each function polls for the partial result generated by its predecessor, applies the scan operator and uploads the result.

Direct channels. This case is similar to the MPI use-case, so `broadcast`, `gather`, `scatter`, and `reduce` are implemented with a binomial tree to avoid the bandwidth limitations of a single function. `Allreduce` uses recursive doubling [15], `barrier` is implemented as an `allreduce` with one-byte input and the no-op reduction operator and `scan` is implemented with a two-phase tree-based operation [63, 64].

D. Implementation

The FMI library has been implemented in roughly 1,900 lines of C++ code. The open-source framework is available as prebuilt Docker images with necessary dependencies. Furthermore, we provide *Infrastructure–as–a–Code* in the form of AWS Lambda layers and CloudFormation templates [65]. However, our approach is platform agnostic and should run in any serverless environment such as KNative without changes. FMI users can use layers to integrate the message-passing library into serverless applications without any build steps. Furthermore, we implement a hole-punching library and server `TCPunch` as to the best of our knowledge, there is no open-source solution for C/C++ available. `TCPunch` supports hole punching for full cone, restricted cone, and port restricted cone NAT implementations. The library does not contain any FMI-specific logic and exposes a simple interface (Listing 1), allowing other applications to integrate TCPunch [1].

Listing 1 Example usage of the TCPunch client library.

```
#include <tcpunch.h>
// Client 1
int sock_fd = pair("faas_job_key",
  "hole_punch_server_addr");
int n = send(sock_fd, pBuf, size, 0);
// Client 2
int sock_fd = pair("faas_job_key",
  "hole_punch_server_addr");
int n = recv(sock_fd, recv_buffer, size, 0);
```

[1]For details on the source code and configuration of hole punching and message passing, we refer readers to a technical report: [anonymized]

E. Interface

The FMI interface is heavily inspired by MPI, helping programmers familiar with MPI use the library without adjustment. The interface is designed with high degree of compatibility: we primarily extend the MPI interface with modern C++ features, e.g., we remove the need for explicit typing in many operations (Listing 2).

Listing 2 Example usage of the FMI C++ interface.

```
#include <fmi.h>
// The functions are part of a communicator comm
// Here, the communicator contains 3 functions
// Each function has an unique id: 0,1,2
// Defining send buffer:
FMI::Comm::Data<std::vector<int>> vec({0, 1, 2});
// Defining receive buffer:
FMI::Comm::Data<std::vector<int>> recv(1);
// Collective operation
comm.scatter(vec, recv, 0);
// Test that each function got the correct data
assert(recv.get()[0] == my_id);
```

As in MPI, all message-passing operations are based on the concept of a communicator. Each communicator is uniquely named and is based on a *group* of N FaaS functions, each one with a unique identifier in the range $[0, N)$ [66]. Therefore, an application can create multiple communicators with different numbers of peers, different lifetimes and providing the flexibility needed to support the many communication patterns of serverless. For collectives that reduce data, such as `(all)reduce` and `scan`, users can provide an arbitrary function object to be used as a reduction operation.

Concurrent invocation of multiple parallel functions can be implemented in existing FaaS systems, and we envision that future serverless runtimes will provide such an option natively.

Languages. Support for new languages can be easily added to the system by implementing a wrapper around the communicator library. We demonstrate the support for Python, a language popular in serverless, with the help of the `Boost.Python` library (Listing 3).

IV. Communication Channel Performance

An ideal communication channel for serverless functions should support both **low latency** and **high throughput** communication. While many cloud technologies can be used for serverless communication, none of them fulfills all of the requirements (Tab. I).

Listing 3 Example usage of the FMI Python interface.

```
import fmi
// The functions are part of a communicator comm
// Here, the communicator contains 3 functions
// Each function has an unique id: 0,1,2
// Defining a datatype:
dtype = fmi.types(fmi.datatypes.int)
// Root function sends data:
if my_id == 0:
    comm.bcast(42, 0, dtype)
// All other functions receive data:
else:
    assert comm.bcast(None, 0, dtype) == 42
```

Channel	Latency	Bandwidth	Cost	Scalability	Max. Message	Push vs Pull?	Message Persistence	Serverless?
Object Storage	Very High	Low	Low	Provider-side	5 TB	Pull	✓	✓
NoSQL Database	High	Very Low	High	Provider-side	400 kB	Pull	✓	✓
In-Memory Cache	Low	High	Low	User-side	512 MB	Pull	–	✗
Direct TCP	Very Low	High	Free	User-side	Unlimited	Push	✗	✓

TABLE I: Serverless communication channels. In the following sections, we quantify the differences and characterizations.

Additional requirements for serverless storage include **elastic scaling** with serverless parallelism and efficient support for **arbitrary object sizes** [67]. The latter is necessary to support the different communication patterns of serverless applications that can involve both fine-grained messaging and exchanging large data objects. Furthermore, serverless storage can offer message persistence for additional fault tolerance. In-memory caches can access some past messages while the instance is running, but will be lost upon releasing the resource.

An aspect to consider is that not all communication channels support *push* messages — messages where the receiver blocks and waits for the data to arrive. Instead, the intended recipient must actively and frequently poll the channel to verify if the expected message is available — also known as *pull* messages. Active polling introduces additional complexity (Sec. IV-A) and adds an important performance-cost trade-off (Sec. V).

Finally, not all systems support a truly serverless deployment where no resource provisioning by the user is required. Estabilishing direct communication channels should be offered as a service for serverless by the cloud provider. Until such a service is available, hole punching can be used as a viable alternative. Many tenants can use hole punching simultaneously, and the service scales horizontally according to the traffic. Therefore, while direct communication currently requires deploying a hole punching server, the server requires minimal resources as its only responsibility is to accept connection requests.

On the other hand, setting up the Redis cluster requires a significant amount of work, and right-sizing the cluster is the user's responsibility. The system traffic must be monitored since an underprovisioned Redis cluster will not lead to failures but instead cause performance degradation, complicating server management further.

To understand the performance implications of selected cloud communication channels, we consider two scenarios: With a single sender and a single receiver, we examine *point–to–point* communication, the basic building block of all communication operations (Sec. IV-B). Then, we consider a *one-to-many* scenario with one sender and a variable number of receivers. This benchmark intentionally stresses bandwidth scalability of the different channels (Sec. IV-C), so we explicitly do not use algorithmical optimizations in this step.

A. Benchmarking Setup

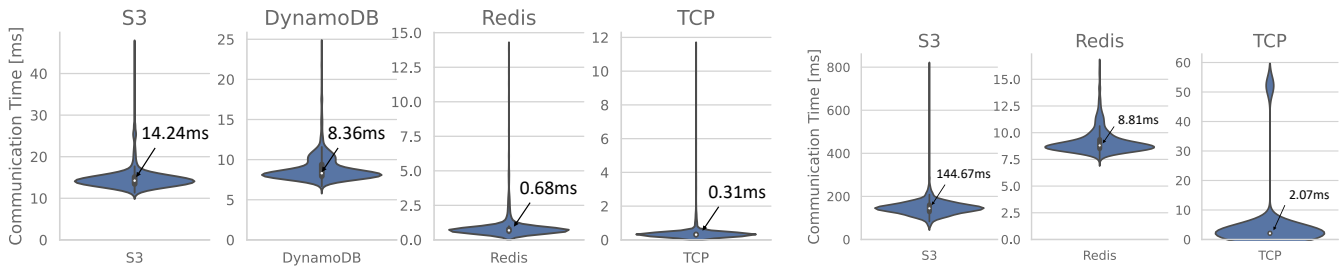
We analyze the following communication channels in the AWS cloud: S3 (object storage), ElastiCache Redis (in-memory data store), DynamoDB (NoSQL key-value store), and direct TCP communication with NAT hole punching. For all of them, we implement the message exchange in serverless Lambda functions written in C++. We assign 2 GiB of RAM to Lambda functions to decrease the likelihood of functions' co-location in a single virtual machine [68], and we run the experiments in the cloud region `eu-central-1`.

The S3 and DynamoDB stores do not require additional configuration beyond creating cloud resources. We use the pay–as–you–go billing model for DynamoDB, and we deploy Redis on the `cache.t3.small` instance with 1.37 GiB RAM and two vCPUs. The hole punching server requires little resources as it only needs to store a few bytes per connection during the setup, and we deploy it on `t2.micro` instance (1 GiB RAM, one vCPU). This comes with a cost of slightly less than 1.5¢ per hour.

Polling. To communicate over S3, DynamoDB, or Redis, the producer creates an object or an item in a predetermined location. Unfortunately, there is no explicit notification mechanism to inform the consumers that data is available. It is possible to launch new functions in an asynchronous manner on data update, but one cannot notify an existing function. For small, short running functions this provides no issue. However, for large stateful functions repeatedly stopping and resuming is inefficient. Therefore, consumers need to repeatedly poll the store using the predetermined key until they get a successful response. We implement a hybrid backoff strategy to reduce the number of required GET operations, as each one million of reads costs approximately $0.5. For the first 100 retries, the backoff time is linearly increased from 1 ms to 100 ms, resulting in relatively short communication times for small files. Afterward, we set the backoff time to 2 times the number of retries, i.e., 202 ms, 204 ms, and bound the maximum number of retries to 500. This backoff strategy is not necessary in the ElastiCache Redis implementation as here polling does not incur additional costs.

B. Point–to–Point

To measure point–to–point communication, we execute a ping-pong benchmark and report half of the round-trip time. For storage-based communication, the time includes both put and get requests. For small messages (Fig. 2a), inter-function TCP can achieve microsecond latency. For large messages (Fig. 2b), direct communication over TCP is the fastest option with a reasonably symmetrical density, concentrated around the mean. The times for Redis follow a similar distribution around a higher mean. S3 has a higher median communication time, with values not symmetrically centered around the mean. The distribution shows a long right-hand tail.



(a) One byte message.

(b) 1 MB message.

Fig. 2: Point–to–point communication latency for messages with one byte (left) and 1 MB (right), 1000 repetitions.

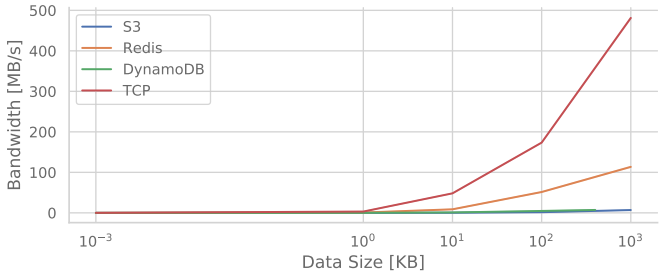


Fig. 3: Bandwidth of point–to–point communication with varying sizes.

For the next experiment, we vary message size from 1 byte to 1 MiB and present the median bandwidth (Fig. 3). Direct communication remains the fastest communication channel for all data sizes, with the difference to cloud storage being smaller for larger sizes: the relative overhead of the cloud proxy becomes smaller compared to the transmission time. While other authors have reported high bandwidths for S3 [67], our measurements include both the send–receive (put–get) communication and the overhead of polling.

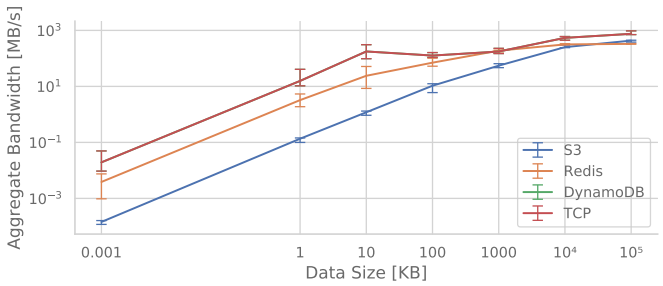


Fig. 4: Bandwidth of one-to-many communication with varying message size and 8 receivers.

C. One–To–Many

To test a one-to-many communication, we use one Lambda producer that sends messages to multiple consumers. This experiment allows us to assess bandwidth limitations and performance with multiple functions receiving from one source. In each experiment, consumers create a direct TCP connection to the producer to acknowledge the reception of a message. The producer stops time measurement upon receiving confirmation from all consumers, and we subtract the TCP latency measured in the previous benchmark (Sec. IV-B) from the result. An alternative approach would require estimating the clock drift between distributed Lambda functions [69, 70], but tuning these protocols in the noisy serverless environment is difficult.

We vary the message size, and present the aggregated bandwidth across all 8 receivers in Fig. 4. While direct communication results in the highest bandwidth, the difference to Redis and S3 decreases with the number of participants. The results are similar to the point–to–point benchmark for small data sizes. For messages larger than 10 KiB, there is a strong increase in communication time due to bandwidth limitations. Both Redis and S3 show good scaling in this benchmark, as the latter offers automatic scaling with the number of users.

We also investigated the feasibility of increasing the number of consumers beyond 64. S3 handles scalability with 128 consumers well, but we observe irregular failures on Redis with 128 and 256 consumers, likely due to resource limitations. Our hole punching server easily supported the connection setup with 256 functions, even with a `t2.micro` instance. However, the producer's bandwidth becomes the bottleneck as the number of consumers grows, highlighting the need for specialized algorithms for collective communication.

V. The Price of Performance

When modeling and designing HPC collectives [16, 71], time, memory, and energy trade-offs [41] must be considered. In the serverless world, we must also consider the price of cloud operations. The price–performance trade-off has always been a major issue in serverless [72]: allocating more powerful instances decreases computation time and resource occupancy, but it does not always lead to lower costs. Therefore we must include both the cost of data transfer and the runtime of functions spent in transmitting data. To model the time, we use the alpha-beta model — one of the simplest ways to describe parallel communication. This model considers α, the latency of the communication channel, and β, the inverse of its bandwidth . The time to send a message of size s becomes $T = \alpha + s \cdot \beta$.

To compare direct and mediated channels we consider the latency and bandwith of two functions in a point-to-

Bandwidth	Value	Latency	Value
$1/\beta$(s3)	50 MB/s	α(s3)	14.7 ms
$1/\beta$(ddb)	7 MB/s	α(ddb)	8.9 ms
$1/\beta$(redis	100 MB/s	α(redis)	0.88 ms
$1/\beta$(direct)	400 MB/s	α(direct)	0.39 ms

TABLE II: Performance model parameter values for AWS (S3, DynamoDB, ElastiCache Redis, and Lambda).

point communication. We report parameter values for AWS in Tab. II. The results show that in-memory store outperforms object storage in both bandwidth and latency, but they are both inferior to direct communication.

We consider the cost per second of executing serverless functions, hosting in-memory cache, and using cloud storage (Tab. III). We do not incorporate the fixed fee per function invocation in this analysis, as these costs are the same for each communication channel and are negligible for long-running functions.

Direct communication. As we have seen in Sec. IV-C, TCP communication has no inherent cost, but limited bandwidth – and thus the increased communication time might generate higher costs. The cost of direct communication is limited to the runtime spent in communication by participating FaaS functions, but we consider the hole punching service needed by adding the cost of the virtual machine ($p_{\texttt{hps}}$) running the service.

Mediated channels. For object storage or in-memory data stores, we define a minimal transfer time as the least time it takes for a piece of data to be written to the intermediary storage by a function and then read by another - therefore assuming there is no time spent waiting and polling for the data to become available. The actual transfer time will be longer in practice, as the functions sending and receiving are unlikely to be perfectly synchronized. This will require additional polling on the part of the receiving function, and therefore result in delays. When using object storage such as S3 or DynamoDB for communication, there are no additional infrastructure costs because the system is managed by the provider and charged on a per-use basis. We further assume that all data is ephemeral and immediately deleted after execution, which leads to negligible storage costs. We therefore only pay for the up- ($p_{\texttt{s3,u}}$, $p_{\texttt{ddb,u}}$) and downloads ($p_{\texttt{s3,d}}$, $p_{\texttt{ddb,d}}$). For communication over an in-memory data store, only infrastructure costs for the store are incurred ($p_{\texttt{redis}}$), and these only depend on how long the instance is running.

Cost of FaaS functions. The total cost of communication is the sum between the cost to run the FaaS functions during the communication and the cost of moving the data through the channel. While the function runtime is not directly influenced by the channel choice, this choice influences the communication time, which in turn determines this cost. One exchange with P participants, each with M GiB of RAM, that takes t seconds on average has a cost that can be calculated as follows:

$$c_{\texttt{function}} = P * t * p_{\texttt{faas}} * M \quad (1)$$

Item	Value ($)	Description
$p_{\texttt{faas}}$	$1.67 \cdot 10^{-5}$	Lambda GiB per s.
$p_{\texttt{hps}}$	$3.72 \cdot 10^{-6}$	`t2.micro` EC2 instance per s.
$p_{\texttt{redis}}$	$1.05 \cdot 10^{-5}$	`cache.t3.small` ElastiCache per s.
$p_{\texttt{s3,d}}$	$4.3 \cdot 10^{-7}$	S3 GET per request.
$p_{\texttt{s3,u}}$	$5.4 \cdot 10^{-6}$	S3 PUT per request.
$p_{\texttt{ddb,d}}$	$7.62 \cdot 10^{-8}$	DynamoDB read per 1kB.
$p_{\texttt{ddb,u}}$	$1.5 \cdot 10^{-6}$	DynamoDB write per 1kB.

TABLE III: Price components of the model for AWS in `eu-central-1`, US dollars.

Channel	Time (ms)	FaaS ($)	Channel ($)	Total ($)
S3	16.70	1.12	5.83	6.95
DynamoDB	151.76	10.10	1,580.00	1,590.10
Redis	10.88	0.73	0.16	0.84
Direct	2.89	0.19	0.01	0.20

TABLE IV: Price analysis for communication over S3, DynamoDB, ElastiCache Redis, direct TCP communication.

The cost of Lambda instances increases linearly with the memory they are allocated, hence the M term in the cost equation. Although the average used for the time t may not necessarily be representative for the individual experienced communication times, it remains a useful approximation given the large number of FaaS functions serverless systems commonly handle.

Price-performance analysis. We now compute the cost and time required by each communication channel to communicate 1MB between two 2GiB Lambda functions a million times by instantiating the models previously described and present the results in Tab. IV. The direct communication is more than four times cheaper *and* faster than all alternatives, making it the best choice from a modeling standpoint. Furthermore, the cost of running the hole punching service will be shared by many function instances, making it negligible in practice.

VI. Evaluation

We now evaluate the performance and efficiency of our message-passing interface. We focus our evaluation on collective operations, as the point-to-point performance has been analyzed in Sec. IV-B. First, we examine the scaling of FMI Collectives on AWS Lambda (Sec. VI-A). Then, we verify FMI's performance in a comparison against MPI on virtual machines (Sec. VI-B), which allows us to quantify the overheads brought by the serverless environment (Sec. VI-C). Finally, we demonstrate how the integration of FMI into a serverless machine learning framework helps improve performance and decrease costs (Sec. VI-D).

A. Performance of FMI Collectives in FaaS

To evaluate FMI's collectives on a FaaS platform, we use AWS Lambda functions with 2 GiB RAM. For Redis, we use one `cache.t3.small` (1.37 GiB RAM, 2 vCPUs) instance, and we set the polling interval for S3 to 20ms. We evaluate collective operators with the following operations:

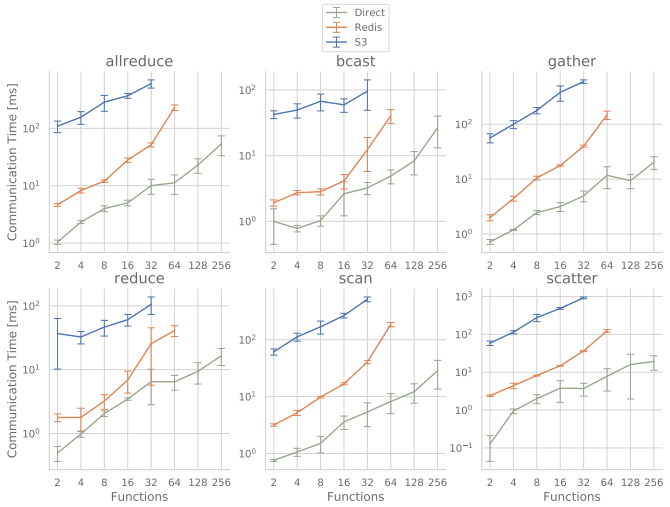


Fig. 5: Evaluation of FMI collectives on AWS Lambda.

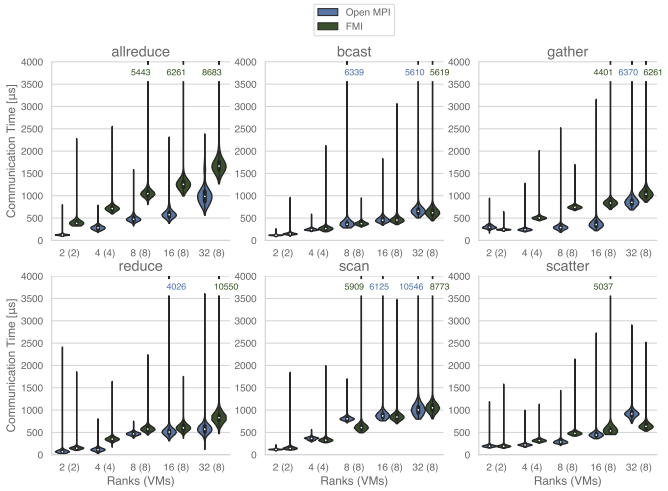


Fig. 6: Comparison of FMI with Open MPI in virtual machines.

- **allreduce**: Adding an integer per process.
- **bcast**: Broadcasting an integer.
- **gather**: The root process receives 5,000 integers in total.
- **reduce**: Adding an integer per process.
- **scan**: Prefix sum with an integer per process.
- **scatter**: The root process sends 5,000 integers in total.

The Redis and direct communication experiments are repeated 30 times, while for S3 we use only ten repetitions due to cost.

The results are summarized in Figure 5. For Redis, we observe a significant increase in communication times starting at 64 functions and even timeouts at 128 functions. This demonstrates the limitation of using a provisioned service where the user is responsible for scaling resources, and the correct instance size is not always known a priori. Choosing the minimal size that supports a given workload can be time-consuming and expensive because the communication times gradually increase, which often leads to overprovisioning. For S3, we limit the evaluation to 32 functions for cost reasons due to the many GET requests, further exacerbated by stragglers and short polling intervals. Like resource provisioning with Redis, the user has to select the polling configuration.

> Direct TCP communication enabled by FMI is necessary to achieve high performance collective operations.

B. Comparison of FMI and MPI in Virtual Machines

To compare the performance of FMI and MPI, we deploy both on virtual machines for an unbiased comparison, since MPI is not available on FaaS platforms.

Setup. We execute the MPI benchmarks on `t2.xlarge` virtual machines, running Ubuntu 20.04.1 VMs with 16 GiB of RAM and 4 vCPUs. Virtual machines are configured to use the Amazon Time Sync Service which improves the precision of the time measurements. We configure Open MPI to use one rank per node when using up to eight peers, and we use up to 4 processes per node otherwise. We use Open MPI 4.0.3 wih the default configuration, which uses the TCP transfer layer for communication. Similarly, we used a non-tuned FMI installation with the fastest channel. For this and the next experiment, FMI was compiled with our Docker image, and we use `gcc` 9.3.0 with the flag `-O3` to compile MPI benchmark. While the Amazon cloud does not provide any concrete numbers for the network bandwidth, we observed 1 Gbit/s when benchmarking the bandwidth between two VMs with `iperf`.

Performance The performance and variance of FMI collectives is comparable to Open MPI (Fig. 6). We repeat the evaluation of each collective operation 1,000 times, after discarding a first warm-up measurement, and use a barrier before each experiment. These results also show that co-location and using shared memory is beneficial as Open MPI is able to reach similar communication times for 8 and 16 ranks in some of the experiments. Our implementation of the collectives is competitive and our framework does not introduce significant overhead.

> FMI is competitive with established MPI implementations, bringing the HPC message-passing performance closer to the serverless world.

C. Evaluating the Overhead of FaaS Platforms

Thanks to FMI, we can quantify the performance losses incurred by the serverless environment by comparing operations executed in virtual machines (IaaS) and FaaS. Figure 7 compares the results of FMI communication in serverless (Sec. VI-A) with FMI and Open MPI performance on virtual machines (Sec. VI-B). The VM-based benchmark results in significantly lower communication times, even if the used software and communication algorithms are identical. This suggests that the performance of FaaS is currently hampered by opaque traffic limitations caused by the cloud infrastructure.

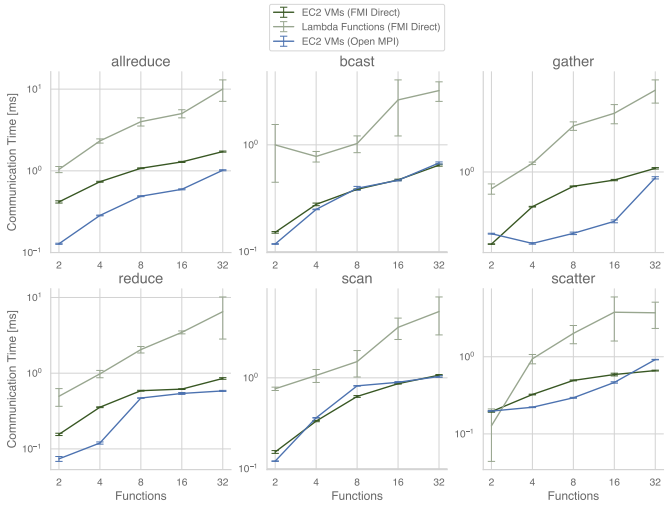


Fig. 7: Comparison of collectives on AWS EC2 and AWS Lambda with minimal message sizes.

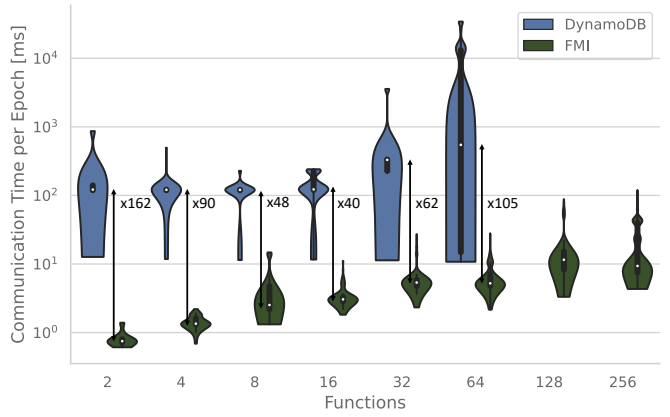


Fig. 8: Performance comparison of FMI and the LambdaML DynamoDB communication time. The annotation denotes the speedup of the median communications time provided by FMI. No measurements beyond 64 functions are provided for DynamoDB due to outliers and timeouts.

D. Practical Case Study: Distributed Machine Learning

To demonstrate the benefits of integrating FMI into serverless applications, we use LambdaML, a state-of-the-art framework for distributed machine learning on AWS Lambda [4]. In this experiment, we use distributed K-Means with the DynamoDB backend since this channel was seen as the best performing. For the FMI experiment, we replace the author's optimized allreduce operation with the corresponding FMI collective. We use the HIGGS [73] dataset with 1 MB file per function. For such small datasets, the communication overhead is especially important because the computation time is relatively short. Lambda functions are configured with 1 GiB RAM, and we train for 10 epochs. We enable autoscaling for DynamoDB and use direct communication over TCP in FMI.

Performance. Figure 8 presents the distribution of communication times per epoch, i.e., how much time passed between functions ready to exchange data until they accumulate the centroids of the current epoch, averaged across ten epochs. FMI significantly reduces both the median and maximum communication time by up to 105 and 1224 times, respectively, when running with 64 functions. We did not increase the number of functions beyond 64 for DynamoDB due to the timeouts and outliers we observed for 64 functions. In constrast, FMI scaled well with good performance and few outliers up to 256 functions. Another reason behind the poor performance of vanilla LambdaML is the `base64` serialization of binary data that is needed in communication using DynamoDB. On the other hand, FMI directly operates on binary data. Furthermore, we avoid unnecessary buffer copies by using `numpy` arrays on top of the existing memory buffer from the C++ library.

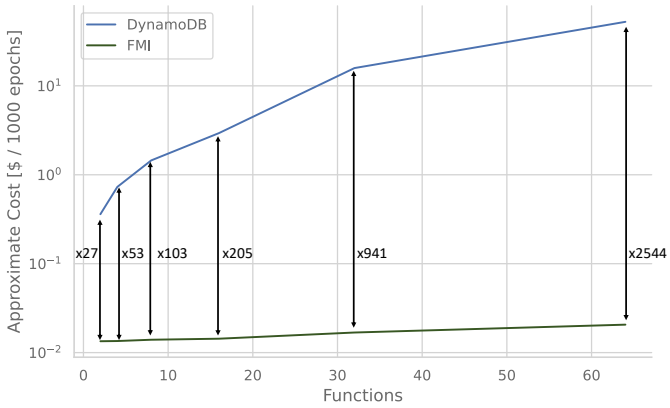


Fig. 9: Cost comparison of FMI and the LambdaML DynamoDB channel. The annotation denotes the cost reduction of the provided by FMI.

Cost. We estimate the cost of moving data and running the Lambda function for the duration of the communication epoch. For FMI, we assume an hourly cost of running the hole punching service. For DynamoDB, we assume one read and write unit per second for each function, as this is the minimum traffic AWS requires each function to provision. This assumption is rather optimistic because one function can generate multiple reads during one collective operation in LambdaML's implementation, as functions repeatedly send requests until an item becomes visible. Therefore, our estimation of the monetary benefits of using FMI is conservative.

Despite this pessimistic approximation, FMI results in significantly cheaper communication costs (Fig. 9). When using only a few functions, the cost is relatively similar because of the hourly cost of the hole punching server. At 32 functions, a user pays approximately $15.84 for 1000 epochs of communication with DynamoDB while using FMI lowers these costs to $0.02, a reduction by a factor of 941. This trend will increase with more functions, as the infrastructure costs depend linearly on the number of functions and the communication time increases significantly. On the other hand, the infrastructure costs for FMI are practically independent of

the number of functions, due to the limited requirements of the hole punching server. Users of FMI can further benefit from a cloud-managed communication establishing service that could offer lower prices.

Integration. We integrate FMI into the K-Means benchmarks with **only four lines of code changed**. A complete integration would reduce the LambdaML codebase by several hundred lines of code because the communication methods specific to machine learning are no longer needed.

FMI can be integrated into distributed serverless applications with minimal overhead, providing performance and cost improvements of two degrees of mangitude.

VII. Related Work

Solution	General Purpose	Object Storage	In-Memory Storage	Direct Communication	Central Server
Cirrus [1]					✗
Crucial [2]	✗		✗		
gg [74]	✗	✗	✗		
Lambada [3]		✗			
Boxer [25]				✗	
LambdaML [4]		✗	✗		✗
Locus [75]			✗		
mu [7]	✗				✗
Pocket [8]	✗		✗		
PyWren [76]	✗	✗			
Starling [6]		✗			
FMI (this work)	✗	✗	✗	✗	✗

TABLE V: Comparison of existing communication solutions.

Multiple works partially address the topic of communication in serverless environments and implement specialized systems for a given workload (Table V). In contrast, FMI provides a modular, high-performance, and general-purpose solution for point-to-point and group communication, with support for various communication channels and a model-driven selection at runtime.

Ephemeral Storage for Serverless. Pocket [8] is a specialized data store for intermediate data in serverless, with automatic resource scaling and multiple storage tiers. Pocket is orthogonal to our work and can be integrated into FMI as a cheaper alternative to the in-memory store. Locus [75] and Crucial [2] include specialized communication channels for serverless analytics and distributed synchronization. The former combines tiers of fast and slow storage, while the latter implements a shared object layer in an in-memory data store with a Java interface.

Serverless Communication. Emerging frameworks support stateful and distributed FaaS jobs, but many of them focus on domain-specific optimizations. Systems such as gg [74], mu [7], and PyWren [76] are designed to handle general-purpose tasks, and they use cloud stores, dedicated in-memory caches, and messaging servers.

ServerlessNetworking proposes reliable inter-function communication over UDP with help of the UDT library [77], and uses an external service for UDP hole punching. This leads to significant communication times, with the latency of some messages in the order of seconds.

Another approach uses a dedicated storage server deployed in a virtual machine to relay messages and implement application-specific operations [1]. However, these require manual management of scalability and are not always portable between applications.

Other systems target specific workloads and execution patterns. LambdaML [4] and Cirrus [1] are specialized frameworks for machine learning. They implement communication patterns common in machine learning training and introduce optimizations such as asynchronous communication, custom parameter servers, and dedicated data stores for intermediate data.

Lambada [3] implements communication specialized for scan and exchange operators common in data analytics, including concurrent storage operations and multi-level exchanges that minimize the number of storage operations. Boxer [25] extends Lambada with TCP hole punching. The query execution engine Starling [6] uses object storage and attempts to mitigate the high latency of operating on many files with parallelization, pipelining and multi-stage operations. The modular FMI system supports adding domain-specific communication optimizations, similarly to the multitude of specializations for MPI collectives.

Serverless Platforms. SONIC [78] extends OpenLambda with application-aware data passing. SAND [79] implements a dedicated hierarchical message bus, and Cloudburst [80] adds co-located caches and an autoscaling key-value store. The optimized communication channels available on a given platform can be integrated into FMI, letting users benefit from the high performance of message-based communication while hiding the complexity and specialization.

VIII. Possible Extensions

Collective algorithms variety.

Another interesting research direction is the extension of FMI with more collective algorithms. We currently provide one optimized algorithm per collective and channel type. Thanks to extensive research in this area, it is known that the best performing algorithm depends on various factors and different selection mechanisms have been developed. FMI is well suited for the integration of additional collective algorithms and dynamic selection at runtime. Cost is an additional interesting design dimension, as the fastest collective algorithm may not be the cheapest. Finally, the performance of collectives can be sensitive to imbalance and noise [81, 82] and require dynamic adaptive schemes for robustness [81, 83]. This is a significant problem in serverless, where performance variability is high [72].

Dynamic group membership. Ephemeral functions can be used to implement adaptive and dynamic parallelism of *evolving* and *malleable* applications [84]. Thus, serverless group communicators could support joining and leaving the group by functions at any time. While this topic is beyond the scope of this paper, it would bring the communication models

closer to the serverless ecosystem and allow a more seamless integration.

IX. Conclusions

In this work, we propose FMI: an easy-to-use, modular, high-performance framework for general-purpose communication in FaaS platforms. We analyze and benchmark communication channels available to serverless applications, including direct communication with NAT hole punching. We derive performance and cost models to support the model-driven selection of optimal communication protocols. Finally, we design the FMI interface after MPI, ensuring the library can be easily integrated into existing C++ or Python codebases, immediately providing significant cost savings and performance improvements.

We evaluate FMI by comparing the performance of both point–to–point and collective communication to MPI. We demonstrate the benefits of FMI for serverless in a case study of distributed machine learning, showing easy integration and decreased communication time by up to 162x. Thus, FMI's efficient and scalable message passing brings serverless communication closer to the performance of MPI communication in HPC, and lifts one of the most important limitations of serverless computing.

References

[1] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ml workflows," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 13–24. [Online]. Available: https://doi.org/10.1145/3357223.3362711

[2] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López, "On the faas track: Building stateful distributed applications with serverless architectures," in *Proceedings of the 20th International Middleware Conference*, ser. Middleware '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 41–54. [Online]. Available: https://doi.org/10.1145/3361525.3361535

[3] I. Müller, R. Marroquín, and G. Alonso, "Lambada: Interactive data analytics on cold data using serverless cloud infrastructure," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 115–130.

[4] J. Jiang, S. Gan, Y. Liu, F. Wang, G. Alonso, A. Klimovic, A. Singla, W. Wu, and C. Zhang, "Towards demystifying serverless machine learning training," in *ACM SIGMOD International Conference on Management of Data (SIGMOD 2021)*, June 2021. [Online]. Available: https://www.microsoft.com/en-us/research/publication/towards-demystifying-serverless-machine-learning-training/

[5] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 193–206. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/pu

[6] M. Perron, R. Castro Fernandez, D. DeWitt, and S. Madden, "Starling: A Scalable Query Engine on Cloud Functions," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. Portland OR USA: ACM, Jun. 2020, pp. 131–141.

[7] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'17. USA: USENIX Association, 2017, pp. 363–376.

[8] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, pp. 427–444.

[9] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn, "Collective communication: theory, practice, and experience," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 13, pp. 1749–1783, 2007. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1206

[10] M. Barnett, L. Shuler, R. van de Geijn, S. Gupta, D. Payne, and J. Watts, "Interprocessor collective communication library (intercom)," in *Proceedings of IEEE Scalable High Performance Computing Conference*, 1994, pp. 357–364.

[11] R. Rabenseifner, "Automatic mpi counter profiling of all users: First results on a cray t3e 900-512," in *Proceedings of the message passing interface developer's and user's conference*, vol. 1999, 1999, pp. 77–85.

[12] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[13] S. Gorlatch, "Send-receive considered harmful: Myths and realities of message passing," *ACM Trans. Program. Lang. Syst.*, vol. 26, no. 1, p. 47–56, jan 2004. [Online]. Available: https://doi.org/10.1145/963778.963780

[14] P. Sanders, J. Speck, and J. L. Träff, "Two-tree algorithms for full bandwidth broadcast, reduction and scan," *Parallel Comput.*, vol. 35, no. 12, p. 581–594, dec 2009. [Online]. Available: https://doi.org/10.1016/j.parco.2009.09.001

[15] R. Thakur and W. D. Gropp, "Improving the performance of collective operations in mpich," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, J. Dongarra, D. Laforenza, and S. Orlando, Eds. Berlin,

Heidelberg: Springer Berlin Heidelberg, 2003, pp. 257–267.

[16] B. Tu, J. Fan, J. Zhan, and X. Zhao, "Performance analysis and optimization of mpi collective operations on multi-core clusters," *The Journal of Supercomputing*, vol. 60, no. 1, pp. 141–162, Apr 2012. [Online]. Available: https://doi.org/10.1007/s11227-009-0296-3

[17] S. Li, T. Hoefler, and M. Snir, "Numa-aware shared-memory collective communication for mpi," in *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 85–96. [Online]. Available: https://doi.org/10.1145/2462902.2462903

[18] S. Jain, R. Kaleem, M. G. Balmana, A. Langer, D. Durnov, A. Sannikov, and M. Garzaran, "Framework for scalable intra-node collective operations using shared memory," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. IEEE Press, 2018.

[19] C. Renggli, S. Ashkboos, M. Aghagolzadeh, D. Alistarh, and T. Hoefler, "Sparcml: High-performance sparse communication for machine learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356222

[20] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of mpi collective operations," *Cluster Computing*, vol. 10, no. 2, pp. 127–143, Jun 2007. [Online]. Available: https://doi.org/10.1007/s10586-007-0012-0

[21] P. Husbands and J. Hoe, "Mpi-start: Delivering network performance to numerical applications," in *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, 1998, pp. 17–17.

[22] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson, "What serverless computing is and should become: The next phase of cloud computing," *Communications of the ACM*, vol. 64, no. 5, pp. 76–84, 2021.

[23] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. J. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing," *CoRR*, vol. abs/1902.03383, 2019. [Online]. Available: http://arxiv.org/abs/1902.03383

[24] V. Giménez-Alventosa, G. Moltó, and M. Caballer, "A framework and a performance assessment for serverless mapreduce on aws lambda," *Future Generation Computer Systems*, vol. 97, pp. 259–274, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X18325172

[25] M. Wawrzoniak, I. Müller, R. Fraga Barcelos Paulus Bruno, and G. Alonso, "Boxer: Data analytics on network-enabled serverless platforms," in *11th Annual Conference on Innovative Data Systems Research (CIDR'21)*, 2021.

[26] T. Hoefler, A. Lumsdaine, and J. Dongarra, "Towards efficient mapreduce using mpi," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, M. Ropo, J. Westerholm, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 240–249.

[27] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," *arXiv preprint arXiv:1812.03651*, 2018.

[28] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas, "Windows azure storage: A highly available cloud storage service with strong consistency," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 143–157. [Online]. Available: https://doi.org/10.1145/2043556.2043571

[29] "AWS S3," https://aws.amazon.com/s3/, 2006, accessed: 2022-01-30.

[30] "Azure Blob Storage," https://azure.microsoft.com/en-us/services/storage/blobs/, 2008, accessed: 2022-01-30.

[31] "Google Cloud Storage," https://cloud.google.com/storage, 2010, accessed: 2022-01-30.

[32] "AWS Dynamo DB," https://aws.amazon.com/nosql/key-value/, 2012, accessed: 2022-01-30.

[33] "Azure Cosmos DB," https://azure.microsoft.com/en-us/services/cosmos-db/, 2017, accessed: 2022-01-30.

[34] "Amazon ElastiCache for Redis," https://aws.amazon.com/elasticache/redis/.

[35] "Amazon ElastiCache for Memcached," https://aws.amazon.com/elasticache/memcached/.

[36] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi, "The quic transport protocol: Design and internet-scale deployment," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 183–196. [Online]. Available: https://doi.org/10.1145/3098822.3098842

[37] M. Holdrege and P. Srisuresh, "IP network address translator (NAT) terminology and considerations," RFC 2663, Aug. 1999.

[38] B. Ford, P. Srisuresh, and D. Kegel, "Peer-to-peer communication across network address translators," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. USA: USENIX Association, Apr. 2005, p. 13.

[39] J. L. Eppinger, "TCP Connections for P2P Apps: A Software Approach to Solving the NAT Problem," *Carnegie Mellon University, Technical Report*, vol. ISRI-05-104, Jan. 2005.

[40] J. C. Mogul and G. Minshall, "Rethinking the tcp nagle algorithm," *SIGCOMM Comput. Commun. Rev.*, vol. 31, no. 1, p. 6–20, jan 2001. [Online]. Available: https://doi.org/10.1145/382176.382177

[41] T. Hoefler and D. Moor, "Energy, memory, and runtime tradeoffs for implementing collective communication operations," *Supercomputing Frontiers and Innovations*, vol. 1, no. 2, p. 58–75, Sep. 2014. [Online]. Available: https://superfri.susu.ru/index.php/superfri/article/view/12

[42] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *Journal of Parallel and Distributed Computing*, vol. 69, no. 2, pp. 117–124, 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731508001767

[43] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang, "Magpie: Mpi's collective communication operations for clustered wide area systems," in *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '99. New York, NY, USA: Association for Computing Machinery, 1999, p. 131–140. [Online]. Available: https://doi.org/10.1145/301104.301116

[44] M. Alfatafta, Z. AlSader, and S. Al-Kiswany, "Cool: A cloud-optimized structure for mpi collective operations," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 746–753.

[45] P. Sanders, J. Speck, and J. L. Träff, "Full bandwidth broadcast, reduction and scan with only two trees," in *Proceedings of the 14th European Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. PVM/MPI'07. Berlin, Heidelberg: Springer-Verlag, 2007, p. 17–26.

[46] X. Luo, W. Wu, G. Bosilca, Y. Pei, Q. Cao, T. Patinyasakdikul, D. Zhong, and J. Dongarra, "Han: a hierarchical autotuned collective communication framework," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 2020, pp. 23–34.

[47] J. Pješivac-Grbović, G. Bosilca, G. E. Fagg, T. Angskun, and J. J. Dongarra, "Mpi collective algorithm selection and quadtree encoding," *Parallel Computing*, vol. 33, no. 9, pp. 613–623, 2007, selected Papers from EuroPVM/MPI 2006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819107000804

[48] S. Vadhiyar, G. Fagg, and J. Dongarra, "Automatically tuned collective communications," in *SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, 2000, pp. 3–3.

[49] P. Sack and W. Gropp, "Faster topology-aware collective algorithms through non-minimal communication," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 45–54. [Online]. Available: https://doi.org/10.1145/2145816.2145823

[50] N. Karonis, B. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan, "Exploiting hierarchy in parallel computer networks to optimize collective operation performance," in *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, 2000, pp. 377–384.

[51] M. Bayatpour, S. Chakraborty, H. Subramoni, X. Lu, and D. K. D. Panda, "Scalable reduction collectives with data partitioning-based multi-leader design," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3126908.3126954

[52] A. A. Awan, K. Hamidouche, A. Venkatesh, and D. K. Panda, "Efficient large message broadcast using nccl and cuda-aware mpi for deep learning," in *Proceedings of the 23rd European MPI Users' Group Meeting*, ser. EuroMPI 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 15–22. [Online]. Available: https://doi.org/10.1145/2966884.2966912

[53] M. Bayatpour, J. Maqbool Hashmi, S. Chakraborty, H. Subramoni, P. Kousha, and D. K. Panda, "Salar: Scalable and adaptive designs for large message reduction collectives," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018, pp. 12–23.

[54] M. Copik, K. Taranov, A. Calotoiu, and T. Hoefler, "rfaas: Rdma-enabled faas platform for serverless high-performance computing," *arXiv preprint arXiv:2106.13859*, 2021.

[55] K. Satzke, I. E. Akkus, R. Chen, I. Rimac, M. Stein, A. Beck, P. Aditya, M. Vanga, and V. Hilt, "Efficient gpu sharing for serverless workflows," in *Proceedings of the 1st Workshop on High Performance Serverless Computing*, 2020, pp. 17–24.

[56] J. Kim, T. J. Jun, D. Kang, D. Kim, and D. Kim, "Gpu enabled serverless computing framework," in *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. IEEE, 2018, pp. 533–540.

[57] S. H. Mirsadeghi and A. Afsahi, "Topology-aware rank reordering for mpi collectives," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 1759–1768.

[58] A. Bhatele, T. Gamblin, S. H. Langer, P.-T. Bremer, E. W. Draeger, B. Hamann, K. E. Isaacs, A. G. Landge,

J. A. Levine, V. Pascucci, M. Schulz, and C. H. Still, "Mapping applications with collectives over sub-communicators on torus networks," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–11.

[59] S. Pickartz, C. Clauss, S. Lankes, and A. Monti, "Enabling hierarchy-aware mpi collectives in dynamically changing topologies," in *Proceedings of the 24th European MPI Users' Group Meeting*, ser. EuroMPI '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3127024.3127031

[60] M. P. I. Forum, "MPI: A Message-Passing Interface Standard Version 3.0," 09 2012.

[61] M. Copik, T. Grosser, T. Hoefler, P. Bientinesi, and B. Berkels, "Work-stealing prefix scan: Addressing load imbalance in large-scale image registration," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 3, pp. 523–535, 2022.

[62] S. Zhuang, Z. Li, D. Zhuo, S. Wang, E. Liang, R. Nishihara, P. Moritz, and I. Stoica, "Hoplite: Efficient and fault-tolerant collective communication for task-based distributed systems," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ser. SIGCOMM '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 641–656. [Online]. Available: https://doi.org/10.1145/3452296.3472897

[63] G. Blelloch, "Scans as primitive parallel operations," *IEEE Transactions on Computers*, vol. 38, no. 11, pp. 1526–1538, 1989.

[64] P. Sanders and J. L. Träff, "Parallel Prefix (Scan) Algorithms for MPI," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, B. Mohr, J. L. Träff, J. Worringen, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, vol. 4192, pp. 49–57.

[65] "Provision Infrastructure As Code – AWS CloudFormation – Amazon Web Services," https://aws.amazon.com/cloudformation/.

[66] T. Hoefler, C. Siebert, and A. Lumsdaine, "Group operation assembly language - a flexible way to express collective communication," in *2009 International Conference on Parallel Processing*, 2009, pp. 574–581.

[67] A. Klimovic, Y. Wang, C. Kozyrakis, P. Stuedi, J. Pfefferle, and A. Trivedi, "Understanding ephemeral storage for serverless analytics," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 789–794. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/klimovic-serverless

[68] A. Wang, J. Zhang, X. Ma, A. Anwar, L. Rupprecht, D. Skourtis, V. Tarasov, F. Yan, and Y. Cheng, "InfiniCache: Exploiting ephemeral serverless functions to build a Cost-Effective memory cache," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 267–281. [Online]. Available: https://www.usenix.org/conference/fast20/presentation/wang-ao

[69] T. Hoefler, T. Schneider, and A. Lumsdaine, "Accurately measuring overhead, communication time and progression of blocking and nonblocking collective operations at massive scale," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 25, no. 4, pp. 241–258, 2010. [Online]. Available: https://doi.org/10.1080/17445760902894688

[70] ——, "Accurately measuring collective operations at massive scale," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1–8.

[71] J. A. Rico-Gallego, J. C. Díaz-Martín, R. R. Manumachu, and A. L. Lastovetsky, "A survey of communication performance models for high-performance computing," *ACM Comput. Surv.*, vol. 51, no. 6, jan 2019. [Online]. Available: https://doi.org/10.1145/3284358

[72] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "Sebs: A serverless benchmark suite for function-as-a-service computing," in *Proceedings of the 22nd International Middleware Conference*, ser. Middleware '21. Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3464298.3476133

[73] P. Baldi, P. Sadowski, and D. Whiteson, "Searching for exotic particles in high-energy physics with deep learning," *Nature Communications*, vol. 5, no. 1, p. 4308, Jul. 2014.

[74] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein, "From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 475–488. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/fouladi

[75] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 193–206.

[76] E. Jonas, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," *CoRR*, vol. abs/1702.04024, 2017. [Online]. Available: http://arxiv.org/abs/1702.04024

[77] "ServerlessNetworking," http://networkingclients.serverlesstech.net/, 2019, accessed: 2022-03-22.

[78] A. Mahgoub, L. Wang, K. Shankar, Y. Zhang, H. Tian, S. Mitra, Y. Peng, H. Wang, A. Klimovic, H. Yang *et al.*, "{SONIC}: Application-aware data passing for chained serverless applications," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 285–301.

[79] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "Sand: Towards high-performance serverless computing," in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '18. USA: USENIX Association, 2018, pp. 923–935.

[80] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful functions-as-a-service," *Proc. VLDB Endow.*, vol. 13, no. 12, p. 2438–2452, Jul. 2020. [Online]. Available: https://doi.org/10.14778/3407790.3407836

[81] A. Faraj, P. Patarasuk, and X. Yuan, "A study of process arrival patterns for mpi collective operations," *International Journal of Parallel Programming*, vol. 36, no. 6, pp. 543–570, 2008.

[82] X. Luo, W. Wu, G. Bosilca, T. Patinyasakdikul, L. Wang, and J. Dongarra, "Adapt: An event-based adaptive collective communication framework," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 118–130. [Online]. Available: https://doi.org/10.1145/3208040.3208054

[83] P. Patarasuk and X. Yuan, "Efficient mpi bcast across different process arrival patterns," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1–11.

[84] D. G. Feitelson and L. Rudolph, "Toward convergence in job schedulers for parallel supercomputers," in *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 1–26.