rFaaS: RDMA-Enabled FaaS Platform for Serverless High-Performance Computing

Marcin Copik  
marcin.copik@inf.ethz.ch  
ETH Zürich  
Switzerland

Alexandru Calotoiu  
ETH Zürich  
Switzerland

Konstantin Taranov  
ETH Zürich  
Switzerland

Torsten Hoefler  
ETH Zürich  
Switzerland

ABSTRACT
The rigid MPI programming model and batch scheduling dominate high-performance computing. While clouds brought new levels of elasticity into the world of computing, supercomputers still suffer from low resource utilization rates. To enhance supercomputing clusters with the benefits of serverless computing, a modern cloud programming paradigm for pay-as-you-go execution of stateless functions, we present rFaaS, the first RDMA-aware Function-as-a-Service (FaaS) platform. With hot invocations and decentralized function placement, we overcome the major performance limitations of FaaS systems and provide low-latency remote invocations in multi-tenant environments. We evaluate the new serverless system through a series of microbenchmarks and show that remote functions execute with negligible performance overheads. We demonstrate how serverless computing can bring elastic resource management into MPI-based high-performance applications. Overall, our results show that MPI applications can benefit from modern cloud programming paradigms to guarantee high performance at lower resource costs.

1 INTRODUCTION

The landscape of high-performance computing is dominated by the Message-Passing Interface (MPI), a de facto standard distributed programming paradigm. Together with job batch scheduling [40] and multithreaded frameworks for shared-memory programming, MPI is the leading use case for clusters and supercomputers [74]. New MPI standards and implementations have brought the benefits of emerging network protocols, above all with remote direct memory access (RDMA) networks such as InfiniBand [64] and the inclusion of RDMA programming through one-sided communication [44]. Yet, the current setup does not address all of the challenges of distributed computing, and a predominant example is the ability to adapt resource allocation to changing application requirements. Evolving and malleable applications [39] achieve lower efficiency when resource allocation cannot be adjusted. These applications can consist of multiple phases with varying parallelism. Starting from MPI 2.0, applications are permitted to change the number of processes during execution. However, this feature has not been explored by many applications because of a complex setup and lack of integration with resource managers [76]. In the rigid HPC world, job schedulers and applications use only a fixed number of resources, leading to overallocation and underutilization of cores.

Achieving high utilization rates of supercomputers has always been challenging, and past predictions showed a pessimistic research outlook: "the goal of achieving near 100% utilization while supporting a real parallel supercomputing workload is unrealistic" [56]. Recent results are not much more optimistic: while a usage analysis of the Kraken supercomputer capacity over a year indicates average utilization as high as 94% [87], and a median utilization of around 90% on the Mira supercomputer [73], a four-year study of the Blue Waters system capacity presents monthly utilization rates that rarely exceed 80% [57]. Furthermore, on average three-quarters of node memory is not utilized [72]. On such systems, a 10% decrease in monthly utilization rate leads to hundreds of thousands of dollars of investment into hardware that stays unused. To assess the modern scale of the problem, we analyzed over a week the utilization of the Piz Daint supercomputing system [15], and present in Figures 2a and 2b the CPU and memory utilization data, respectively. The rapid and frequent changes indicate that resources do not stay idle for an extended period of time, and this gap cannot be addressed with persistent and long-running allocations. However, fine-grained and ephemeral programming models could take advantage of such resources - to the benefit of both HPC users and administrators, as idle computing and memory resources could be offered at much lower costs. This problem is not limited to HPC and supercomputers: data centers suffer from low utilization as well, caused by resource overprovisioning to handle peak demand [60].
In recent years, the world of computing has seen two major innovations brought by the cloud: elastic resource management and a cost decrease of 5 to 7 times [22]. Even though the early examination of cloud systems showed that Infrastructure-as-a-Service (IaaS) resources are not a viable alternative to high-performance systems, as they are characterized by significant overheads, high costs, and unsatisfactory I/O performance [51, 53, 70], their performance and attractiveness has improved over time [71, 89]. 

Virtual machines and containers, the important virtualization solutions in the cloud systems, have been found to be an efficient abstraction level for high-performance applications [50, 77], and cloud network performance improved as well [90, 91]. The advantages of the cloud have been quickly identified to improve the performance of on-premises clusters by allocating computing resources in the cloud [34] (HPC plus cloud) and executing elastic MPI applications [76] (HPC in the cloud). In particular, the concept of HPC-as-a-Service [19] brings a cloud abstraction model to manage and access HPC resources. Yet, no work has fully embraced the cloud revolution to improve the efficiency of existing supercomputing systems. The question of incorporating elastic cloud resources into MPI applications remains open due to a lack of a cloud-native programming model.

Function-as-a-Service (FaaS) is a new cloud paradigm combining the full elasticity of cloud resources with a maximally simplified programming model: users program stateless functions, and the cloud takes away the responsibility of scheduling invocations of such functions. Thanks to the fine-grained parallelism and the pay-as-you-go billing system, serverless functions have become a solution for all tasks that can benefit from an elastic abstraction of computing resources. While the progress towards HPC-as-a-Service continues [23], functions are not yet ready for HPC applications due to high invocation latencies and computing costs [31, 55]. For FaaS to become a viable programming model for high-performance applications, it must overcome the crucial performance and integration challenges: (1) fully utilize the potential of network devices, (2) provide low-latency invocations with minimal added overheads, (3) offer the always-available computing resources for invocations on the critical path, (4) and integrate into existing and proven solutions for programming and dispatching high-performance software.

We address these challenges in rFaaS, the first RDMA-capable serverless platform with a decentralized resource management model (Sec. 3). We show how novel hot serverless invocations improve over warm executions with an added latency of little over 300 nanoseconds on top of the fastest available network transmission. We discuss the characteristics of HPC applications that can exploit the benefits of rFaaS functions (Sec. 4). We present a programming model for straightforward integration of rFaaS into new and existing C++ MPI codebases, with a high degree of compatibility with other standard-compliant frameworks. With the incorporation of serverless computing into MPI applications, we make a major step towards elastic HPC in the cloud. Furthermore, we provide a batch system integration for the dynamic allocation of serverless functions on idle resources (Sec. 5). Supercomputing systems could reasonably offer idle resources at discounted rates to incentivize better utilization. Therefore, this HPC-as-a-Service solution can increase the efficiency and utilization of the overall cluster. We demonstrate the elasticity, efficiency, and high performance of rFaaS with an evaluation on microbenchmarks and show that HPC workloads can be accelerated with serverless functions (Sec. 6).

Our paper makes the following contributions:

- We present the design and implementation of the first RDMA-capable serverless platform, including (1) a decentralized FaaS resource management protocol and (2) a novel, low-latency, and zero-copy hot type of serverless invocations. rFaaS is publicly available on an open-source license.
- We conduct an experimental verification against state-of-the-art open-source and commercial serverless platforms summarized in Figure 1 and show that FaaS (1) has a median overhead over pure RDMA transmission of little over 300 nanoseconds, (2) achieves the available link bandwidth, and (3) scales efficiently up to 64 invocations on two nodes.
- We design a C++ programming model and an integration into cluster management systems to invoke rFaaS functions on spare cluster capacity. We show that we can improve supercomputers’ utilization by applying modern cloud abstractions to take advantage of short-lived idle resources.
- We show how rFaaS can be used in an HPC context by accelerating MPI applications. With three case studies, we show that speedups up to 2.2x can be achieved through task offloading into cheap remote functions.

2 BACKGROUND

2.1 FaaS Computing

Function-as-a-service (FaaS) is a cloud service concerned with executing stateless and short-running functions. The serverless functions are dynamically allocated in the cloud, and the users are freed from the responsibility of creating, maintaining, and scaling the infrastructure.
from the usual responsibilities of managing resources. The cloud provider charges users only for the time and resources used in a function execution, and applications with irregular or infrequent workloads can benefit from the elastic allocation of computing resources and the pay-as-you-go billing system. For a cloud operator, the fine-grained executions provide an opportunity to increase system efficiency through oversubscription and more efficient scheduling. Serverless is adopted by major cloud systems [1, 3–5].

We characterize the serverless platforms briefly with a high-level overview presented in Figure 3 and refer interested readers to a wider discussion in the literature [31, 55, 86]. Functions are invoked via triggers (A), including internal cloud events such as database update or a new entry in a queue, and the standard external trigger via a cloud HTTP gateway that exposes functions to the outside world. A function scheduler (B) places the invocation in a cloud-native execution environment (C), and the function code is downloaded from the cloud storage (D). Function are allowed to initiate connections to external cloud resources and services, and can also use the filesystem of its sandbox as a temporary storage. A sandbox instance handles many consecutive invocations, so resources are cached and reused across executions.

Invocations. The primary types are cold and warm. Cold invocations occur if the FaaS manager cannot find an idle sandbox for a given function, and must allocate a new one. The latency includes the time to allocate a sandbox, download the function code from external storage and start an executor process. In a warm invocation, the function payload is sent directly to the executing process.

The unpredictable and high-latency cold startups are a major issues with serverless [67, 81] as they can add seconds of overhead to each invocation. Modern lightweight virtual machines are designed to support low-latency and burstable serverless invocations [20]. However, even warm invocations can incur significant overheads. On AWS Lambda, each invocation is processed by a dedicated management service to decide function placement [20]. The function input is limited to a few megabytes, so users must transmit larger payloads via the high-latency public cloud storage. The invocation’s critical path is even longer in OpenWhisk [2], as it includes a controller, database, load balancer, and a message bus [78].

High-Performance Serverless. Serverless is used by compute-intensive workloads such as data analytics, video encoding, linear algebra, and machine learning [25, 41, 42, 52, 65, 69, 75, 79]. The elastic parallelism of FaaS has so far not gained significant attention in the world of HPC applications. Such applications require low-latency communication and optimized data movement. They cannot tolerate the large overheads of invoking remote functions. They need a pricing model that is fair towards compute-intensive functions [55]. Although recent research focuses on improving the performance of serverless functions by exploiting data locality and co-locating invocations on the same machine [21], HPC applications need fast remote invocations to achieve high scalability.

3 RDMA-BASED SERVERLESS PLATFORM

In this section, we describe rFaaS and its most important features. rFaaS is a serverless platform tailored for the needs of high-performance applications, combining the flexibility of FaaS systems with the low overhead executions offered by cluster systems (Figure 4). rFaaS implements the main FaaS paradigm of remote executions of stateless functions but avoids major performance overheads of serverless computing by replacing the REST and RPC-based invocation interface with direct memory operations on remote servers.

Our philosophy in implementing rFaaS is to drastically reduce the critical path of warm and cold invocations. We achieve this goal by reducing the number of parties involved in transmitting function data and removing the centralized gateway and resource manager from the invocation path. Figure 5 shows an overview of rFaaS: a decentralized allocation system where clients negotiate a lease of computing resources from a frequently refreshed list of available servers (Sec. 3.2). Our functions gain a direct RDMA connection to executor servers without sacrificing their serverless nature: no assumptions are made about the underlying computing and storage hardware as in other FaaS platforms. We capitalize on this gain further by implementing an RDMA-based invocation system designed to minimize invocation latency (Sec. 3.3).

3.1 Components of rFaaS

Resource Manager. A global database of active resources is a necessary component of each serverless platform. Servers become available for function execution as soon as the database contains an entry with a description of their resources and connection details (D). The role of the resource manager is to update and distribute a ranked list of executor servers. The cluster batch system adds and removes servers, and the manager uses heartbeats to verify the status of executor servers periodically and remove unresponsive
resources. Clients can read the list of executor servers and receive updates asynchronously through RDMA operations (9).

**Function Executor Server.** When clients begin offloading serverless tasks to rFaaS, they select executor servers to achieve the desired number of parallel workers. These servers offer the idle and unused hardware resources (CPU cores, memory) to support dynamic execution of serverless functions. Clients negotiate an allocation of computing resources with an executor manager (8). The dedicated executor manager process is responsible for connecting new clients, initializing containerized executors, removing containers that are idle for a long time or exceed specified time limits, and accounting for resource consumption. When an allocation is successful, executor managers initialize an isolated execution context with an RDMA-capable execution process. Finally, clients can establish a direct RDMA connection with each allocated executor process and invoke functions by writing function header and payload directly into their memory (9). The results are returned to the client in a similar fashion, and the allocation status is cached on the client’s side for consecutive invocations on warmed-up resources.

### 3.2 Decentralized Allocation

A feature rFaaS provides is decentralized resource management, differentiating it from other commercial and open-source FaaS implementations. To execute a function, clients do not involve the resource manager. Instead, they select a permutation of resource servers and send allocation requests directly to those servers. Clients use a random permutation of a sorted list of servers to decrease the likelihood of contention and conflicts, and ensure that each server is requested exactly once. Once successful allocation, managers allocate isolated execution contexts, initialize RDMA-aware execution processes, and notify clients. At that point, clients can send function invocations. Connections to executor processes are cached locally by clients to provide fast consecutive executions on warmed-up resources. To support a straightforward de-allocation of temporary and on-demand executors, clients use the connection status to check if the execution context is alive.

### 3.3 Low-Latency Invocation

A critical feature of rFaaS is ensuring invocations have a low-latency invocation. While an on-demand allocation of idle resources improves the economics of cluster batch systems, without low-latency invocations it would be counterproductive to incorporate rFaaS functions into HPC applications. In Figure 6, we present the steps and overheads of various invocation models in rFaaS. Our platform preserves the FaaS semantics of, and we extend the invocation models with a new type of hot invocation that guarantees zero-copy executions on always ready and available hardware.

**Cold.** The cold invocation includes significant overheads caused by the initialization of an execution context. In rFaaS, clients negotiate the allocation directly with executor servers by sending requests specifying the desired number of cores, memory, and timeout for the allocation. Clients send allocation requests iteratively until they succeed in allocating the desired number of computing resources. A failed allocation request is returned immediately to the client. Executor servers initialize an isolated execution sandbox and assign the requested computing and memory resources to it. The executor process starts in the sandbox, accesses the selected RDMA device, registers memory buffers, creates worker threads pinned to assigned cores. Each executor has a configurable number of thread workers that work independently of each other, and each one corresponds to a single function instance. Clients can allocate multiple workers in a single allocation request. When the initialization is done, the client receives the executor’s connection settings, establishes connections to all threads, and can write requests directly to the workers. This process leads to a warm or hot invocation, depending on the delay between allocation and the execution. Overall, sandbox initialization adds on average 25 ms and 2.7 seconds of overhead for bare-metal and Docker-based executor, respectively, on an HPC node (Section 6). The overheads of forking and initializing a new executor dominates the cold invocation time.

**Warm.** Warm invocations occur when a sandbox and execution process is already allocated. A client transmits the function payload to a worker thread of the execution process using a direct RDMA connection. We implement warm invocations in rFaaS with the help of RDMA completion events. Threads do not share RDMA resources, and they wait independently for completion events corresponding to new invocation requests. Once a completion event arrives, the thread wakes and executes the request. Handling RDMA completion events impacts performance, but significantly decreases the pressure on computing resources compared to active poll methods.

---

Figure 5: rFaaS system: resource manager receives (1) and distributes (2) computing resource availability, clients allocate executors (3) and remotely invoke functions (4).

Figure 6: Lifetime of an rFaaS function. Similarly to serverless systems, the cold start times are dominated by sandbox initialization. Warm and hot invocation times include rFaaS overhead and latency of RDMA write of N bytes of payload.
Compared to native RDMA performance, warm invocations have an overall overhead of less than 6 microseconds for a round-trip invocation. The thread enters the hot invocation mode immediately after execution and polls RDMA events without sleeping to improve consecutive invocations’ performance. We roll back to warm execution after a predefined time without seeing a new invocation.

Hot. The novel hot invocations further improve the performance of warm FaaS executions by adding the new obligation that threads actively poll for invocation requests. Thread workers switch from blocking on completion queues to active polling of them. Switching to polling mode decreases the invocation latency since threads do not enter a blocked state to wait for an interrupt generated by the RDMA driver. This configuration decreases the overall overhead for a round-trip invocation to ca. 300 nanoseconds on average.

3.4 Scalability
A high-performance serverless platform must handle scaling in three domains: number of active allocations in an HPC system (users of FaaS), size of each allocation (number of MPI ranks offloading work), and the number of functions invoked by each process.

Horizontal Scaling. The number of rFaaS clients in a system can reach many thousands with jobs using serverless acceleration on all of their MPI processes. To scale with the demand, we offer two strategies: replication and connection optimization. First, to support the many rFaaS clients in the system, we replicate the resource manager where each replica serves data on executor availability. The replicated datastores employ expensive, strongly consistent transactions to prevent stale reads in distributed settings. However, stale reads are not a concern for the replicated resource manager. A stale announcement of new resources can lead to a slightly smaller availability of computing resources for a subset of clients and only for a short time. Similarly, removing resources from the manager is not a problem since rFaaS clients must tolerate the volatility of transient resources. As a result, our replicated resource manager can implement a less expensive but scalable eventual consistency [58], subject to stale reads but guaranteeing that all replicas eventually have the same view on the available computing resources.

Second, we optimize the connection structure of resource managers to support efficient notifications. Between resource and executor managers, we use reliable RDMA connections, as they need to support atomic operations. The updates in executor availability must be distributed to thousands of clients, but the network throughput of RDMA connections decreases significantly with the number of clients [28, 58]. Reliability is not strictly needed to distribute delta updates, and packet losses are infrequent [58]. Therefore, we can use unreliable datagrams and multicast for that task as it scales much better than RDMA connections.

Executor Scaling. Thanks to the decentralized allocation policy and a random selection, the executor managers are expected to usually receive requests only from a small fraction of potential clients. However, when the HPC cluster achieves almost ideal utilization, the list of available computing resources becomes very small. As a result, significant contention can appear when a large number of clients start a random walk over a small list of rFaaS executor servers. To prevent clients from overwhelming executor managers, the managers notify a selected resource manager as soon as they are fully allocated. The resource manager distributes the information to replicas and clients, and executor servers will not receive new requests until they announce availability again. Resource starvation can thus be detected much quicker, and contention can be avoided.

Parallel Invocations. rFaaS implements parallel functions invocations by simultaneous dispatching of many function execution requests to threads of remote executors. Since a client has a direct RDMA connection to each thread worker, it can invoke functions independently. The scalability is achieved by exploiting the non-blocking nature of RDMA write operations and using disjoint memory buffers to store invocation results. The use of multiple RDMA connections improves network utilization as more processing units of a network controller are involved [58]. Thread workers execute functions in parallel without interfering with each other. In addition, each executor thread is accounted separately and switches between hot-warm invocations on its own, further aiding elasticity.

3.5 Fault tolerance
rFaaS clients can experience failures in two ways: when the server is removed from the pool of active executors or is shut down uncontrollably, and when the function crashes the execution process. The client library caches connections for further invocations so a server disappearance will be detected through a disrupted RDMA connection, and will be removed from the local cache. Handling function failures requires the the executor manager. The manager frequently verifies the status of the executor process. When it detects that the process exited prematurely, it notifies the client of the failure. The user-side library repeats the invocation on other servers for a number of retries, to avoid infinite invocations of broken functions.

Finite Resources. Cloud services provide the illusion of infinite resources, and thanks to the large scale, serverless platforms can guarantee a Service Level Agreement (SLA) of 99.95% (AWS [9], Azure [8], Google [14]). On the other hand, rFaaS is an example of opportunistic computing, and its range of operations is limited to the spare capacity of an HPC cluster - requests beyond the available hardware cannot be fulfilled. A failed allocation process can be repeated after a user-defined wait time, with exponential backoff time. This process decreases the contention on executors, but does not guarantee resource availability: repeated backoffs might never succeed when the cluster is close to full utilization.

We extend the fault tolerance for resource exhaustion through the internal allocation of executors that are private to the MPI job. This benefits applications that cannot accommodate sudden exhaustion of resources, due to emerging load imbalance. In distributed applications, processes that succeeded allocating public executors in rFaaS, offer compute capacity to others by launching a private executor. These private executors provide an option for the processes without serverless allocations to offload a part of their workload internally and distribute tasks equally across all workers. Contrary to cloud-based FaaS platforms that acquire new resources, rFaaS clients still use the same hardware. Speedup is thus achieved even when without enough remote resources for the entire application.
3.6 Isolation
In addition to bare-metal executors, we include containerized executors to ensure privacy and security in the multi-tenant execution in rFaaS. The main requirements imposed by rFaaS are virtualization support for RDMA-capable network controllers and negligible performance overheads. The current implementation uses Docker containers to implement isolated execution contexts for user functions. Use of Docker in HPC systems has raised security concerns, as the Docker daemon should not be accessible to non-trusted users [16], this is not a concern in rFaaS: the containers are launched by the executor manager, and users never have neither control nor access to the daemon. We limit the user’s code from accessing any resources, data, and code not provided with the invocation. We use the Single Root I/O Virtualization (SR-IOV) for the virtualization of network controllers in a multi-tenant environment. Virtual network functions provide isolated but high-performance access for different users [36]. Software implementations for virtualization come at the cost of non-negligible overheads and increased CPU usage [6, 61].

In principle, other solutions could be used: HPC-optimized containers such as Singularity [62] and Shift [43], and microVMs such as Firecracker [7, 20] that provide a higher level of isolation with negligible performance overheads. To integrate other sandboxes with rFaaS, virtualization or passthrough to the RDMA NIC must be provided, and new instances can be launched on-demand without interacting with the batch system.

3.7 rFaaS versus FaaS
While rFaaS implements the essential semantics of FaaS computing - remote invocations on transient and multi-tenant resources with pay-as-you-go billing - we tailor the design of the serverless platform to the HPC world. The trigger mechanism is replaced with decentralized allocations and direct connection to an executor, removing the proxies and caching the connection for subsequent invocations. While serverless platforms deploy function code from storage when instantiating a sandbox, the rFaaS client sends the code directly to memory, removing the latencies of external storage and sandbox I/O operations from the critical path.

rFaaS is a fundamental building block for fast and cheap invocations on idle resources. We do not provide a universal solution for resource starvation, as different applications require alternative approaches. Applications with tasking, fine-grained parallelism, and load imbalance can benefit from rFaaS even if not all workers succeed in resource allocations. On top of the standardized interface of rFaaS, additional features can be implemented according to the needs of specific programming frameworks: native datatypes and serialization, collective operations, logging of invocations. Similarly, rFaaS does not come with a dedicated authorization system, as this adds overhead and can be provided through the HPC batch systems.

4 rFaaS APPLICATION TARGETS
rFaaS can provide significant performance boosts to applications. The guiding principle — the application never waits for remote invocations to finish — is achieved by dividing the work such that the round trip time of invocations and the time they are executed are hidden by local work. Fine-grained invocations allow developers many opportunities to accelerate their applications. Consequently, the low-latency invocations are critical for such tasks, and latency plays a part in deciding what can be safely offloaded to rFaaS.

We use an analytical model to estimate the overheads of warm and hot invocations in rFaaS, based on prior work of LogP [33] and LogP [48] models. The network performance is expressed with parameters such as latency, CPU overhead on the sender and receiver, and gap factor. By learning the network parameters, estimating the remote function execution time, and measuring the rFaaS overheads (Section 6), we model the round-trip invocation time.

We design a model to decide when remote invocations can be integrated into HPC applications, then show how to use rFaaS as an accelerator for HPC problems. We provide examples of applications and benchmarks that are either a natural fit or can be adapted to use rFaaS to offload some of their work. This list is not exhaustive but provides an intuition on using rFaaS efficiently in practice.

Massively parallel applications. These applications are extremely malleable and can therefore make efficient use of rFaaS. A solver for the Black-Scholes equation [45] is a good example, as it generates many independent tasks of comparable runtime. Assuming we want to achieve the best possible performance, we measure the runtime of one task \( T_{local} \) and then compare this to the runtime \( T_{inv} \) of one invocation using rFaaS, to which we add the round-trip network time \( L \). There exists a number \( N_{local} \) of tasks such that:

\[
N_{local} \cdot T_{local} \geq T_{inv} + L
\]  

(1)

Therefore, if the number of tasks is greater than \( N_{local} \), up to \( N_{remote} \) tasks can be safely computed using rFaaS without incurring any waiting time. \( N_{remote} \) is determined as the number of tasks necessary to saturate the available bandwidth \( B \). Therefore, the throughput of the system only depends on the network link bandwidth, and the amount of work available to rFaaS.

Task-based applications with no sharing within tasks. Task-based applications are programs that consist of a series of tasks that must be executed, where some tasks can depend on the results of others, inducing a task dependency graph [80]— basically a graph stating the order in which tasks must be executed. Task-based applications can profit from rFaaS, as Eq. 1 still holds in this case. However, the number of tasks that can be offloaded at any time depends on the width of the task dependency graph at that time - the wider the graph, the more parallelism is exposed, and therefore more tasks can be transferred to rFaaS. As an example, we consider the prefix scan in electron microscopy image registration [29]. The width of...
the task graph in a distributed scan varies significantly between program phases, affecting the parallel efficiency. The dispatch of tasks expands parallel resources only when needed and limits the static resource allocation to the most efficient configuration.

Hybrid BSP Applications. A common occurrence in HPC are applications that use MPI and a shared-memory parallel programming framework, e.g., OpenMP, and we propose to boost the performance of such programs using rFaaS. We assume a bulk synchronous execution at the MPI level: each rank performs mostly the same type of computation. In this case, we consider candidates for remote invocations OpenMP parallel loops or task constructs with no shared accesses. By applying Eq. 1, we can determine how many loop iterations or tasks can be safely offloaded without incurring waiting times. Even if a small number of iterations or tasks are solved remotely, this can still amount to significant computational effort in total, as each MPI process performs the same work.

Since rFaaS has finite resources, the application must handle the case of resource starvation. While this is an application-specific problem, we provide a general principle of resource-aware serverless acceleration for BSP programs presented in Figure 7. In addition to public executors offered by the rFaaS service, MPI processes run private executors that allow for job-internal offloading. Thanks to the rigid structure of bulk synchronous computations, MPI ranks can mutually exchange their acceleration status before entering computational loops and find acceleration partners offering private acceleration for BSP programs presented in Figure 7. In addition to implementations proved that this concept is an efficient interface for dispatching tasks to accelerator devices [30, 46]. The programming model presented in Figure 9 hides the complexity of RDMA verbs under a lightweight C++ abstraction. It can be easily integrated into existing parallel applications as presented in Listing 1, and it can be adapted in the future to full compatibility with C++.

5 rFaaS as HPC-as-a-Service

To enable true HPC-as-a-Service through serverless computing, the functions must easily integrate with existing parallel applications and cluster management systems. They should not add a significant burden on administrators when deploying the new, dynamic and flexible computing platform and accounting system. To that end, we present an integration model of rFaaS into the MPI-centric world of supercomputers (Figure 8). To enable native and straightforward integration of fast serverless functions with high-performance applications, we design a C++ programming model for high degree of compatibility with other tasking systems (Sec. 5.1). rFaaS is not only a standalone serverless platform — it is designed as a pluggable component into existing cluster configurations (Sec. 5.3), and provides a simple yet effective accounting procedure (Sec. 5.4).

5.1 Programming Model

To design the programming interface for rFaaS, we take inspiration from recent developments in C++ standard for parallel and asynchronous executors [15]. The prior work executors and their implementations proved that this concept is an efficient interface for dispatching tasks to accelerator devices [30, 46]. The programming model presented in Figure 9 hides the complexity of RDMA verbs under a lightweight C++ abstraction. It can be easily integrated into existing parallel applications as presented in Listing 1, and it can be adapted in the future to full compatibility with C++.

Allocator. The allocator (①) provides memory allocation for RDMA-enabled memory buffers and encapsulates the memory region reserved for function header (②). The allocator can be integrated effortlessly to serialize standard containers such as std::vector and std::array, and the class ensures that all memory buffers are page-aligned to achieve the highest bandwidth on RDMA [59].

Invoker. The client’s invoker implements the submission of remote function invocations (③). It also manages RDMA connections...
MPI Integration. rFaaS programming model integrates natively with MPI applications. Each MPI process manages the remote function placements independently, as shown in Figure 8. Even if a large MPI run attempts to allocate resources simultaneously, the contention is avoided by randomly selecting computing resources.

The allocation of rFaaS functions can be performed ahead of time (7) to hide the cold invocation latencies since warm executor threads are sleeping and not incurring major charges. Remote resources can be allocated and deallocated as needed (3), adjusting to the varying parallelism of different phases of an MPI application. With the abstractions for work submission and memory management, rFaaS interface can be used by modifying only those parts of the application that benefit from its elastic parallelism (6).

5.2 Function Deployment

As a serverless solution, rFaaS supports execution of arbitrary stateless functions, and similarly to the function apps offered by Azure Functions [3], rFaaS enables execution of different functions in the same execution process. However, unlike other serverless platforms, we do not rely on external storage to provide function code during allocation, as it adds significant overhead and makes on-the-fly code updates difficult. Instead, in rFaaS the function code is written to executor memory during the cold initialization process (M1). The user selects a shared library submitted to the remote invocation. Both sides query and sort the library’s exported symbols, and during invocation, only the relative function index is submitted.

Listing 2 presents the standard function interface in rFaaS. Each thread allocates the memory regions and notifies the client about the address and access key for function’s input. To invoke the function (M2), its input is written to the provided remote buffer, and the immediate value contains a invocation identifier and a function index. Each invocation includes the size argument specifying the actual number of bytes written by the invoker. The function returns the number of bytes in the output array that will be sent back to the client. The input buffer contains a twelve-byte header containing the address and access key for a buffer on the client’s side, and the executor writes the output directly to the client’s memory using an RDMA write. Thus, users gain the flexibility to invoke functions that return results into different memory regions. The immediate value of the return write contains the execution status and the invocation identifier. The latter allows the client’s library to immediately identify which execution has just finished.

The library can include general-purpose static variables to store data between invocations, similarly to a sandbox's temporary filesystem. It is the user’s responsibility to ensure that functions preserve their stateless nature while accessing a shared resource. Additional data can be made available to the executor through a high-performance filesystem such as Lustre [85].

5.3 Batch Systems

Efficient utilization of idle cluster resources requires two basic functionalities from a serverless platform: a release of nodes for FaaS processing with an immediate announcement to all users, and a single-step removal of executors from the serverless resource pool. rFaaS implements those requirements in a simple interface designed for integration with cluster job management systems.

Resource release. The batch system offers unused CPU cores for the execution of short-running serverless functions. To that end, the global resource manager offers a single REST API call to register new resources (B1). The node information is sent to the resource manager, the manager adds the server to the list of resources, and publishes updates to all registered clients through RDMA operations. Thus, rFaaS users become aware of the newly available computing location in a microsecond-scale latency. This is required to support efficient allocations on spare capacity that can be available only for a very short period of time (Figure 2).

Resource retrieval. Idle nodes cannot be used as FaaS executors without a safe and low-latency abort functionality. Otherwise, the resources might not be freed efficiently for batch jobs with higher priority. Batch systems use REST API to send remove call with a parameter describing the allowed time for resource deallocation (B2). When the request is immediate (no further computing time is allowed), all active functions invocations are aborted, termination replies are sent to clients through existing RDMA channels, and the final billing update is sent to the resource manager. Otherwise, active invocations might be permitted to finish the computation if their remaining compute time is lower than the permitted time limit. No further invocations will be granted while the batch system retrieves resources and active connections are gracefully terminated.

5.4 Accounting

The pricing of rFaaS is presented in the equation below, and it includes two basic cost components: allocation of cluster resources $C_a$, and active computation time $C_c$. 

$$C = C_a \cdot t_a + C_c \cdot t_c$$

The total allocation $t_a$ measured in GB-second is calculated across all executors as a product of allocation time and memory requested. Whereas the total active computation time $t_c$ measured in seconds

```
void foo(void* in, uint32_t in_size, void* out) {
    double* input = reinterpret_cast<double*>(in);
    double* output = reinterpret_cast<double*>(out);
    // Application can be dispatched as shared library
    uint32_t out_len = solve(input, in_len, out); // Return value defines the output size
    return sizeof(double) * out_len;
}
```

Copik, et al.
represents the total time all remote workers are busy with computations. Costs $C_a$ and $C_c$ are measured in $\$/per GB-second and $\$/per second, respectively, and represent the total cost of occupying and actively using system resources (i.e., cores, memory) for a given period of time. The pricing system of rFaaS is similar to traditional FaaS systems for provisioned function invocations [12], where clients are charged for pre-allocation of cloud resources. However, unlike traditional FaaS, rFaaS does not charge for invocation calls, only charging for actual computation time. As a result, clients of rFaaS only pay $C_c$ for actual computation, not including times when remote executor threads sleep due to warm invocations.

On HPC systems, we propose that these elastic resources should be offered at a significant discount. This can ensure better utilization rates while not impacting the ability to schedule classical jobs, and acknowledging they do not offer the same guarantees of availability.

The accounting procedure is implemented in a global database associated with the resource manager (A2). The manager exposes memory regions for RDMA atomic *fetch-and-add* operations, providing executor managers with an RDMA-native way of accumulating cost results without consuming CPU resources. Similarly, executors use atomic operations to report billing updates to their local manager (A2). We accumulate charges with a granularity of one second, and billing data is updated after accumulating new charges larger than the granularity threshold. This avoids a loss of accounting data due to abrupt termination of rFaaS executor instances. Contention of atomic operations is not an issue, as cost accumulation is never on the critical path of function invocation.

6 rFaaS IN PRACTICE

To demonstrate the fitness for purpose of rFaaS for HPC, we answer critical questions in the form of extensive evaluation.

(1) Is rFaaS fast enough for HPC?

(2) Are the overheads for initialization prohibitively large?

(3) Does rFaaS scale with larger messages?

(4) Does rFaaS scale with more workers?

(5) Does rFaaS improve performance for a typical HPC task?

(6) Are very short rFaaS functions usable in HPC computations?

(7) Is rFaaS performance competitive against OpenMP?

**Platform.** For the purpose of this evaluation, we deployed rFaaS in a local cluster and execute benchmark code on 4 nodes, each with two 18-core Intel(R) Xeon(R) Gold 6154 CPU @ 3.00GHz and 377 GB of memory. Nodes are equipped with Mellanox MT27800 Family NIC with a 100 Gb/s Single-Port link that is configured with RoCEv2 support. Nodes communicate with each other via a switch, and we measured an RTT latency of 3.69 $\mu$s and a bandwidth of 11,686.4 MiB/s. We use Docker 20.10.5 with executor image ubuntu:20.04 and the Mellanox’s SR-IOV plugin to run containers over virtual device functions, Our software is implemented in C++, using g++ 8.3.1, and OpenMPI 4.0.5.

6.1 Invocation Latency

We begin with the most important characteristic for rFaaS: the latency of invoking a remote function. We measure hot and warm invocations of a non-op function with the same input and output size. We use a warmed-up, single-threaded, bare-metal executor with the main thread pinned to a CPU core, perform 10,000 repetitions, and report the median. We compute the non-parametric 99% confidence intervals of the median [47, 63], and find that the interval bounds are very tight (<1%). To assess the overheads of rFaaS invocations, we measure the latency of RDMA and TCP/IP transmissions. For the former, we use ib_write_lat from the perfest package, execute it with thread pinning and warm-up iterations, and report the median. For the latter, we use netperf with page-aligned buffers and process pinning, and report the mean.

The results for data sizes from 1 byte to 4 kB presented in Figure 10 show that the overhead of a no-op function in rFaaS in a process is 326 nanoseconds on average, when compared to RDMA writes. The measurements for a Docker-based executor present additional ca. 50 nanoseconds overhead over RDMA writes when using a container. The only exception is the message size of 128 bytes, where the overhead increases to 630 nanoseconds. There, RDMA can use message inlining for both directions of the transmission which improves the performance of small messages significantly [58]. However, the communication in rFaaS is asymmetrical: we transmit 12 bytes more for the input. The maximal supported inlining size is 128 bytes on our device, forcing rFaaS to use non-inlined write operations for one direction. The average overhead of a warm execution is 4.67 microseconds. Here, the containerization adds a measurable performance overhead, and Docker-based warm executions have an additional overhead of ca. 650 nanoseconds.

With slightly more than 300 nanoseconds of overhead, rFaaS enables remote function invocation with no noticeable performance penalty, conclusively answering: *rFaaS is fast enough for HPC.*

6.2 Cold Invocation Overheads

Figures 11a and 11b present the overhead of a single cold invocation on a bare-metal and Docker-based executor, respectively. The data comes from 1000 invocations with a single no-op C++ function, compiled into a shared library of size 7.88 kB. In all tested configurations, the longest step is the creation of workers. All other steps: the connection establishment to the manager, submitting an allocation and code, and code invocation, take single-digit milliseconds to accomplish. We can therefore claim that rFaaS does not introduce significant overheads apart from the sandbox initialization.

While the current version of Docker shows an overhead of approximately 2.7 seconds to spawn workers, approaches such as...
invocations are between 695x and 3,692x faster than AWS Lambda executions, thanks to the performance attainable with a low-latency network and the native support for transmitting raw data that suits high-performance applications very well. rFaaS is between 17x and 28x times faster than nightcore, another FaaS platform with microsecond-scale latencies. Similarly, rFaaS provides a speedup between 5,904x and 22,406x when compared to OpenWhisk.

Therefore, we answer that rFaaS provides significant performance improvements over contemporary FaaS platforms and rFaaS scales well with message size.

6.4 Scalability with parallel workers.
To verify that RDMA-capable functions scale efficiently to handle integration into scalable applications, we place managers on 36-core CPUs and evaluate the overheads associated with parallel invocations. We execute the no-op function on warmed-up, bare-metal executors having allocated from 1 to 32 worker threads.

Figure 12 presents the round-trip latencies for invoking functions with 1 kB and 1 MB payloads, respectively. The overhead of handling many concurrent connections is insignificant on hot invocations with a smaller payload. While the Docker executor shows performance increase (hot) and decrease (warm) on the 1 kB payload, the difference on 1 MB payload is less than 1%. However, execution times increase significantly with the number of workers when sending 1 MB data, due to saturating network capacity (100 Gb/s). This shows that rFaaS scaling is limited only by the available bandwidth.

Therefore, we claim that parallel scaling of rFaaS executors is bounded only by network capacity.

6.5 Use-case: matrix-matrix multiplication
To learn how much performance can be gained by offloading complex tasks to the spare capacity of HPC clusters, we use a matrix-matrix multiplication kernel as a stand-in for compute-intensive tasks in general, and compare the performance of a traditional MPI application with an elastic one that uses rFaaS acceleration.

We run an MPI application where each rank performs a matrix-matrix multiplication, averages it over 100 repetitions, and we measure the median kernel time across MPI ranks. MPI ranks are distributed across two 36-core nodes, and we pin each rank to a single core. Then, we deploy an MPI + rFaaS application where each rank
OpenMP + rFaaS
16 ranks 32 ranks 64 ranks
400 500 600 700 800
Matrix size
500 1000Time [ms]
MPI + rFaaS
MPI
16 ranks 32 ranks 64 ranks
400 500 600 700 800
Matrix size
0.5
1.0
1.5
2.0
2.5
Speedup versus MPI.$
MPI + nightcore
16 ranks 32 ranks 64 ranks
400 500 600 700 800
Matrix size
0.5
1.0
1.5
2.0
2.5
Speedup MPI + rFaaS
MPI
1000 2000
Linear system size
0
5
10
15
Speedup parallel
serial
Performance boost
through idle resources.

allocates a single bare-metal rFaaS function. rFaaS executors are deployed on two other 36-core nodes, and with such concentration of MPI and rFaaS computing resources, we show that sharing the network bandwidth does not prevent efficient serverless acceleration. Because of a high computation to communication ratio, we split workload equally, and both MPI rank and the function compute half of the result matrix. For a matrix size $N$, each invocation accepts two matrices with $N^2$ elements each, performs $2N^2$ floating-point operations, and returns a matrix with $N^2$ elements.

Figure 13a shows rFaaS provides a speedup between 1.88x and 1.94x depending on the number of MPI processes. This speedup is consistent as we vary the size of the multiplied matrices. We implemented the same function on the nightcore FaaS platform. As functions there accept only JSON arguments, we serialize matrices with C++ libraries [17, 18]. We deployed two instances with a fixed number of 32 function instances, and performed the same evaluation. The MPI + nightcore achieves worse speedup due to the overhead of serialization, and lower utilization of network bandwidth. Functions with a good ratio of computation to unique memory accesses can be accelerated with rFaaS. As long as this condition holds, rFaaS improves the performance of HPC workloads.

6.6 Use-case: linear solver
To show a serverless acceleration of a BSP-style problem, we consider the Jacobi linear solver, where a part of each iteration is offloaded to rFaaS. For a linear system of size $N \times N$, we perform approximately $2N^2$ floating-point operations in each iteration. The function receives in total $2N + N^2$ elements of the system matrix, right-hand vector, and the current solution approximation. With an equal split of workload, the function traverses half of the system and returns $\frac{2N}{3}$ elements. The $O(N^2)$ order of both communication and computation would require offloading a small fraction of the work to balance computation and communication. Instead, we perform a classical serverless optimization of caching resources in a warmed-up sandbox. Since the matrix and right-hand vector do not change between iterations, we submit them only for the first invocation. As long as the allocated function is not removed, we send only an updated solution vector in consequent iterations.

We evaluate the approach in the same setting as matrix multiplication (Section 6.5), with MPI ranks averaging Jacobi method with 1000 iterations over ten repetitions. Figure 13b demonstrates a speedup between 1.7 and 2.2 when rFaaS acceleration is used. Since each iteration takes just between 1 and 15 milliseconds, results must be returned with a minimal overhead to offer performance comparable with the main MPI process: the low-latency invocations in rFaaS apply to millisecond-scale computations.

6.7 Use-case: Black-Scholes
We show how rFaaS can accelerate a pure OpenMP application, using the Black-Scholes solver [45] from the PARSEC suite [24]. Black-Scholes solves the same partial differential equation for different parameters, and we dispatch independent equations to bare-metal parallel executors. We evaluate the benchmark on native input with approximately 229 MB of input and 38 MB of output and present the results in Figure 13c. We show that offloading the entire work to rFaaS scales efficiently compared to OpenMP, as long as the workload per thread is not close to the network transmission time of approximately 30 ms. We achieve further speedup by enhancing the OpenMP application with offloading half of the work to the same number of parallel workers on idle HPC resources. rFaaS offers scalable parallelism bounded by network performance only.

7 RELATED WORK
High-performance FaaS. FuncX [27] is a federated and distributed FaaS platform designed to bring serverless function abstraction to scientific computing. Nonetheless, FuncX does not take advantage of HPC networks and implements an hierarchical and centralized design, that has long invocation paths between clients and remote workers. As a result, even warm invocations take at least 90ms. Nightcore [54] is a FaaS runtime that offers microsecond-scale
warm invocations. Nightcore introduces the concept of internal function calls — invocations made by a running function. As a result, these internal function calls do not require inter-node communication and are satisfied locally. SAND [21] is a serverless platform optimized for workflows of serverless functions through grouping of functions and dedicated message buses for subsequent invocations. In contrast, rFaaS exploits co-location via explicit parallelism of executor allocation, and optimizes invocation latencies through RDMA communication. Archipelago [82] and Wukong [26] allow users to submit jobs that are represented as a directed acyclic graph (DAG) of functions. They perform latency-aware scheduling of a submitted DAGs: Wukong uses a decentralized and dynamic scheduling built on top of AWS Lambda, while Archipelago focuses on resource partitioning for decentralized schedulers and optimizing the control plane of a serverless platform. In contrast, rFaaS provides an HPC-centered FaaS platform where both allocation and invocation are decentralized and optimized with a direct client - worker connection. Application-layer solutions such as Wukong, and optimization strategies such as sandbox warming up in Archipelago, can be ported to rFaaS as well.

Elastic MPI. Raveendran et al. [76] proposed a framework for MPI programs that adapt to the elasticity of the cloud by restarting applications with different numbers of processes. Martin et al. [68] presented Flex-MPI, an automatic reconfiguration framework for malleable MPI applications. Huang et al. [49] presented Adaptive MPI, an MPI implementation in Charm++ with virtualization and reconfiguration. Other adaptive MPI solutions focus on checkpointing and migration [32, 37, 38]. In contrast, rFaaS concentrates on the dynamic acceleration of MPI programs with resources allocated on-the-fly, and it requires neither restarting nor reconfiguration of the MPI program to incorporate new parallel resources. rFaaS functions can be executed on transient resources that are available only for a short time. SR-IOV is a virtualization solution that offers high performance at lower resource costs.

ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 programme (grant agreement EPIGRAM-HS, No. 801039, and grant agreement RED-SEA, No. 955776), and from the Schweizerische Nationalfonds zur Förderung der wissenschaftlichen Forschung (SNF, Swiss National Science Foundation) through Project 170415. We would also like to thank the Swiss National Supercomputing Centre (CSCS) for providing us with access to their supercomputing machines Daint and Ault.

REFERENCES

8 CONCLUSIONS

This work is the first to show that FaaS computing, the new cloud model of stateless function invocations on dynamically allocated resources, can be enhanced with low-latency network protocols such as RDMA. We introduced hot serverless functions, a new invocation type that expands the guarantees of warm serverless functions with always-ready computing resources. We presented an integration of our serverless platform with MPI and batch processing, the de facto standard use patterns of supercomputers, to prove that high performance can be achieved with elastic resource management.

We conducted an exhaustive evaluation of RDMA-capable functions and demonstrated invocations with less than one microsecond of overhead, efficient parallel scalability, and processing latencies significantly lower than other FaaS platforms.

Overall, our results show that HPC applications can benefit from modern cloud programming paradigms to guarantee high performance at lower resource costs.