Software Resource Disaggregation for HPC with Serverless Computing

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

Marcin Chrapek  
ETH Zürich  
Switzerland

Larissa Schmid  
Karlsruhe Institute of Technology  
Germany

Alexandru Calotoiu  
ETH Zürich  
Switzerland

Torsten Hoefler  
htor@inf.ethz.ch  
ETH Zürich  
Switzerland

ABSTRACT

Aggregated HPC resources have rigid allocation systems and programming models which struggle to adapt to diverse and changing workloads. Consequently, HPC systems fail to efficiently use the large pools of unused memory and increase the utilization of idle computing resources. Prior work attempted to increase the throughput and efficiency of supercomputing systems through workload co-location and resource disaggregation. However, these methods fall short of providing a solution that can be applied to existing systems without major hardware modifications and performance losses. In this paper, we use the new cloud paradigm of serverless computing to improve the utilization of supercomputers. We show that the FaaS programming model satisfies the requirements of high-performance applications and how idle memory helps resolve cold startup issues. We demonstrate a software resource disaggregation approach where the co-location of functions allows idle cores and accelerators to be utilized while retaining near-native performance.

1 INTRODUCTION

Modern HPC systems come in all shapes and sizes, with varying computing power, accelerators, memory size, and bandwidth [51]. Yet, they all share one common characteristic: resource underutilization. Past predictions showed a pessimistic research outlook: “the goal of achieving near 100% utilization while supporting a real parallel supercomputing workload is unrealistic” [48]. Node utilization of supercomputer capacity varies between 80% and 94% on different systems [49, 68, 95], with up to 75% of memory is underutilized as these resources are overprovisioned for workloads with the greatest demands as can be seen in Fig. 2. A 10% decrease in monthly utilization can lead to hundreds of thousands of dollars of investment in unused hardware. This gap cannot be addressed with persistent and long-running allocations. HPC operators should incentivize users to use spare CPU cores or idle GPUs to accelerate their applications, improving the cost and energy efficiency of the system. To that end, users need fine-grained resource allocations and elastic programming models.

Furthermore, heterogeneity of HPC systems is increasing over time [51], with five more TOP500 systems using GPUs every year. In 2019, 28% of systems had accelerators. In November 2019, seven out of the top ten systems at TOP500 had a GPU, as did 14 out of the top 20 [4]. However, actual GPU utilization is often quite low. For example, on the Titan system, only 20% of the overall jobs used GPUs [87]. Furthermore, some applications that use GPUs make no use of the CPUs on the node, reinforcing a need to co-locate GPU and CPU jobs [51]. Resource disaggregation and job co-location are two techniques that aim to increase system throughput and enable fine-grained allocations.

Disaggregated resources are consolidated and allocated later in the exact amount needed by the application (Sec. 2.2). Disaggregation is used for specialized hardware [61] and it can improve memory’s performance–per–dollar by up to 87% [57]. While hardware-level memory disaggregation solutions are being developed [7, 57, 70], they require dedicated hardware and have high costs [7]. Instead, we propose a software system that does not require dedicated interconnects and extensions but runs on the HPC systems available today.

Sharing nodes by co-located jobs improves performance, throughput, and efficiency of HPC systems [45, 52]. However, space-sharing by applications that simultaneously stress the same shared resources leads to contention [31, 50]. Memory and I/O contention cause a slowdown of up to three times and several orders of magnitude, respectively [8, 51, 89], and many supercomputing systems disable node sharing for that reason. While job striping [14, 52] increases performance by spreading application processes and colocating them with other workloads, it requires understanding the symbiosis of co-located applications or partitioning shared sources (Sec. 2.3). Thus, new approaches are needed to reduce performance interference and provide multi-tenant security. With a flexible management and scheduling system, runtime adaptivity could reduce core-hour consumption by up to 44% in some applications [46, 62]. Sadly, the evolving and malleable applications [35] achieve lower
efficiency in the rigid systems and cause overallocation and under-utilization.

The solution to the problems outlined above can be found in HPC-as-a-Service [5], bringing the elasticity of cloud abstraction models to manage and access HPC resources. One of these models could be Function-as-a-Service (FaaS), a new paradigm that offers users a simple method of programming stateless functions. The cloud provider handles function invocations on dynamically allocated resources in a serverless fashion. The fine-grained allocations with pay-as-you-go billing could resolve the problem of runtime adaptivity in HPC. Yet, no work has fully embraced this cloud revolution to improve the efficiency of existing supercomputers.

We empower users and allow them to safely reclaim and use idle resources by colocating workloads in isolated containers. In this paper, using co-location as a starting point, we present the first FaaS system that implements software disaggregation of resources in a supercomputing system (Fig. 1). We show that dynamic function placement provides a functionally equivalent solution to disaggregated computing on homogenous resources (Fig. 3, Sec. 3). Our system allocates functions on idle resources while requiring changes to neither the hardware nor the operating systems. We then define the requirements that FaaS functions must fulfill to overcome the limitations of the classical, cloud-oriented functions, and show how a high-performance serverless platform rFaaS [24] can be adapted to the Cray supercomputers and containerization solutions common in HPC systems (Sec. 4). Finally, we present an HPC-centric programming model and integration for FaaS (Sec. 5). We use the pools of idle memory to host function sandboxes, reducing cold startups and increasing resource availability. We evaluate the new system on a set of representative HPC and disaggregated memory blades [57]. However, many methods have not been adapted to the Cray supercomputers and disaggregated memory blades [57]. However, many methods have not been adapted to the Cray supercomputers and disaggregated memory blades [57].

2 BACKGROUND AND MOTIVATION

Serverless provides a new resource allocation paradigm that can mitigate the low resource utilization (Sec. 2.1). Functions can provide a software approach to fine-grained allocations of disaggregated resources, overcoming the disadvantages of hardware solutions (Sec. 2.2). Functions can improve on the existing techniques and billing systems for co-locating workloads (Sec. 2.3).

2.1 Resource Utilization in HPC

Utilization of supercomputer capacity varies between 80% and 94% on different systems [49, 68, 95]. To assess the modern scale of the problem, we analyzed the utilization of the Piz Daint supercomputer, and disentangle the CPU and memory utilization in Figures 2a and 2b, respectively. The rapid and frequent changes indicate that resources do not stay idle long, and 70-80% of unallocated nodes stay idle for less than 10 minutes (Fig. 2c). This gap cannot be addressed with persistent and long-running allocations.

The aggregated and statically allocated computing nodes lead to wasting memory and network resources [61, 64, 69]. The average node memory usage can be as little as 24%, and 75% of jobs never utilize more than 50% of on-node memory. The average network and memory bandwidth utilization are very low, with occasional bursts of intensive traffic [61]. The memory system contributes roughly 10-18% of the appropriation and operational expenditures [9, 98]. While turning idle memory off could decrease the static energy consumption [69], it would also negatively affect memory parallelism [64]. Additionally, the contribution of memory in energy usage of datacenters has been decreasing in the last years [9]. Unfortunately, the problem of memory utilization is fundamentally not solvable with current static allocations on homogenous resources because these do not represent the heterogeneity of HPC workloads.

Heterogeneity of HPC systems is increasing over time [51], with five more TOP500 systems using GPUs every year. In 2019, 28% of systems had accelerators. In November 2019, seven out of the top ten systems at TOP500 had a GPU, as did 14 out of the top 20 [4]. However, actual GPU utilization is often quite low. For example, on the Titan system, only 20% of the overall jobs used GPUs [87]. Furthermore, some applications that use GPUs make no use of the CPUs, reinforcing a need to co-locate GPU and CPU [51].

2.2 Resource Disaggregation

Remote and disaggregated memory has been considered in data centers for almost a decade now [7, 28, 37, 38, 57]. Disaggregation replaces overprovisioning for the worst case with allocating for the average consumption but retaining the ability to expand resources dynamically. Remote memory has been proposed for HPC systems [69], but it comes with a bandwidth and latency penalty. While modern high-speed networks allow retaining near-native performance in some applications [37], remote memory is considered challenging for fault tolerance, and performance reasons [7].

Hardware-level solutions can elevate performance issues, e.g., by providing a dedicated high-speed network [70] and using dedicated memory blades [57]. However, many methods have not been adopted because of the major investments needed [38], such as changes in the OS and hypervisor, explicit memory management, or hardware support [55, 57, 57, 69]. Pricing models with dedicated memory billing are needed to avoid throughput degradation in HPC systems with resource disaggregation [96].

Resource disaggregation is not common in HPC because of performance overheads and increased complexity.
2.3 HPC Co-location

Co-location can help mitigate the underutilization problem by allowing more than one batch job to run on the same node. While some studies have not found a significant difference between node-sharing and exclusive jobs [81, 90], many applications experience performance degradation through contention in shared memory and network resources [31, 50]. Prior work has attempted to improve scheduling on a node by detecting sharing and contention in memory bus, bandwidth, and network interface [8, 53, 56, 82, 93]. When co-locating HPC workloads, it is essential to determine optimal node sharing and partition shared resources.

Node sharing Symbiotic applications can improve their performance when co-located [14, 82, 89], but determining which workload pairs show positive symbiosis is hard. Methods include user hints and offline experiments [88, 89], profiling and online monitoring [8, 53, 83], and machine learning [31]. For co-location, systems should select applications with different characteristics [52, 88, 89], and node oversubscription can provide further benefits [45, 92]. Another difficulty imposed by sharing is the unfairness of traditional billing models when applied to jobs with performance impacted by the interference [14, 15]. Finally, sharing introduces security vulnerabilities when tenants are not isolated. Co-located serverless functions provide isolation with the function sandbox.

Partitioning Partitioning shared resources can reduce the effects of negative interference [91]. Last-level cache (LLC) and memory bandwidth should be partitioned since cache contention is not the dominant factor in performance degradation [97].

Node sharing is beneficial for efficiency of HPC, as long as it avoids harmful interference. Short functions are good candidates for interference-aware co-location.

3 SOFTWARE DISAGGREGATION WITH FAAS

We focus on the three resources that can be disaggregated: CPU cores, memory, and GPUs. While targeting idle nodes is the first step, we want to go further and handle idle resources within active nodes. To achieve this goal, we enhance long-running jobs that often underutilize resources when exclusively occupying nodes. By co-locating them with short-term, flexible tasks with intensive but complementary resource consumption, we can take advantage of the different idle resources available. Serverless functions are perfect for co-location: they offer fine-grained scaling, multi-tenant isolation and are very easy to checkpoint, snapshot, and migrate.

To motivate users not to use nodes in exclusive mode, billing is adjusted to incentivize sharing of unused processors, GPUs, or memory. As in job striping, where users are encouraged to spread processes across a larger number of nodes to benefit from increased throughput, we recommend the same approach to leave at least one core free on each node to execute remote memory and GPU functions without introducing temporary oversubscription.

We discuss three major scenarios in which our software disaggregation approach mitigates resource underutilization. Scientific applications often have constraints on the number of parallel processes or the problem size beyond those imposed by the hardware. For example, LULESH [43] must use a cubic number of parallel processes. There, job configurations are unlikely to perfectly match the available number of cores per node and offer a natural opportunity to share the unused cores. Memory allocation grows cubically with the size of the problem, making it unlikely that all node memory will be used. Furthermore, the co-location of many MPI ranks executing LULESH application leads to contention in the memory subsystem [21], forcing users to spread processes across nodes.

3.1 Co-location - Sharing CPUs and more

We improve utilization by locating FaaS executors on idle cores in a node. Thus, our new serverless approach implements job striping, where MPI processes do not occupy an entire node and are co-located with other applications to better utilize resources [14, 52]. Functions can use the rest of the node’s resources while minimizing the performance impact on the batch application. Since FaaS functions are easy to profile and characterize, they can be...
matched with jobs that present different resource availability patterns. Even when resource consumption cannot be aligned, partitioning shared memory and CPU resources can provide the fairness needed for each application. Furthermore, short-running MPI processes are similar to FaaS functions (Sec. 5.2). Adaptive MPI implementations [20, 62] rescale applications by adding and removing processes on the fly, and new MPI ranks can be allocated in a serverless fashion. We demonstrate the benefits of co-locating such MPI processes with the example of the NAS benchmarks (Sec. 6.2).

### 3.2 Memory Sharing for Applications

In HPC, the memory usage of a job varies between processes and within the lifetime of the job, with a difference of up to 62.5x for some applications [98]. Furthermore, applications with poor scaling require gigabytes of memory per process, while capability computing can use less than 10% of available memory [98]. Therefore, HPC nodes will always have overprovisioned memory to support heterogeneous workloads, leading to much of the memory remaining idle. While high-memory jobs are not frequent in HPC systems, they still need to be accommodated, requiring memory reclamation to be short-term and ephemeral.

We propose two methods to effectively use idle node memory in HPC applications. First, we use free memory to keep FaaS containers warm and allow functions to be started quickly and efficiently, resolving an important issue of expensive cold starts in serverless (Sec. 4.2). Then, we offer other jobs the ability to run remote memory functions. Functions allocate a memory block of the desired size and expose remote memory access to idle memory, allowing HPC applications for remote paging [69]. Modern networks provide remote memory access with acceptable latencies [38], and the function-based approach offers fine-grained scalability with easily controllable lifetime and multi-tenant isolation.

The function is invoked by the user application, returns the remote memory location, and continues to run until the user explicitly terminates it. This, in turn, requires extending previous serverless communication limitations (Sec. 4.4). However, once the function is launched, the memory access interface is the same as in other RDMA-based disaggregation solutions. Since we offer one-sided remote memory access, such functions can be added to the system with minimal CPU overhead [38], allowing many remote memory functions to run on the same node and co-location with compute-intensive applications such as LULESH. When the batch system needs to reclaim idle memory, function containers can be migrated to other nodes and swapped to the parallel filesystem. The client library can make submitting functions seamless for the user, with functions running either directly from warm containers in otherwise idle memory or loaded from the swapped container if necessary.

### 3.3 GPU Sharing

![Diagram](image)

While the heterogeneity of HPC systems is growing, not every application can be modified to benefit from GPU acceleration. HPC systems should co-locate CPU-only and GPU-enabled jobs, as these are often complementary [51]. For example, the main version of LULESH does not use accelerators at all, instead relies on a hybrid implementation with OpenMP and MPI.

We disaggregate GPU and CPU resources by co-locating GPU functions. The function can be co-located with a CPU-only application, as it requires only a single CPU core to manage device and data transfers. Such functions can be launched with containers specialized for HPC systems (Sec. 4.3). Furthermore, the function can keep warm data in the device’s memory since there is no need to evict it unless there is an incoming job or another function that needs to use the device.

Although there exist systems for remote GPU access [30], they add latency to each command. However, applications such as machine learning inference can consist of hundreds of kernels with synchronization in between [86]. By running one GPU function process to ensure GPU access, we avoid adding inter-kernel latency in the remote GPU scenario.

### 3.4 Sharing Fairness

Sharing node resources introduces performance overhead caused by interference between these shared resources. This a concern for large-scale jobs because network and OS noise can significantly impact the scalability of applications [27, 41, 42]. However, the guarantee of exclusive access to on-node resources can be illusive, as jobs are affected by the inter-node sharing of network resources [13]. Thus, the disaggregation approach must consider only the contention on node resources, as network performance cannot be controlled by the HPC user.

Our disaggregation approach can use the existing fairness methods to minimize and compensate for performance losses associated with resource exhaustion and co-location overheads. External performance interference can be estimated [77], and dedicated pricing systems compensate users for performance losses [15]. When the application is sensitive to contention on memory resources, partitioning can constrain the co-located workload to ensure the proportional allocation of shared cache and bandwidth.

Furthermore, we propose that the disaggregation is applied selectively to workloads, and jobs are exempted since they allocate a large fraction of the entire system and can be sensitive to interference. Since many jobs use less than 256 nodes [49, 68], disaggregation can target small and medium-scale jobs first to increase system throughput, while not impacting scalability.
4 HPC FAAS RUNTIME

Serverless computing brings an abstract view of data center resources allocated on the fly by the provider and hidden from the user. This abstraction frees users from any responsibility for provisioning and allows for elastic computing, where users are billed only for the resources used. FaaS is the dominating programming model where users program and upload stateless functions to the cloud. However, classical cloud functions have been designed for the hardware and software stack common in the cloud. The situation changes in supercomputing systems with performance-oriented architecture and programming models. We map cloud functions into HPC environments and identify seven major issues that serverless faces in high-performance systems (Table 1). Based on these results, we define requirements that HPC functions must meet to become an efficient component of a high-performance application (Fig. 4).

rFaaS. To demonstrate how serverless functions can be used in the HPC context, we select and extend the serverless platform rFaaS [24]. rFaaS allows consecutive invocations to execute on the same resource allocated with a temporary lease. Furthermore, it employs a direct RDMA connection between the client and function executor, optimizing both the latency and the bandwidth of serverless. However, building a portable function environment on a supercomputing system is technically challenging, mainly due to changing software environments and the restricted execution model of batch jobs, designed primarily for static and long-running applications. We demonstrate that a serverless platform can be used even with batch systems that default towards exclusive jobs with implicit resource assignments that execute homogeneous applications. To that end, we present a specialization of the rFaaS platform to the Cray XC40/XC50 system Piz Daint [3].

4.1 Slow Warm Invocations

Problem When an invocation of a classical function is triggered, the payload is redirected to a selected function sandbox. Even a warm invocation in an existing sandbox can introduce dozens of milliseconds latency [23] due to a centralized rerouting of invocations that does not use high-speed network transport. However, functions must have microsecond-scale latency because the overhead of remote invocations can outweigh all benefits of computing with additional resources (Sec. 5.1).

Solution We achieve single-digit microsecond invocation latency by using fast networks and shortened critical path of invocation offered by rFaaS. To deploy rFaaS on a Cray system, we use the libfabric to target uGNI, the user network interface for Cray interconnects [73]. We faced two major problems: first, the libfabric installation within a container must be replaced with the main system installation to achieve the high performance and manage access to uGNI. We resolve the issue by manually mounting system directories in the container, as the available HPC containers do not support injection of libfabric at the runtime [58]. Then, network interfaces such as uGNI are designed to communicate within a single batch job, which is not the case for FaaS: client application in one job communicates with a serverless executor running as another batch job. To support serverless functions on the Cray system, we implement the allocation and distribution of Cray security credentials DRC [78].

4.2 Expensive Cold Starts

Problem When no existing sandbox can handle the invocation, a new one is allocated and initialized with an executor process running the user code. This cold start has a devastating effect on performance since it adds hundreds of milliseconds to the execution time in the best case [23, 59, 80]. Standard mitigation techniques include lightweight and prewarmed sandboxes and faster bootstrap methods [6, 29, 63], but the most common one is retaining containers for upcoming invocations. However, its effectiveness is limited as idle containers occupy memory and are purged frequently.

Solution Instead of decreasing negative cold start effects, we focus on reducing their frequency with the help of unutilized node memory. This solution is compatible with batch systems and fits the short-term availability of resources perfectly because idle containers can be removed immediately without consequences. The availability of CPU cores to handle invocations can be guaranteed by modifying allocations to keep one or two cores per node (out of the 30 or more) available. We modify the rFaaS resource management to remember the retained containers and adjust the allocation algorithm to target nodes with warm containers. Then, the cold start overhead is dominated by establishing RDMA connection and not by the expensive initialization of a new container.

4.3 Incompatible Container Systems

Problem Serverless in the cloud is dominated by Docker containers and micro virtual machines [6]. However, the adoption of containers has been constrained by security concerns, and virtual machines limit access to the accelerator and network devices. Containers must be run in the rootless mode to avoid privilege escalation attacks.
To support multi-tenancy on HPC nodes, these issues must be mitigated while retaining near-native performance. **Solution** Serverless sandboxes must be tailored to the needs of HPC functions, and we consider containers designed for scientific computing: Singularity [54] and Sarus [11]. Both provide native access to compute and I/O devices and integrate batch resource management (Table 2). Furthermore, containers provide native support for high-performance MPI installations with dynamic relinking of containerized applications. This enhancement is essential for HPC functions to support elastic execution of MPI processes.

### 4.4 Lack of a High-Performance I/O

**Problem** Classical serverless functions cannot accept incoming network connections in the cloud as they operate behind the NAT gateway. Instead, functions must resort to using persistent cloud storage, with latencies in the tens of milliseconds, and transmitting results back to the invoker — there is no high-performance I/O available to the functions in the data center ecosystem. However, HPC applications can produce terabytes of data, and in such applications, the transmission of results from a function to the invoking MPI process quickly becomes impractical. HPC applications need high-performance I/O operations that are offered through the scalable parallel filesystem [10], thus replacing the need for cloud storage. Furthermore, this environment is too restricted for remote memory functions that accept incoming connections and return the memory buffer information to the user without ending the invocation.

**Solution** First, we make the HPC parallel filesystem accessible by mounting high-performance partitions and allowing the function to access the user’s data. These allow the creation of persistent artifacts of function invocations, communicate large amounts of data between invocations, and bring serverless performance in line with what is expected of HPC applications. Then, we enhance the rFaaS invocation protocol with a portable interface for functions to start communication, accept incoming connections and return data without terminating the invocation, allowing HPC users to implement functionalities that do not fit the classical cloud model, such as serving remote memory to clients.

### 4.5 Incompatible Resource Management

**Problem** Serverless platforms support the allocation of CPU cores, and memory is allocated proportionally to CPU resources [1, 2]. However, software disaggregation techniques require large allocations of one hardware resource while not using another one extensively. Furthermore, using cluster resources requires two basic functionalities: a release of nodes for FaaS processing and removal of executors from the serverless resource pool.

**Solution** First, we extend the rFaaS resource management protocol with memory and GPU device availability. Computing and memory resources are allocated and billed independently: users configure memory size according to their needs and can add a GPU device. Since we are operating on reclaimed idle resources, there is no monetary loss coming from partial resource consumption by functions: every single allocation is an increase in system utilization.

Then, we implement an interface in rFaaS designed for integration with cluster job management systems (Fig. 5). The global resource manager offers a single REST API call to register resources (11), which are used immediately, supporting allocations on the spare capacity available only for a very short time (Fig. 2). Released resources include CPU cores, memory, and GPUs that have not been explicitly allocated by the tenant. Thus, the allocation policy becomes opt-in - resources not requested by the user are not assigned by default to their jobs.

Furthermore, we allow the batch manager to retrieve resources for batch jobs with higher priority. Batch systems use the REST API to send the remove call with a parameter describing the allowed time for resource deallocation (18). When the request is immediate (no additional computing time is allowed), all active function invocations are aborted, and termination replies are sent to clients. Otherwise, active invocations might be allowed to finish the computation, but no further invocations will be granted.

### 5 HPC FAAS PROGRAMMING MODEL

Serverless computing provides a performance boost through idle resources, but it needs model-driven incorporation into HPC applications. First, we propose to use fine-grained invocations to offload computations (Sec. 5.1). The guiding principle – the application never waits for remote invocations to finish – is achieved by dividing the work so that the network transport and computation times are hidden by local work. Consequently, the low-latency invocations are critical for such tasks, and latency plays a role in deciding what can be safely offloaded to a function. Second, we propose to run MPI applications as functions, providing a backend for short-running computations on idle hardware (Sec. 5.2).

<table>
<thead>
<tr>
<th></th>
<th>Docker</th>
<th>Singularity</th>
<th>Sarus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Format</td>
<td>Docker</td>
<td>Custom</td>
<td>Docker-compatible</td>
</tr>
<tr>
<td>Repositories</td>
<td>Docker registry</td>
<td>None</td>
<td>Docker registry</td>
</tr>
<tr>
<td>Devices support</td>
<td>Through plugins</td>
<td>Automatic</td>
<td>Automatic</td>
</tr>
<tr>
<td>Resources</td>
<td>Native, cgroup</td>
<td>Automatic</td>
<td>Automatic</td>
</tr>
<tr>
<td>Batch System</td>
<td>None</td>
<td>Slurm</td>
<td>Slurm</td>
</tr>
<tr>
<td>MPI Support</td>
<td>None</td>
<td>Native</td>
<td>Native</td>
</tr>
</tbody>
</table>

**Table 2:** Comparison of container systems for cloud and HPC [11, 54]. Automatic resource and device support in Singularity and Sarus are done via Slurm.

**Figure 5:** Co-location made easy: rFaaS functions running on batch-managed clusters.
5.1 Integration
We use an analytical model to estimate the overheads of rFaaS invocations, based on prior work on the LogP [26] and LogFP [40] 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 (Sec. 6.1), we model the round-trip invocation time.

We design a model to decide when remote invocations can be integrated into HPC applications, and then show how to use rFaaS as an accelerator for HPC problems. The model is applied to each offloaded task separately to support the varying computational and I/O requirements of heterogeneous applications. We provide examples of applications and benchmarks that are either a natural fit or can be adapted to use serverless offloading. 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 efficiently offload tasks as functions. A solver for the Black-Scholes equation [39] is a good example, as it generates many independent tasks with comparable runtime. Assuming that we want to achieve the best possible performance, we measure the runtime of each task $T_{\text{local}}$ and then compare this to the runtime $T_{\text{inv}}$ of one invocation using rFaaS, to which we add the round-trip network time $L$. Time $T_{\text{local}}$ can be obtained with offline profiling tools common in performance modeling workflows [17, 21], providing measurements and models for runtime decisions without the overhead of additional invocations. There exists a number $N_{\text{local}}$ of tasks such that:

$$N_{\text{local}} \cdot T_{\text{local}} \geq T_{\text{inv}} + L$$ (1)

Therefore, if the number of tasks is greater than $N_{\text{local}}$, up to $N_{\text{remote}}$ tasks can be computed remotely without incurring any waiting time. $N_{\text{remote}}$ 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.

Task-based applications with no sharing within tasks. Task dependency graph [79] specifies the order of execution and dependencies between tasks in a program, which can be offloaded using the guide in Eq. 1. However, the number of tasks that can be offloaded depends on the width of the task dependency graph - 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 [22]. The width of the task graph in a distributed scan varies significantly between program phases; thus, dynamic serverless offloading achieves higher efficiency than a static resource allocation.

5.2 MPI Functions
An HPC function can implement the same computation and communication logic as an MPI process. These can be allocated with lower provisioning latency than through a batch system, and use computing resources with short-time availability. When running in a sandbox, serverless functions can execute on a multi-tenant node, resolving one of the major security concerns that prevent node sharing in a production system. In the HPC context, FaaS can be more than just a backend for website and database functionalities; functions can represent full-fledged computations that involve communication and synchronization.

A further benefit can be provided with support for adaptive MPI implementations. These usually require infrastructure extensions to support elastic scaling [18, 20]. Instead, new MPI ranks can be scheduled as functions without going through the batch system. In HPC, FaaS brings low bootup times and flexible resource management to evolving and malleable jobs [35], desired traits to scale up in the application phase with adaptive parallelism.

6 CASE STUDIES
We evaluate our HPC software disaggregation approach in three steps, attempting to answer the following questions:

1. Can rFaaS offer low-latency invocations needed for HPC disaggregation?
2. What is the overhead of co-locating functions and batch jobs?
3. Can disaggregation improve system utilization?
4. Can HPC applications on a supercomputer benefit from serverless acceleration with rFaaS?

Before answering these questions, we first summarize our experimental setup.

Ault. We deploy rFaaS in a cluster and execute the benchmark code on nodes each with two 18-core Intel Xeon Gold 6154 CPUs @ 3.00GHz and 377 GB of memory. We use Docker 20.10.5 with executor image ubuntu:20.04, and our software is implemented in C++, using g++ 10.2, and OpenMPI 4.1.

Daint. We deploy rFaaS in a cluster and execute the benchmark code on nodes each with two 18-core Intel Xeon E5-2695 v4 @ 2.10GHz and 128 GB of memory. The GPU nodes have one 12-core Intel Xeon E5-2690 v3 @ 2.60GHz with 64 GB of memory, and a NVIDIA Tesla P100 GPU. All nodes are connected with the Cray Aries interconnect, and we implement a new backend in rFaaS with libfabrics to target the uGNI network communication library. We use Clang 12 and Cray MPICH. The current billing mode for Daint works at the level of entire nodes. Moving forward, a billing model at the granularity of individual cores would both incentivize users to only allocate the resources they require and allow multiple tenants to share nodes.

6.1 rFaaS on Cray Systems
To evaluate whether rFaaS provides the low-latency invocations needed in HPC (Sec. 4.1) we measure the round-trip time of function invocations on Piz Daint. We use the libfabrics backend that supports the uGNI provider for Cray systems. We evaluate a no-op function with different sizes of input and output data. We test the warm invocations that use non-busy waiting methods that have lower CPU overhead at the cost of increased latency, and the hot invocations that process invocations faster by continuously polling for new work.

We compare rFaaS using warm and hot using queue wait and busy polling methods, and show the results in Fig. 6. While warm
executors need more time to respond and are thus slower than the queue wait approach, the hot executions have comparable median performance to *libfabric* busy polling and even display a more stable behavior with fewer outliers.

*rFaaS* provides invocations that are fast enough for the integration of functions into HPC applications.

### 6.2 Co-location

**CPU Sharing.** To evaluate the overhead of co-locating applications by sharing CPUs, we use the LULESH [43] and MILC [12] applications as a classical batch job, using 64 MPI processes and various problem sizes. We deploy LULESH on 2 Piz Daint nodes, using 32 out of the 36 available cores. It’s important to note that LULESH can only run using a cubic number of processes, e.g., 8, 27, 64, 125, etc. Therefore, using all cores of a node is impossible in many configurations. Then, we run concurrently NAS benchmarks, with changes in LULESH performance explained by the measurement noise. More importantly, only requesting 32 out of 36 cores on each node translates to a core-hour cost reduction of \( \approx 11\% \), more than offsetting any impact of co-location. We evaluate the increased system utilization by comparing our co-location with two other scenarios: a realistic exclusive node allocation and an *ideali* allocation where small-scale jobs execute exclusively but are billed for used cores only. Figure 8 demonstrates significant utilization improvements of up to 52%.

While the performance loss on the function container is higher, it is not a limitation as HPC functions effectively support users with a way to use resources that would otherwise be wasted: computing a co-located FaaS-like application is free, and the only cost is performance overhead. We also propose that the cost of running co-located *rFaaS* jobs should be lower than that of classical batch jobs to incentivize reclaiming these resources and to offset the fact that such jobs have lower priority and might be preempted.
Figure 9: Overhead of batch jobs co-located with rFaaS functions providing remote memory. Reported mean with standard deviation over ten repetitions.

**Memory Sharing.** We evaluate the impact of allowing rFaaS to use idle memory. On the Ault system, we run LULESH using 27 and 125 cores, and MILC using 32 cores out of 36 available cores. We deploy rFaaS with the remote memory function setup in a Docker container. The rFaaS function allocates 1 GB of pinned memory available for RDMA operations, and returns the buffer data to the owner. While running LULESH and MILC, we perform RDMA read and write operations of 10 MB repeatedly with different intervals between repetitions to test how additional traffic affects performance (Fig. 9). The results show that LULESH is not sensitive to the variable perturbation, regardless of problem size, while MILC is more sensitive at larger problem sizes. When scaling LULESH to multiple nodes, the overall runtime of the job is affected minimally, proving that compute-intensive applications can share network bandwidth to improve the overall system throughput. Interestingly, the rate at which data is read or written does not affect performance even when adding 10GB/s of traffic to the system.

**GPU Sharing.** We also run the GPU version of LULESH and MILC on three GPU nodes of the Piz Daint system using 27 ranks and 9 cores out of the 12 available on each node for LULESH and 32 ranks (divided as 11, 11, and 10 cores) for MILC. Then, we run Rodinia GPU benchmarks [19] in a Sarus container (Fig. 10). These benchmarks simulate GPU functions as each only takes a few hundred milliseconds. The overall overhead remains very low (< 5%), with the exception of two outliers (6.1% and 10.5%) – both encountered only for the smallest problem size of LULESH. However, only requesting 9 out of 12 cores on each GPU translates to a core-hour cost reduction of 25%, yet again more than offsetting any impact of co-location. For MILC, the overhead is slightly higher, with the smaller problem sizes experiencing a stronger perturbation.

Co-locating batch jobs with rFaaS functions and FaaS-like HPC workloads does not introduce significant overheads in batch jobs, regardless of the resource being shared. Allocating only required resources leads to an overall reduction in costs for batch jobs, even taking co-location overheads into account.

6.3 HPC Integration

To prove that offloading computations to HPC functions in rFaaS offers performance competitive to that of parallel applications, we integrate HPC functions into OpenMP benchmarks executed on the Piz Daint supercomputer and the Ault cluster. We compare the runtimes of benchmarks using OpenMP parallelism with runs where the amount of resources has been doubled by allocating one function to each rank and process. Thus, we verify whether applications can be accelerated by offloading computations to cheap idle resources while constrained by the network bandwidth.

Thus, functions with a good ratio of computation to unique memory accesses can be accelerated with serverless functions, even if they require transmitting large inputs.

6.3.1 Use-case: Black-Scholes simulation. Figure 11a demonstrates an OpenMP Black-Scholes benchmark from the PARSEC suite modified to use rFaaS offloading. The serial execution takes 726 milliseconds on an input of 229 MB. We compare the OpenMP version against complete remote execution with rFaaS, and against doubling parallel resources with cheap serverless allocation. The application demonstrates that parallel computations can be efficiently offloaded...
Figure 11: rFaaS in practice, reported medians with non-parametric 95% CIs.

7 RELATED WORK

Resource Underutilization. Snavely et al. [14] proposed node sharing with co-location of applications with compatible resource consumption patterns. However, the detection and avoidance of performance interference is a major issue and requires changes to pricing models [15], batch systems, and schedulers [15, 36, 66, 84]. Instead, we propose a decentralized approach with fine-grained functions that does not require changes in batch systems and online monitoring for interference. Finally, idle memory in an HPC system can be used to duplicate memory contents for higher throughput [65].

Elastic MPI. Adaptive and elastic MPI frameworks implement restarting applications with different numbers of processes [75], reconfiguration frameworks [60], processor virtualization [44], and checkpointing with migration [25, 32, 33]. In contrast, functions bring a dynamic acceleration of MPI programs with resources allocated on-the-fly, and require neither restarting nor reconfiguring the MPI program to incorporate new resources.

Supporting malleable and evolving applications requires changes in schedulers and batch systems [18, 71, 72], and MPI extensions are needed to extend and shrink the number of processes [20]. Serverless functions can implement malleable and evolving jobs with high resource availability.

8 DISCUSSION

This paper proposes a functionally equivalent alternative to hardware resource disaggregation, achieved by co-locating a serverless platform with classical HPC batch jobs. In the following, we discuss several questions our approach raises.

How does our solution differ from cloud functions? While exploring secure multi-tenancy via serverless techniques is already new in the context of HPC, we go beyond that: we use co-location only as the starting point and leverage rFaaS to allow the different resource subsets to be accessed separately. Furthermore, unlike the multi-tenant co-location of functions in a cloud, we focus on providing access to different resource categories in the existing node model of an HPC data center.

What are the limitations imposed by rFaaS? The programming model in rFaaS is focused on offloading tasks to elastic executors, similarly to many other serverless approaches to parallel computing [47, 67, 85]. Our software disaggregation solution relies on having enough network bandwidth available to move tasks without incurring significant delays, as these reduce the benefits of parallelization (Sec. 5). Furthermore, MPI applications are adopted to support offloading to remote workers, a challenge faced by all applications that wish to use FaaS methods.

Can software disaggregation stay competitive against hardware solutions? Our approach can be used with off-the-shelf hardware and does not incur additional costs associated with hardware disaggregation. Furthermore, there is zero penalty for running an unmodified HPC application on an aggregated system, whereas disaggregation always adds latency to reach remote resources. Although emerging hardware disaggregation technologies can offer nanosecond-scale latency for remote memory access, the higher latency of remote memory in software disaggregation still can increase system throughput. Many high-performance cloud applications benefit from remote memory [37, 38, 74], indicating that a
software-based approach that does not require a dedicated interconnect can offer competitive performance at lower costs.

**Which applications benefit from co-location?** We demonstrate on two representative HPC applications that software disaggregation increases the system's utilization thanks to tolerable performance overheads. However, co-location has been shown to cause only minor slowdowns and increase overall system throughput in many HPC applications, including memory-bound and network-sensitive workloads [14, 16, 36, 84, 99]. In addition, they can take advantage of job striping and spreading [14, 84] that can be realized in our system due to the reduced costs of under-allocation.

9 **CONCLUSIONS**

HPC suffers from underutilization since many systems do not have access to hardware resource disaggregation. Therefore, we propose a software disaggregation approach to efficiently co-locate long-running batch jobs with serverless functions. We design targeted FaaS approaches for the three main domains of software disaggregation: idle processors, memory, and accelerators. Using a high-performance serverless platform, we demonstrate that the co-location of such workloads allows HPC users to benefit from reclaimed resources while minimizing performance losses, improving system throughput by up to 53% and supporting remote memory with up to 1GB/s traffic without negligible performance overheads. Finally, we provide users with a path to use the reclaimed resources to accelerate MPI and OpenMP applications.

**ACKNOWLEDGMENTS**

This project received funding from EuroHPC-JU under grant agreements DEEP-SEA, No 95560 and RED-SEA, No 055776. We thank the Swiss National Supercomputing Center (CSCS) for supporting us with compute infrastructure.

**REFERENCES**


