Software Resource Disaggregation for HPC with Serverless Computing

Marcin Copik\textsuperscript{1}, Marcin Chrapek\textsuperscript{1}, Alexandru Calotoiu\textsuperscript{1}, Torsten Hoefler\textsuperscript{1}
\textsuperscript{1}Department of Computer Science, ETH Zurich

Abstract—Aggregated HPC resources have rigid allocation systems and programming models and struggle to adapt to diverse and changing workloads. Thus, 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.

I. INTRODUCTION

Modern HPC systems come in all shapes and sizes, with varying computing power, accelerators, memory size, and bandwidth [1]. Yet, they all share one common characteristic: resource underutilization. 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” [2]. While the scale of reported computation utilization varies across systems, up to 75% of memory is underutilized as these resources are overprovisioned for workloads with the greatest demands (Sec. II-A). On such systems, a 10% decrease in monthly utilization rate leads to hundreds of thousands of dollars of investment in unused hardware. HPC operators should incentivize using idle resources to boost the cost and energy efficiency of the system. Users can then take advantage of spare CPU cores, idle GPUs or even unused memory to accelerate their applications. To that end, they need fine-grained resource allocations and elastic programming models.

Fine-grained resource management of tightly coupled resources in HPC is very difficult [3]. Memory bandwidth requirements differ by up to four times between applications [4], and memory and network bandwidths vary significantly between supercomputers [1]. Even the optimal number of threads is application-specific and “rarely equal to the number of cores on the processor” [5]. In the growing heterogeneity of HPC systems [6], applications have to adapt their resource allocations to avoid contention and low hardware utilization. In turn, this requires new solutions for schedulers, resource managers, and runtimes [7–9]. Resource disaggregation and job co-location are two techniques that aim to increase system throughput and enable fine-grained resource management.

Disaggregated resources are consolidated and allocated later in the exact amount needed by the application (Sec. II-B). Disaggregation is used for specialized hardware [6] and it can improve memory’s performance–per–dollar by up to 87% [10]. However, the communication needed to access remote resources impacts bandwidth and latency [3, 6] and remote memory comes with scalability and fault tolerance challenges [11]. While hardware-level memory disaggregation solutions are being developed [10–12], they require dedicated hardware and have high costs [11]. Instead, we propose a software system that does not require dedicated interconnects and extensions but runs on the HPC systems available today.

Fig. 1: Software disaggregation with FaaS: increasing resource utilization without modifications to the HPC hardware. Sharing nodes by co-located jobs improves performance, throughput, and efficiency [13, 14]. However, space-sharing by applications that simultaneously stress the same shared resources leads to contention [15, 16]. Memory and I/O contention cause a slowdown of up to three times and several orders of magnitude, respectively [1, 17, 18], and many supercomputing systems disable node sharing for that reason. While job stripping [5, 14] increases performance and throughput by spreading application processes and co-locating them with other workloads, it requires understanding the symbiosis of co-located applications or partitioning shared sources (Sec. II-C). 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 [19, 20]. Sadly, the evolving and mal-leafable applications [21] achieve lower efficiency in the rigid systems and cause overallocation and underutilization.

The solution to the problems outlined above can be found in HPC-as-a-Service [22], 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 abstracted and dynamically allocated resources in a **serverless** fashion. The fine-grained allocations with pay–as–you–go billing could resolve the problem of dynamic allocations in HPC. Yet, no work has fully embraced this cloud revolution to improve the efficiency of existing supercomputing systems.

We want to 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. III). Our system allocates functions on idle resources while requiring changes to neither the hardware nor the operating systems. Then, we define the requirements that HPC functions must fulfill to overcome the limitations of the classical, cloud-oriented functions, and show how **high-performance serverless platform** FaaS [23] can be adapted to the Cray supercomputers and containerization solutions common in HPC systems (Sec. IV). Finally, we present an HPC-centric **programming model and integration** for FaaS (Sec. V). 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 FaaS benchmarks (Sec. VI). To the best of our knowledge, our work is the first FaaS system that integrates into HPC applications to support evolving and malleable jobs.

Our paper makes following contributions:

- We adapt a high-performance FaaS platform to supercomputing systems, and show that HPC-native function invocations scale to thousands of cores.
- We present integration of FaaS into HPC batch scheduling system and the MPI programming model, and show how functions can be used to accelerate HPC applications.
- We introduce a novel co-location strategy for HPC workloads that merges system and uses pools of underutilized memory to host function sandboxes.

**Fig. 2**: Piz Daint utilization for a two week period in April 2022: querying SLURM with a two minute interval.

**Fig. 3**: **Software disaggregation**: co-location provides semantics of resource disaggregation on an unmodified system.

**II. BACKGROUND AND MOTIVATION**

**Serverless** provides a new resource allocation paradigm needed to resolve the issue of idle resources, as a rigid system cannot handle large-scale and small-scale jobs without sacrificing memory utilization (Sec. II-A). Functions can provide a software approach to fine-grained allocations of disaggregated resources, overcoming the disadvantages of hardware solutions (Sec. II-B). Functions can improve on the existing techniques and billing systems for co-locating workloads (Sec. II-C).

**A. Resource Utilization in HPC**

Recent surveys indicate that node utilization of supercomputer capacity varies between 80% and 94% on different systems [24–26]. To assess the modern scale of the problem, we analyzed the utilization of the Piz Daint supercomputing system [27] over a week, and present the CPU and memory utilization data in Figures 2a and 2b, 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.

The aggregated and statically allocated computing nodes lead to wasting both memory and network resources [3, 6, 28]. 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 [6]. The memory system contributes roughly 10–18% of the appropriate and operational expenditures [29, 30]. While turning idle memory off could decrease the static energy consumption [3], it would also negatively affect memory parallelism [28].
Additionally, the contribution of memory in energy usage of datacenters has been decreasing in the last years [30].

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. While capacity computing applications with poor scaling require gigabytes of memory per process, capability computing can use less than 10% of available memory [29]. The differences between MPI ranks and applications add further imbalance.

Heterogeneity of HPC systems is increasing over time [1], 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 [31]. However, actual GPU utilization is often quite low. For example, on the Titan system, only 20% of the overall jobs used GPUs [32]. 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 [1].

HPC resources are underutilized and overprovisioned. Batch jobs cannot use the idle computing resources of their short availability, and the diverse workloads force massive overprovisioning of memory.

B. Resource Disaggregation

Remote and disaggregated memory has been considered in data centers for almost a decade now [10, 11, 33–35]. Disaggregation replaces overprovisioning for the worst case with allocating for the average consumption but retaining the ability to expand memory dynamically. Remote memory has been proposed for HPC systems [3], but it comes with a bandwidth and latency penalty. While modern high-speed networks allow retaining near-native performance in some applications [35], remote memory is considered challenging for fault tolerance, scalability, and performance reasons [11].

Furthermore, many disaggregation methods have not been adopted because of the major investments needed [34], such as changes in the OS and hypervisor, explicit memory management, or hardware support [3, 10, 10, 36]. Pricing models with dedicated memory billing are needed to avoid throughput degradation in HPC systems with resource disaggregation [37]. Hardware-level solutions can elevate performance issues, e.g., by providing a dedicated high-speed network [12] and using dedicated memory blades [10].

Remote memory and resource disaggregation are not widely used in HPC because of performance overheads and increased complexity.

C. 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 jobs and exclusive jobs [38, 39], many applications experience significant performance degradation through contention in shared memory and network resources [15, 16]. Prior work has attempted to improve scheduling on a node by detecting sharing and contention in memory bus, bandwidth, and network interface [4, 18, 40–42]. 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 [5, 17, 40], but determining which workload pairs show positive symbiosis is not trivial. Methods include user hints and offline experiments [17, 43], profiling and online monitoring [4, 18, 44], and machine learning [15]. For co-location, systems should select applications with different characteristics [14, 17, 43], and node oversubscription can provide further benefits [13, 45]. Another difficulty imposed by sharing is the unfairness of traditional billing models when applied to jobs with performance impacted by the interference [5, 46]. Finally, sharing introduces security vulnerabilities when tenants are not isolated. Co-located serverless functions provide isolation with the function sandbox and accurate charging with the fine-grained billing.

Partitioning Partitioning shared resources can reduce the effects of negative interference. Last-level cache (LLC) can be partitioned to co-locate latency-critical workloads [47], even at the cache line granularity [48]. Efficient cache partitioning can be determined analytically [48] and with the help of online monitoring and hardware solutions [49, 50]. Memory bandwidth should be partitioned since cache contention is not the dominant factor in performance degradation [51]. Practical implementations of these techniques include Intel’s Cache Allocation Technology (CAT) and Memory Bandwidth Allocation (MBA) [52]. These are already used to improve co-scheduled applications’ performance [53].

Node sharing is beneficial for performance and efficiency of HPC, as long as it avoids harmful interference. Fine-grained and short functions can be good candidates for interference-aware co-location.

III. SOFTWARE DISAGGREGATION WITH FAAS

We focus on the three resources that can be disaggregated: CPU cores, memory, and GPUs (Fig. 4). Targeting idle nodes is the first step, and we want to go further and target idle resources within active nodes. To achieve this goal, we propose to enhance existing systems that execute long-running jobs since many of them underutilize resources when exclusively occupying nodes. These jobs are complemented by co-locating short-term, flexible tasks with intensive but complementary resource consumption - with different types of tasks taking advantage of the different idle resources available. Serverless functions are perfect for this goal: they offer fine-grained scaling, containerization provides multi-tenant isolation, and they are very easy to checkpoint, snapshot, and migrate.

To motivate users not to use nodes in exclusive mode, billing would need to be adjusted to incentivize not leaving any resources such as processors, GPUs, or memory unused.
For example, if all cores are used, it is not possible to utilize idle memory, regardless of the amount available. Furthermore, we encourage users to spread jobs out and leave at least one core free, which allows us to run remote memory and GPU functions.

We will now discuss different scenarios for how underutilization can appear and how our software disaggregation approach can mitigate it. Scientific applications often have constraints on the number of parallel processes or the problem size beyond those imposed by the hardware. For example, LULESH [54] must use a cubic number of parallel processes. Thus, job configurations are unlikely to perfectly match the available number of cores per node, and offers a natural opportunity for sharing the unused cores. Memory allocation grows cubically with the problem size, making it unlikely to use all node memory. Furthermore, the co-location of many MPI ranks executing the LULESH applications leads to a slow-down due to contention in the memory subsystem [55], forcing users to spread processes across more nodes and leave some cores idle.

A. Co-location - Sharing CPUs and more

We improve utilization by locating FaaS executors on idle cores in a node (Fig. 4a). Thus, this new serverless approach implements job stripping, where MPI processes do not occupy an entire node and are co-located with other applications to utilize resources better [5, 14].

With functions, we can use the rest of the node’s resources while ensuring the performance of the applications are unimpeached. Since FaaS functions are easy to profile and characterize, they can be matched with batch jobs that present different resource availability patterns. Even when resource consumption cannot be aligned with other applications, partitioning of shared memory and CPU resources can provide fairness needed for each application. Furthermore, short-running MPI processes are similar to FaaS functions (Sec. V-B). Adaptive MPI implementations [19, 56] can already dynamically rescale by adding or removing processes on the fly, and these can be allocated in a serverless fashion. We demonstrate the benefits of co-locating such MPI processes with NAS benchmark applications sharing node with LULESH (Sec. VI-B).

B. Memory Sharing for Functions and Applications

In the HPC, memory usage of a job varies across processes and within the job lifetime, with up to 62.5x difference for some applications [29]. Furthermore, applications with poor scaling require gigabytes of memory per process, while capability computing can use less than 10% of available memory [29]. Therefore, HPC nodes will always have over-provisioned memory to support the 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 for higher performance of 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. IV-B). Then, we offer other jobs the ability to run remote memory functions (Fig. 4b). Functions allocate a memory block of desired size and expose remote memory access to idle memory, allowing HPC applications for remote paging [3]. Modern networks provide remote memory access with acceptable latencies [34], 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 remote memory location, and continues running until terminated explicitly by the user. This, in turn, requires extending previous serverless communication limitations (Sec. IV-E). However, once the function is launched, the interface for memory access 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 [34], 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 the idle memory, function containers can be migrated to other nodes and swapped to the parallel filesystem. The client library can make submitting functions seamless to the user - with functions either running directly...
A. Slow Warm Invocations

When an invocation of a classical function is triggered, the request with the payload is redirected to a sandbox to execute the function. Even such a warm invocation in an existing sandbox can introduce dozens of milliseconds latency [59] due to a centralized gateway handling the rerouting of invocations [60] and not using high-speed network transport [23]. However, functions must come with a microsecond-scale latency because the overhead of remote invocations can outweigh all benefits of computing with additional resources (Sec. V-A).

(Solution) We achieve single-digit microsecond invocation latency by using fast networks and a decentralized protocol with no proxies on the invocation’s critical path. Similar to MPI implementations, the serverless platform needs to make use of the features the interconnect has available in order to offer competitive performance. Therefore, we extend the existing rFaaS implementation with OFED support with uGNI, the user network interface for Cray interconnects [61].

B. Expensive Cold Starts

When no existing container or virtual machine can handle the triggered 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, as it adds hundreds of milliseconds of overhead to the execution in the best case [59, 62, 63]. Serverless systems attempt to minimize the frequency and severity of cold startups by pre-warming containers, using lightweight virtual machines, and implementing faster bootup methods from checkpoints and snapshots [60, 64, 65]. The most common mitigation technique is retaining containers to handle consecutive and frequent invocations. However, its effectiveness is limited as idle containers occupy memory and containers must be purged frequently.

(Solution) Instead of decreasing negative cold start effects, we focus on reducing their frequency with the help of unutilized memory resources. This solution is compatible with how batch systems operate: serverless containers fit the short-term availability of resources perfectly because idle containers can be removed immediately with no consequences. The availability of computing cores for the container can be guaranteed: allocations can be modified to keep one or two cores per node (out of the 30 or more) available. Furthermore, functions can be scheduled on busy nodes with oversubscription, which usually increases the system’s throughput [13]. We modify rFaaS resource management to contain information on retained containers and adjust the allocation algorithm to target nodes with warm containers available. Then, the cold start overhead is dominated by establishing connections and not by the expensive initialization of a new container.
C. Incompatible Container Systems

Serverless in the cloud is dominated by Docker containers and micro virtual machines [60]. However, the adoption of containers has been constrained by security concerns, and virtual machines do not enable access to the accelerator and network devices. Primarily, containers must be executed in a 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 for the needs of HPC functions, and we consider containers designed for scientific computing: Singularity [66] and Sarus [67]. They both provide native access to compute and I/O devices and integrate the batch resource management (Table I). Furthermore, the 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.

D. Lack of a High-Performance I/O

The isolated environment of classical cloud functions provides ephemeral storage only. 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 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 [68], thus replacing the need for cloud storage.

(Solution) We make the HPC parallel filesystem accessible to serverless functions. When deploying a function, we mount high-performance partitions and allow the function to access the user’s data. These allow the creation of persistent artifacts of function invocations and communicate large amounts of data between invocations, and brings serverless performance in line with what is expected of HPC applications.

E. Restricted Communication

Classical serverless functions cannot accept incoming network connections in the cloud as they operate behind the NAT gateway. Another essential restriction in function communication is the single-step return of results at the end of an invocation. In particular, this environment is too restricted for remote memory functions that accept incoming RMA connections. Furthermore, such a function needs to return the memory buffer information to the user without being forced to end the invocation.

(Solution) The restriction on incoming connections can be lifted for HPC functions. We enhance the rFaaS invocation protocol with a portable interface for functions to start communication and accept incoming connections, allowing functions to implement functionalities such as serving remote memory to clients. Furthermore, we implement a new invocation type that allows the function to return data but continue the invocation, allowing HPC users to implement functions that do not fit into the classical cloud model.

F. Homogenous Resource Management

Serverless platforms support the allocation of CPU cores, and memory is allocated proportionally to CPU resources [69, 70]. However, each of the three software disaggregation techniques requires large allocations of one hardware resource while not using another one extensively. Therefore, the default billing model is insufficient for the heterogeneous resources in a supercomputing system.

(Solution) We extend the resource management protocol with memory and GPU device availability. Clients use the same random allocation protocol as in rFaaS, but they prioritize executors supporting the requirements of their functions. 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.

G. Batch System Integration

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.

(Solution) We implement those requirements in a simple interface in rFaaS designed for integration with cluster job management systems (Fig. 6). The global resource manager offers a single REST API call to register resources with the FaaS manager(B1). 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 (Fig. 2). Released resources include CPU cores, memory and accelerators that have not been allocated explicitly by the tenant. Thus, the resource policy becomes opt-in - memory and GPU devices 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 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.

V. FaaS PROGRAMMING MODEL

Serverless computing can provide a performance boost through idle resources, but it needs model-driven incorporation into HPC applications. First, we propose to use the fine-grained invocations to offload computations and accelerate applications (Sec. V-A). The guiding principle – the application never waits for remote invocations to finish — is achieved by dividing the work such 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 part in deciding what can be safely offloaded to a function. Second, we show that some MPI applications can be executed as functions, providing a backend for short-running computations on idle resources and adaptive MPI implementations (Sec. V-B).

A. Integration

We use an analytical model to estimate the overheads of warm and hot invocations in rFaaS, based on prior work on the LogP [71] and LogP [72] 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 VI), 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. The model is applied to each offloaded task separately to support the varying computational and I/O requirements of heterogenous applications. 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.

a) Massively parallel applications: These applications are extremely malleable and can therefore make efficient use of rFaaS. A solver for the Black-Scholes equation [73] 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 $L$. The time $T_{local}$ can be obtained with offline profiling tools common in performance modeling workflows [55, 74], providing measurements and models for runtime decisions without the overhead of additional invocations. There exists a number $N_{local}$ of tasks such that:

$$N_{local} \cdot T_{local} \geq T_{inv} + L$$

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 $B$: $\frac{B}{Data_{inv}}$. Therefore, the throughput of the system only depends on the network link bandwidth, and the amount of work available to rFaaS.

b) 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 [75] - 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 [76]. 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.

B. MPI Functions

A serverless 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 sandboxed environment, serverless
functions can execute on a multi-tenant node, resolving one of the major security concerns that prevent node sharing in a production system. We do not have to think about FaaS as merely an implementation backend for website and database functionalities. In the HPC context, functions could represent full-fledged computations.

A further benefit can be provided with support for adaptive MPI implementations. These usually require infrastructure extensions to support elastic scaling [9, 56]. Instead, new MPI ranks can be scheduled as functions without going through the batch system. FaaS computing brings low bootup times and flexible resource management, desired traits when scaling the application up to benefit from an application phase with an increased level of parallelism.

VI. 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 HPC applications benefit from serverless acceleration with rFaaS?

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

a) Ault: We deploy rFaaS in a cluster and execute the benchmark code on nodes each with two 18-core Intel Xeon 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. We measure a latency of 3.69 µs and a bandwidth of 11.69 GiB/s between nodes. 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.

b) Daint: We deploy CPU and GPU co-location jobs on the supercomputing system Piz Daint [27]. The multi-core nodes have 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 mod 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.

A. rFaaS

To evaluate whether rFaaS provides the low-latency invocations needed in HPC (Sec. IV-A), 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. 2. 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 libfabrics busy polling and even display a more stable behaviour with fewer outliers.

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

B. Co-location

CPU Sharing. To evaluate the overhead of co-locating applications by sharing CPUs, we use the LULESH [54] and
MILC [77] 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 and so on. Therefore, using all cores of a node is impossible in many configurations. Then, we run concurrently NAS benchmarks in the Sarus container on the remaining cores, using CPU binding of tasks. Many NAS benchmark applications have a short runtime, and thus represent a FaaS-like workload in HPC.

We run the NAS benchmarks with 1, 2, 4 and 8 MPI processes, spread equally across two nodes, and launch new executions as soon as the previous ones finish.

Fig. 8 shows that the impact of co-location on the batch job with this workload is negligible, 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 ≈ 11%, more than offsetting any impact of co-location. We evaluate the increased system utilization by comparing our colocation with two other scenarios: a realistic exclusive node allocation, and a partial allocation where small-scale jobs are billed for unused cores only. Figure 9 demonstrates significant utilization improvements of up to 44%, even in comparison to a partial co-location that is not usually supported in HPC clusters.

While the performance loss on the function container is higher, it is not a limitation as HPC functions effectively provide users with a way to use resources that would otherwise be wasted. 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.

Memory Sharing. We evaluate the impact of allowing rFaaS to use idle memory. We run LULESH and MILC on one Ault node, using 27 and 32 cores respectively 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. 10). 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. 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 [78] in a Sarus I container (Fig. 11). 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.

C. MPI Integration

![Graph](image1.png)

**Fig. 12:** rFaaS in practice, reported medians with non-parametric 95% CIs.

To prove offloading computations to rFaaS offers performance competitive with MPI we compare the runtimes of well-scaling benchmarks using MPI exclusively with runs where the same amount of resources used for MPI are added to rFaas. We expect the speed-up will be close to ideal.

1) **Use-case: local 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 job-wise, 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 allocates a single bare-metal rFaaS function. rFaaS executors are deployed on two 36-core nodes, and we show that sharing the network bandwidth does not prevent efficient serverless acceleration. Because of a high computation to communication ratio, we split the workload equally, and both MPI rank and the function compute half of the result matrix. For a matrix of size $N$, each invocation accepts two matrices with $N^2$ elements each, performs $N^3$ floating-point operations, and returns a matrix with $N^2$ elements.

Figure 12a 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. 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.

2) **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 the workload, the function traverses half of the system and returns $\frac{N^2}{2}$ 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 subsequent iterations.

We evaluate the approach in the same setting as the matrix multiplication example (Section VI-C1), with MPI ranks averaging Jacobi method with 1000 iterations over ten repetitions. Figure 12b 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:
3) **Use-case: Black-Scholes simulation:** Figure 12c demonstrates an OpenMP Black-Scholes benchmark from the PARSEC suite modified to use rFaaS offloading. The serial execution takes 726 milliseconds. We compare the OpenMP version against a complete remote execution with rFaaS and doubling parallel resources with cheap serverless allocation. The application demonstrates that parallel computations can be efficiently offloaded until network saturation is reached.

The **low-latency invocations in rFaaS apply to millisecond-scale computations.**

VII. RELATED WORK

a) **Resource Underutilization:** Snively et al. [5] proposed job stripping to enable node sharing through a co-location of applications with compatible resource consumption patterns. However, detection and avoidance of negative performance interference is a major issue and solutions proposed include fair pricing models [46], new batch scheduling algorithms [79, 80], and using resource partitioning [81]. 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 HPC system can be used to duplicate memory contents for higher throughput [82].

b) **Elastic MPI:** Ravendran et al. [83] proposed a framework for MPI programs that adapts to the elasticity of the cloud by restarting applications with different numbers of processes. Martin et al. [84] presented Flex-MPI, an automatic reconfiguration framework for malleable MPI applications. Huang et al. [85] presented Adaptive MPI, an MPI implementation in Charm++ with virtualization and reconfiguration, that uses processor virtualization and reconfiguration for load balancing. Other adaptive MPI solutions focus on checkpointing and migration, including an application-level scheme [86], and a dedicated runtime [87, 88]. In contrast, functions bring dynamic acceleration of MPI programs with resources allocated on-the-fly, and require neither restarting nor reconfiguration of the MPI program to incorporate new parallel resources.

Prabhakaran et al. [8, 89] propose extensions to schedulers and batch systems that support fair and dynamic allocations for malleable and evolving applications which have evolutionary resource requirements. Comprès et al. [56] propose extensions of MPI for invasive programming where applications can extend and shrink the set of MPI processes, with a later support for adaptive batching in SLURM [9]. Serverless functions can implement both malleable and evolving jobs with high resource availability.

c) **Active messages and remote procedure calls:** These approaches [90, 91] asynchronously run short functions at the destination in order to fully utilize available network performance by reducing operating system overheads and providing direct access to network devices.

Serverless functions share similar goals — using the serverless semantics of invoking functions on dynamically allocated and abstracted executors. The pay-as-you-go billing system allows us go further than active messages and enable general-purpose computing on shared nodes with multiple clients running different functions.

d) **High-performance serverless platforms:** While other works propose high-performance serverless platforms with low-latency scheduling [92–95], these are not targeted for HPC systems. FuncX [96] is a federated and distributed FaaS platform designed to bring serverless into scientific computing. Nonetheless, FuncX does not take advantage of HPC networks and implements an hierarchical and centralized design with long invocation paths in dozens of milliseconds.

VIII. DISCUSSION

This paper proposes a functionally equivalent alternative to hardware resource disaggregation: a method that offers software resource disaggregation by co-locating a serverless platform with classical HPC batch jobs. 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. Unlike the multi-tenant co-location of functions in a cloud datacenter, we focus on providing access to different resource categories in the existing node model of an HPC datacenter. 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 datacenter, we focus on providing access to different resource categories in the existing node model of an HPC datacenter.

**What are the limitations imposed by rFaaS?** The programming model of rFaaS is focused on offloading computational tasks to elastic executors, similarly to many other serverless approaches for parallel computing [97–99]. 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. V).

Furthermore, the MPI applications are adopted to support offloading to remote workers - a challenge faced by all applications that wish to use FaaS methods. Offloading MPI processes directly to serverless functions would require a fault-tolerant MPI implementation that can support peer-to-peer communication, which is orthogonal to our research direction.

**Can software disaggregation stay competitive against hardware solutions?** Our software disaggregation can be used with off-the-shelf hardware and does not incur additional costs associated with hardware disaggregation solutions. Furthermore, there is zero penalty for running an unmodified HPC application on an aggregated system whereas disaggregation always adds latency to reach to remote resources. While 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’s throughput. Many high-performance cloud applications benefit from remote memory [34, 35, 100], 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, colocation has been shown to cause only minor slowdowns and increase the overall system’s throughput on many HPC applications, including linear algebra, simulations and graph analytics [5, 80, 101–103], including memory-bound and network sensitive applications. These applications benefit from the colocated execution and a tailored resource allocation. In addition, they can take advantage of job stripping and spreading [5, 80] that can be realized in our system thanks to lowered costs of under allocation.

The slowdowns introduced by colocation can lead to performance deviations on large-scale jobs. Thus, batch systems and HPC administrators should prioritize small–scale jobs for this task. Nevertheless, the influence of noise and slowdowns on the communication delays has been well understood [104], and large-scale jobs can be additionally compensated for the interference resulting from a colocated run. Furthermore, our disaggregation can be composed with existing methodologies for detecting interference and fair pricing models for colocated batch jobs [46].

IX. Conclusions

Underutilized resources are an issue in HPC systems, given that 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 discuss three major domains of software disaggregation: idle processors, memory, and accelerators, and we design targeted FaaS approaches for each resource class. 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 from node sharing. Finally, we show how serverless functions can be used to accelerate MPI applications, provide users with a path to use the resources reclaimed with our approach.

References


[88] K. El Maghraoui, T. J. Desell, B. K. Szymanski, and


