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, heterogeinity 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

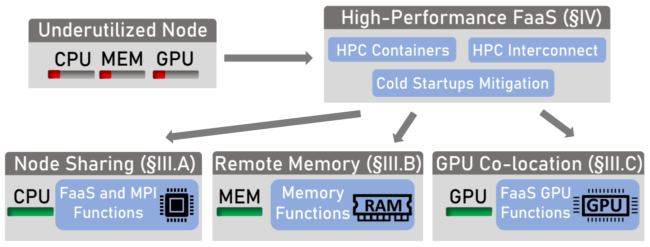


Figure 1: Software disaggregation with FaaS: increasing resource utilization without modifications to HPC hardware.

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 co-locating 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 underutilization.

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 HPC 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 FaaS benchmarks (Sec. 6). To the best of our knowledge, our work is the first integration of FaaS into HPC applications to support evolving and malleable jobs.

Our paper makes the following contributions:

- We introduce a novel co-location strategy for HPC workloads that improves system utilization and uses pools of underutilized memory to host function sandboxes.
- We adapt a high-performance FaaS platform to supercomputers and demonstrate the efficiency of HPC functions.
- We present an integration of FaaS into the HPC batch scheduling system and the MPI programming model and show how functions can be used to accelerate HPC applications.

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. While capacity computing applications with poor scaling require gigabytes of memory per process, capability computing can use less than 10% of available memory [98]. The differences between MPI ranks and applications add further imbalance.

Heterogeinity 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].

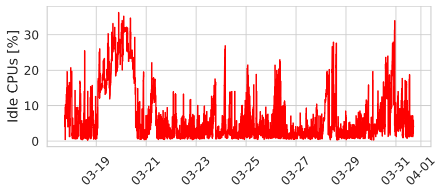
HPC resources are underutilized and overprovisioned. Batch jobs cannot use the idle computing resources due to their short availability, and the diverse workloads force node overprovisioning.

2.2 Resource Disaggregation

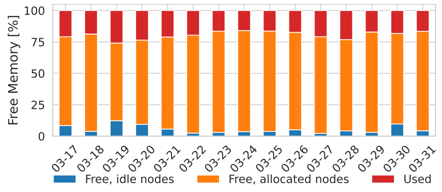
Remote and disaggregated memory has been considered in data centers for almost a decade now [7, 28, 37, 38, 57]. Disaggregation replaces overprovisioning for the worst case with allocating for the average consumption but retaining the ability to expand resources dynamically. Remote memory has been proposed 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].

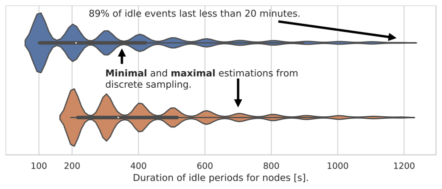
Resource disaggregation is not common in HPC because of performance overheads and increased complexity.



(a) Idle CPU cores rate (%).



(b) Free memory rate (%).



(c) Duration of idle periods for nodes.

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

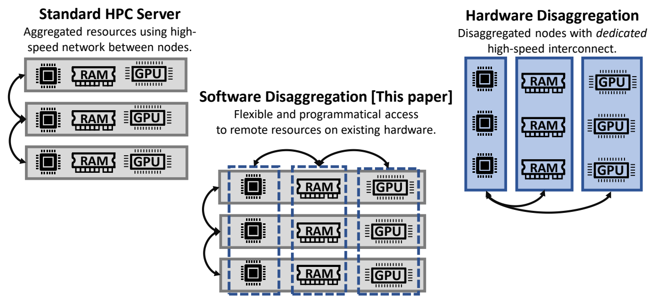

Standard HPC Server
Aggregated resources using high-speed network between nodes.
Software Disaggregation [This paper]
Flexible and programmatical access to remote resources on existing hardware.
Hardware Disaggregation
Disaggregated nodes with dedicated high-speed interconnect.
RAM
GPU

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

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.

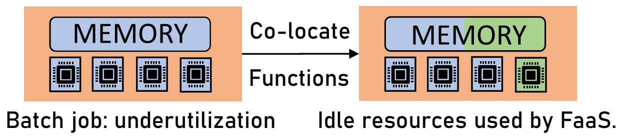
3 SOFTWARE DISAGGREGATION WITH FAAS

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.

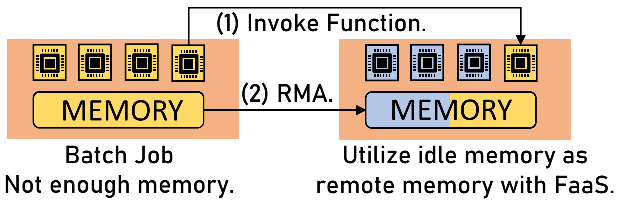
3.1 Co-location - Sharing CPUs and more



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

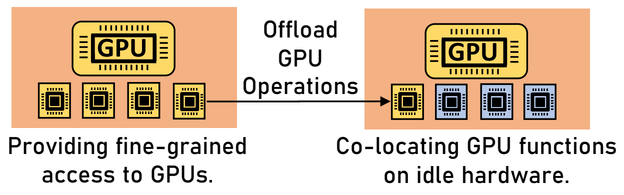


In HPC, the memory usage of a job varies between processes and within the lifetime of the job, with a difference of up to 62.5x for some applications [98]. Furthermore, applications with poor scaling require gigabytes of memory per process, while capability computing can use less than 10% of available 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



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 CPU 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 *hero* 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.

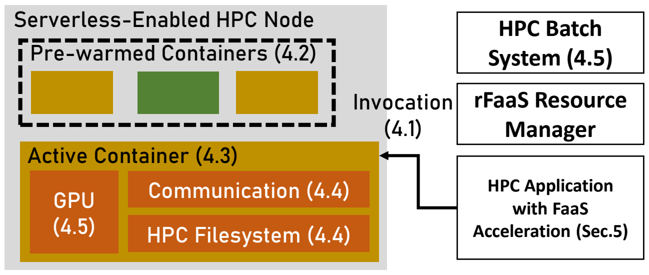


Figure 4: Specializing serverless platform for HPC requirements.

	Cloud FaaS	HPC FaaS
Network	TCP	**uGNI**, ibverbs, AWS EFA
Sandbox	Docker, microVM, unikernels	Singularity, **Sarus**
Storage	Object, block	Parallel file system
Communication	Storage, DB, queue	Direct communication
Placement	VMs, Kubernetes	Batch jobs on HPC nodes

Table 1: Comparison of *cloud functions* environments with *HPC functions*. Technologies used in specialization for Cray machines are in bold.

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 bootup 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 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.

	Docker	Singularity	Sarus
Image Format	Docker	Custom	Docker-compatible
Repositories	Docker registry	None	Docker registry
Devices support	Through plugins	Automatic	Automatic
Resources	Native, cgroup	Automatic	Automatic
Batch System	None	Slurm	Slurm
MPI Support	None	Native	Native

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.

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.

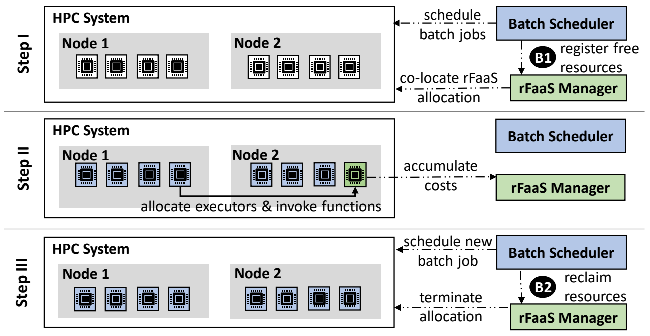


Figure 5: Co-location made easy: *rFaaS* functions running on batch-managed clusters.

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 (B1), which are used it 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 (B2). 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).

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 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. Time T_{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_{local} of tasks such that:

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

Therefore, if the number of tasks is greater than N_{local}, up to N_{remote} tasks can be computed remotely 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.

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 guideline 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 utilizaton?
(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 CPU @ 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 CPU and GPU co-location jobs on the supercomputing system Piz Daint [3]. 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 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

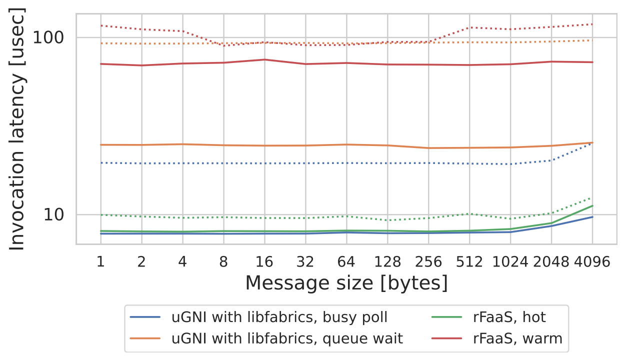


Figure 6: Latency of rFaaS and *libfabric* in different configurations. The straight lines represent the median value, while the dotted line represents the 95th percentile. The plot uses a logarithmic scale.

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.

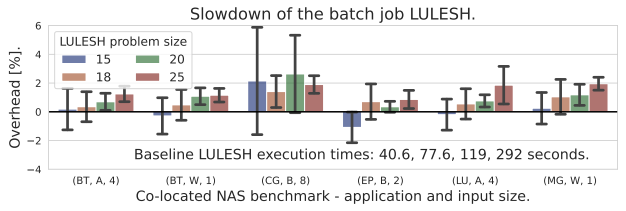
rFaaS provides invocations that are fast enough for the integration of functions into HPC applications.

6.2 Co-location

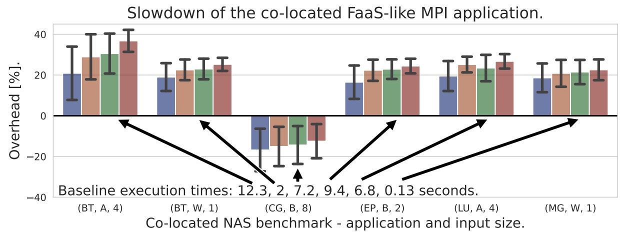
CPU Sharing. To evaluate the overhead of co-locating applications by sharing CPUs, we use the LULESH [43] and MILC [12] applications as a classical batch job, using 64 MPI processes and various problem sizes. We deploy LULESH on 2 Piz Daint nodes, using 32 out of the 36 available cores. It's important 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 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 NAS 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. 7 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 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%.

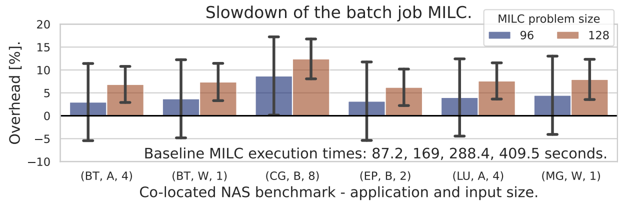
While the performance loss on the function container is higher, it is not a limitation as HPC functions effectively provide users with a way to use resources that would otherwise be wasted: computing a co-located FaaS-like application is free, and the only cost is performance overhead. We also propose that the cost of running co-located *rFaaS* jobs should be lower than that of classical batch



(a) Slowdown of the LULESH batch job.



(b) Slowdown of the FaaS-like MPI application co-located with LULESH.



(c) Slowdown of the MILC batch job.

Figure 7: Overheads of batch jobs co-located with FaaS-like jobs sharing CPUs on idle cores, reported mean with standard deviation over ten repetitions.

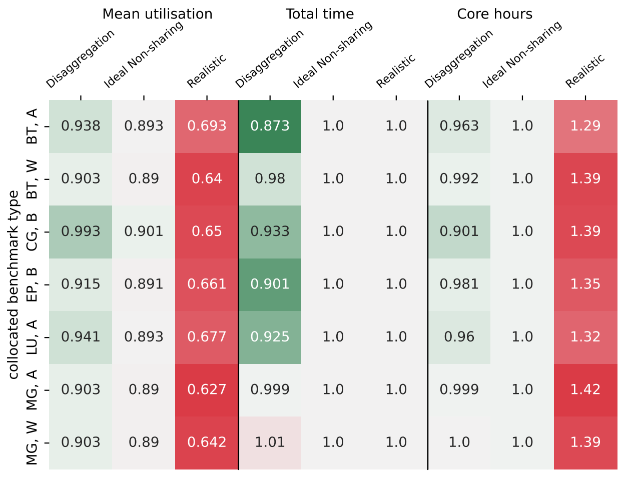
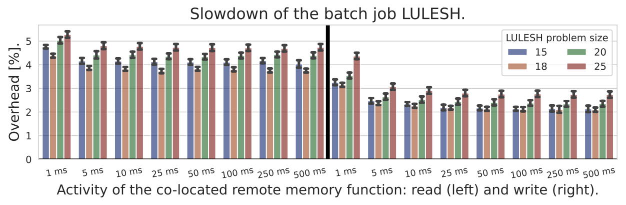
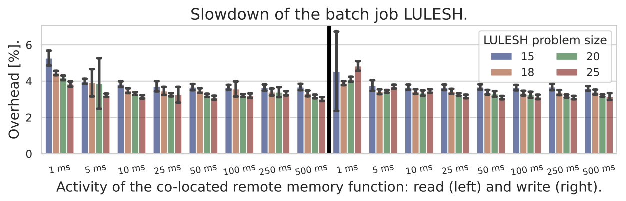


Figure 8: System utilization of co-located execution, a partially co-located execution, and a standard exclusive node allocation.

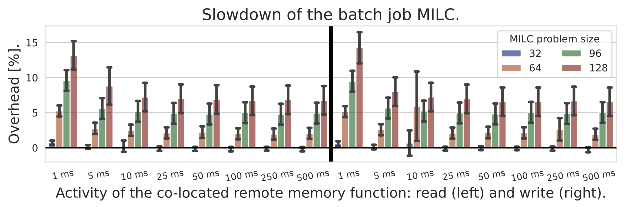
jobs to incentivize reclaiming these resources and to offset the fact that such jobs have lower priority and might be preempted.



(a) LULESH, 27 ranks.



(b) LULESH, 125 ranks.

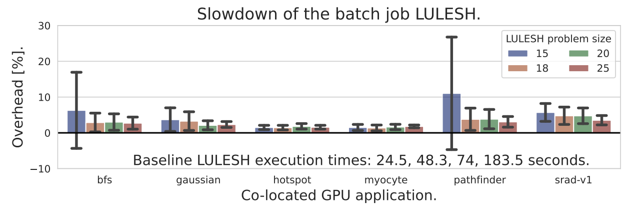


(c) MILC, 32 ranks.

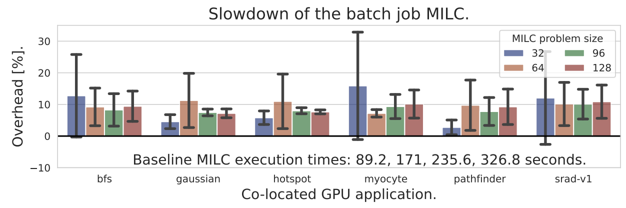
Figure 9: Overhead of batch jobs co-located with *rFaaS* functions providing remote memory. Reported mean with standard deviation over ten repetitions.

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.



(a) Slowdown of the LULESH batch job.



(b) Slowdown of the MILC batch job.

Figure 10: Overheads of batch jobs sharing node with GPU applications. Reported mean with standard deviation.

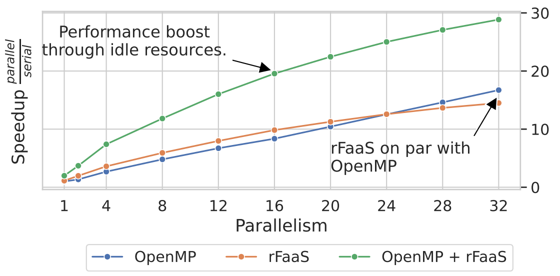
Co-locating batch jobs with *rFaaS* functions and FaaS-like HPC workloads does not introduce significant overheads in batch jobs, regardless of the resource being shared. Allocating only required resources leads to an overall reduction in costs for batch jobs, even taking co-location overheads into account.

6.3 HPC Integration

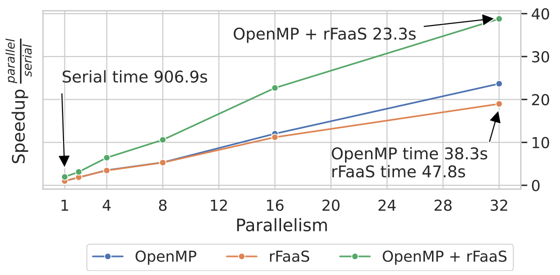
To prove that offloading computations to HPC functions in rFaaS offers performance competitive to that of parallel applications, we integrate HPC functions into OpenMP benchmarks executed on the Piz Daint supercomputer and the Ault cluster. We compare the runtimes of benchmarks using OpenMP parallelism 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.

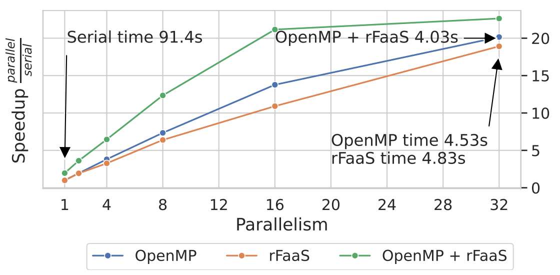
6.3.1 Use-case: Black-Scholes simulation. Figure 11a demonstrates an OpenMP Black-Scholes benchmark from the PARSEC suite modified to use *rFaaS* offloading. The serial execution takes 726 milliseconds on an input of 229 MB. We compare the OpenMP version against complete remote execution with *rFaaS*,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



(a) Black-Scholes method, 100 iterations: *OpenMP*, *rFaaS* and *OpenMP*+rFaaS.



(b) OpenMC, 10,000 particles: *OpenMP*, *rFaaS* and *OpenMP*+rFaaS.



(c) OpenMC, 1,000 particles: *OpenMP*, *rFaaS* and *OpenMP*+rFaaS.

Figure 11: *rFaaS* in practice, reported medians with non-parametric 95% CIs.

until network saturation is reached and that **HPC functions improve multithreaded applications with millisecond-scale runtime**.

6.3.2 Use-case: OpenMC. Figure 11b and Figure 11c demonstrate OpenMC [76], a Monte Carlo particle transport code modified to use *rFaaS* offloading by adding 180 lines of code. We execute the *opr* benchmark [34] modeling an Optimized Power Reactor for the input configurations of simulating 1,000 and 10,000 particles on nodes with two AMD EPYC 7742 64-Core Processor @ 2.25GHz and 256 GB of memory each. Both configurations read an input of 410.8 MB which is loaded from the parallel filesystem by both functions and the client. We compare the OpenMP version against complete remote execution with *rFaaS*, and against doubling parallel resources with cheap serverless allocation. The application demonstrates that **HPC functions can improve real-world HPC applications**.

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 checkpoiting 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, 94, 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

[1] 2020. AWS API Pricing. https://aws.amazon.com/api-gateway/pricing/. Accessed: 2020-08-20.

[2] 2020. Google Cloud Functions Pricing. https://cloud.google.com/functions/pricing. Accessed: 2020-08-20.

[3] 2021. Piz Daint. https://www.cscs.ch/computers/piz-daint/. Accessed: 2020-01-20.

[4] 2021. TOP500, November 2021. https://www.top500.org/lists/top500/2021/11/. Accessed: 2022-03-10.

[5] M. AbdelBaky, M. Parashar, H. Kim, K. E. Jordan, V. Sachdeva, J. Sexton, H. Jamjoom, Z. Shae, G. Pencheva, R. Tavakoli, and M. F. Wheeler. 2012. Enabling High-Performance Computing as a Service. *Computer* 45, 10 (2012), 72–80. https://doi.org/10.1109/MC.2012.293

[6] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[7] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2017. Remote Memory in the Age of Fast Networks. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 121–127. https://doi.org/10.1145/3127479.3131612

[8] C.D. Antonopoulos, D.S. Nikolopoulos, and T.S. Papatheodorou. 2003. Scheduling algorithms with bus bandwidth considerations for SMPs. In *2003 International Conference on Parallel Processing, 2003. Proceedings.* 547–554. https://doi.org/10.1109/ICPP.2003.1240622

[9] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. 2018. The datacenter as a computer: Designing warehouse-scale machines. *Synthesis Lectures on Computer Architecture* 13, 3 (2018), i–189.

[10] Babak Behzad, Surendra Byna, Prabhat, and Marc Snir. 2019. Optimizing I/O Performance of HPC Applications with Autotuning. *ACM Trans. Parallel Comput.* 5, 4, Article 15 (mar 2019), 27 pages. https://doi.org/10.1145/3309205

[11] Lucas Benedicic, Felipe A Cruz, Alberto Madonna, and Kean Mariotti. 2019. Sarus: Highly scalable Docker containers for HPC systems. In *International Conference on High Performance Computing*. Springer, 46–60.

[12] Claude Bernard, Michael C Ogilvie, Thomas A DeGrand, Carleton E DeTar, Steven A Gottlieb, A Krasnitz, Robert L Sugar, and Doug Toussaint. 1991. Studying quarks and gluons on MIMD parallel computers. *The International Journal of Supercomputing Applications* 5, 4 (1991), 61–70.

[13] Abhinav Bhatele, Kathryn Mohror, Steven H. Langer, and Katherine E. Isaacs. 2013. There goes the neighborhood: Performance degradation due to nearby jobs. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* 1–12. https://doi.org/10.1145/2503210.2503247

[14] Alex D. Breslow, Leo Porter, Ananta Tiwari, Michael Laurenzano, Laura Carrington, Dean M. Tullsen, and Allan E. Snavely. 2016. The case for colocation of high performance computing workloads. *Concurrency and Computation: Practice and Experience* 28, 2 (2016), 232–251. https://doi.org/10.1002/cpe.3187 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.3187

[15] Alex D. Breslow, Ananta Tiwari, Martin Schulz, Laura Carrington, Lingjia Tang, and Jason Mars. 2013. Enabling fair pricing on HPC systems with node sharing. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* 1–12. https://doi.org/10.1145/2503210.2503256

[16] Kevin Brown and Satoshi Matsuoka. 2017. Co-locating Graph Analytics and HPC Applications. In *2017 IEEE International Conference on Cluster Computing (CLUSTER).* 659–660. https://doi.org/10.1109/CLUSTER.2017.111

[17] Alexandru Calotoiu, Torsten Hoefler, Marius Poke, and Felix Wolf. 2013. Using Automated Performance Modeling to Find Scalability Bugs in Complex Codes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '13)*. Association for Computing Machinery, New York, NY, USA, Article 45, 12 pages. https://doi.org/10.1145/2503210.2503277

[18] Mohak Chadha, Jophin John, and Michael Gerndt. 2020. Extending SLURM for Dynamic Resource-Aware Adaptive Batch Scheduling. In *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC).* 223–232. https://doi.org/10.1109/HiPC50609.2020.00036

[19] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC).* 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[20] Isaías Comprés, Ao Mo-Hellenbrand, Michael Gerndt, and Hans-Joachim Bungartz. 2016. Infrastructure and API Extensions for Elastic Execution of MPI Applications. In *Proceedings of the 23rd European MPI Users' Group Meeting* (Edinburgh, United Kingdom) *(EuroMPI 2016)*. Association for Computing Machinery, New York, NY, USA, 82–97. https://doi.org/10.1145/2966884.2966917

[21] Marcin Copik, Alexandru Calotoiu, Tobias Grosser, Nicolas Wicki, Felix Wolf, and Torsten Hoefler. 2021. Extracting Clean Performance Models from Tainted Programs. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 403–417. https://doi.org/10.1145/3437801.3441613

[22] Marcin Copik, Tobias Grosser, Torsten Hoefler, Paolo Bientinesi, and Benjamin Berkels. 2020. Work-stealing prefix scan: Addressing load imbalance in large-scale image registration. arXiv:2010.12478 [cs.DC]

[23] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. In *Proceedings of the 22nd International Middleware Conference (Middleware '21)*. Association for Computing Machinery. https://doi.org/10.1145/3464298.3476133

[24] Marcin Copik, Konstantin Taranov, Alexandru Calotoiu, and Torsten Hoefler. 2023. rFaaS: RDMA-Enabled FaaS Platform for Serverless High-Performance Computing. arXiv:2106.13859 [cs.DC]

[25] Iván Cores, Patricia González, Emmanuel Jeannot, María J. Martín, and Gabriel Rodríguez. 2017. An Application-Level Solution for the Dynamic Reconfiguration of MPI Applications. In *High Performance Computing for Computational Science – VECPAR 2016*, Inês Dutra, Rui Camacho, Jorge Barbosa, and Osni Marques (Eds.). Springer International Publishing, Cham, 191–205.

[26] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. 1993. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California, USA) *(PPOPP '93)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/155332.155333

[27] Daniele De Sensi, Tiziano De Matteis, Konstantin Taranov, Salvatore Di Girolamo, Tobias Rahn, and Torsten Hoefler. 2022. Noise in the Clouds: Influence of Network Performance Variability on Application Scalability. *Proc. ACM Meas. Anal.*

Comput. Syst. 6, 3, Article 49 (dec 2022), 27 pages. https://doi.org/10.1145/3570609

[28] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 401–414. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi{ć}

[29] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 467–481. https://doi.org/10.1145/3373376.3378512

[30] José Duato, Antonio J Pena, Federico Silla, Rafael Mayo, and Enrique S Quintana-Ortí. 2010. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *2010 International Conference on High Performance Computing & Simulation*. IEEE, 224–231.

[31] Tyler Dwyer, Alexandra Fedorova, Sergey Blagodurov, Mark Roth, Fabien Gaud, and Jian Pei. 2012. A Practical Method for Estimating Performance Degradation on Multicore Processors, and Its Application to HPC Workloads. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '12)*. IEEE Computer Society Press, Washington, DC, USA, Article 83, 11 pages.

[32] Kaoutar El Maghraoui, Travis J. Desell, Boleslaw K. Szymanski, and Carlos A. Varela. 2007. Dynamic Malleability in Iterative MPI Applications. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*. 591–598. https://doi.org/10.1109/CCGRID.2007.45

[33] Kaoutar El Maghraoui, Boleslaw K. Szymanski, and Carlos Varela. 2006. An Architecture for Reconfigurable Iterative MPI Applications in Dynamic Environments. In *Parallel Processing and Applied Mathematics*, Roman Wyrzykowski, Jack Dongarra, Norbert Meyer, and Jerzy Waśniewski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 258–271.

[34] Lee M.J. et al. 2012. Monte Carlo Reactor Calculation with Substantially Reduced Number of Cycles. In *Proceedings of PHYSOR 2012* (Knoxville, TN).

[35] Dror G. Feitelson and Larry Rudolph. 1996. Toward convergence in job schedulers for parallel supercomputers. In *Job Scheduling Strategies for Parallel Processing*, Dror G. Feitelson and Larry Rudolph (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–26.

[36] Alvaro Frank, Tim Süß, and André Brinkmann. 2019. Effects and benefits of node sharing strategies in HPC batch systems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 43–53.

[37] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 249–264. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gao

[38] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 649–667. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu

[39] Alexander Heinecke, Stefanie Schraufstetter, and Hans-Joachim Bungartz. 2012. A Highly Parallel Black–Scholes Solver Based on Adaptive Sparse Grids. *Int. J. Comput. Math.* 89, 9 (June 2012), 1212–1238. https://doi.org/10.1080/00207160.2012.690865

[40] Torsten Hoefler, Torsten Mehlan, Frank Mietke, and Wolfgang Rehm. 2006. LogfP - A Model for small Messages in InfiniBand. In *Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS), PMEO-PDS'06 Workshop* (Rhodes, Greece).

[41] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2009. The impact of network noise at large-scale communication performance. In *2009 IEEE International Symposium on Parallel & Distributed Processing*. 1–8. https://doi.org/10.1109/IPDPS.2009.5161095

[42] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2010. Characterizing the influence of system noise on large-scale applications by simulation. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.

[43] R. D. Hornung, J. A. Keasler, and M. B. Gokhale. 2011. Hydrodynamics challenge problem. (6 2011). https://doi.org/10.2172/1117905

[44] Chao Huang, Gengbin Zheng, Laxmikant Kalé, and Sameer Kumar. 2006. Performance Evaluation of Adaptive MPI. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, New York, USA) *(PPoPP '06)*. Association for Computing Machinery, New York, NY, USA, 12–21. https://doi.org/10.1145/1122971.1122976

[45] Costin Iancu, Steven Hofmeyr, Filip Blagojević, and Yili Zheng. 2010. Oversubscription on multicore processors. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 1–11. https://doi.org/10.1109/IPDPS.2010.5470434

[46] S. Iserte, R. Mayo, E. S. Quintana-Ortí, V. Beltran, and A. J. Peña. 2017. Efficient Scalable Computing through Flexible Applications and Adaptive Workloads. In *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*. 180–189. https://doi.org/10.1109/ICPPW.2017.36

[47] Eric Jonas, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. *CoRR* abs/1702.04024 (2017). arXiv:1702.04024 http://arxiv.org/abs/1702.04024

[48] James Patton Jones and Bill Nitzberg. 1999. Scheduling for Parallel Supercomputing: A Historical Perspective of Achievable Utilization. In *Job Scheduling Strategies for Parallel Processing*, Dror G. Feitelson and Larry Rudolph (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–16.

[49] Matthew D. Jones, Joseph P. White, Martins Innus, Robert L. DeLeon, Nikolay Simakov, Jeffrey T. Palmer, Steven M. Gallo, Thomas R. Furlani, Michael T. Showerman, Robert Brunner, Andry Kot, Gregory H. Bauer, Brett M. Bode, Jeremy Enos, and William T. Kramer. 2017. Workload Analysis of Blue Waters. *CoRR* abs/1703.00924 (2017). arXiv:1703.00924 http://arxiv.org/abs/1703.00924

[50] Melanie Kambadur, Tipp Moseley, Rick Hank, and Martha A. Kim. 2012. Measuring interference between live datacenter applications. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1109/SC.2012.78

[51] Awais Khan, Hyogi Sim, Sudharshan S. Vazhkudai, Ali R. Butt, and Youngjae Kim. 2021. An Analysis of System Balance and Architectural Trends Based on Top500 Supercomputers. In *The International Conference on High Performance Computing in Asia-Pacific Region* (Virtual Event, Republic of Korea) *(HPC Asia 2021)*. Association for Computing Machinery, New York, NY, USA, 11–22. https://doi.org/10.1145/3432261.3432263

[52] Matthew J. Koop, Miao Luo, and Dhabaleswar K. Panda. 2009. Reducing network contention with mixed workloads on modern multicore, clusters. In *2009 IEEE International Conference on Cluster Computing and Workshops*. 1–10. https://doi.org/10.1109/CLUSTR.2009.5289162

[53] E. Koukis and N. Koziris. 2006. Memory and network bandwidth aware scheduling of multiprogrammed workloads on clusters of SMPs. In *12th International Conference on Parallel and Distributed Systems - (ICPADS'06)*, Vol. 1. 10 pp.–. https://doi.org/10.1109/ICPADS.2006.59

[54] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. 2017. Singularity: Scientific containers for mobility of compute. *PLOS ONE* 12, 5 (May 2017), e0177459. https://doi.org/10.1371/journal.pone.0177459

[55] Shuang Liang, Ranjit Noronha, and Dhabaleswar K. Panda. 2005. Swapping to Remote Memory over InfiniBand: An Approach using a High Performance Network Block Device. In *2005 IEEE International Conference on Cluster Computing*. 1–10. https://doi.org/10.1109/CLUSTR.2005.347050

[56] Jochen Liedtke, Marcus Völp, and Kevin Elphinstone. 2000. Preliminary Thoughts on Memory-Bus Scheduling. In *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System* (Kolding, Denmark) *(EW 9)*. Association for Computing Machinery, New York, NY, USA, 207–210. https://doi.org/10.1145/566726.566768

[57] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (Austin, TX, USA) *(ISCA '09)*. Association for Computing Machinery, New York, NY, USA, 267–278. https://doi.org/10.1145/1555754.1555789

[58] Alberto Madonna and Tomas Aliaga. 2022. Libfa bric-based Injection Solutions for Portable Containerized MPI Applications. In *2022 IEEE/ACM 4th International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. 45–56. https://doi.org/10.1109/CANOPIE-HPC56864.2022.00010

[59] Johannes Manner, Martin Endreß, Tobias Heckel, and Guido Wirtz. 2018. Cold start influencing factors in function as a service. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, 181–188.

[60] Gonzalo Martín, David E. Singh, Maria-Cristina Marinescu, and Jesús Carretero. 2015. Enhancing the performance of malleable MPI applications by using performance-aware dynamic reconfiguration. *Parallel Comput.* 46 (2015), 60–77. https://doi.org/10.1016/j.parco.2015.04.003

[61] George Michelogiannakis, Benjamin Klenk, Brandon Cook, Min Yee Teh, Madeleine Glick, Larry Dennison, Keren Bergman, and John Shalf. 2022. A Case For Intra-Rack Resource Disaggregation in HPC. *ACM Trans. Archit. Code Optim.* (jan 2022). https://doi.org/10.1145/3514245 Just Accepted.

[62] Ao Mo-Hellenbrand, Isaías Comprés, Oliver Meister, Hans-Joachim Bungartz, Michael Gerndt, and Michael Bader. 2017. A Large-Scale Malleable Tsunami Simulation Realized on an Elastic MPI Infrastructure. In *Proceedings of the Computing Frontiers Conference* (Siena, Italy) *(CF'17)*. Association for Computing Machinery, New York, NY, USA, 271–274. https://doi.org/10.1145/3075564.3075585

[63] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 57–70. https://www.usenix.org/conference/atc18/presentation/oakes

[64] Gagandeep Panwar, Da Zhang, Yihan Pang, Mai Dahshan, Nathan DeBardeleben, Binoy Ravindran, and Xun Jian. 2019. Quantifying Memory Underutilization in HPC Systems and Using It to Improve Performance via Architecture Support. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 821–835. https://doi.org/10.1145/3352460.3358267

[65] Gagandeep Panwar, Da Zhang, Yihan Pang, Mai Dahshan, Nathan DeBardeleben, Binoy Ravindran, and Xun Jian. 2019. Quantifying memory underutilization in hpc systems and using it to improve performance via architecture support. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 821–835.

[66] Jinsu Park, Seongbeom Park, and Woongki Baek. 2019. CoPart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[67] Gerard París, Pedro García-López, and Marc Sánchez-Artigas. 2020. Serverless Elastic Exploration of Unbalanced Algorithms. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. 149–157. https://doi.org/10.1109/CLOUD49709.2020.00033

[68] Tirthak Patel, Zhengchun Liu, Raj Kettimuthu, Paul Rich, William Allcock, and Devesh Tiwari. 2020. Job Characteristics on Large-Scale Systems: Long-Term Analysis, Quantification, and Implications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 84, 17 pages.

[69] Ivy Peng, Roger Pearce, and Maya Gokhale. 2020. On the Memory Underutilization: Exploring Disaggregated Memory on HPC Systems. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 183–190. https://doi.org/10.1109/SBAC-PAD49847.2020.00034

[70] Christian Pinto, Dimitris Syrivelis, Michele Gazzetti, Panos Koutsovasilis, Andrea Reale, Kostas Katrinis, and H. Peter Hofstee. 2020. ThymesisFlow: A Software-Defined, HW/SW co-Designed Interconnect Stack for Rack-Scale Memory Disaggregation. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 868–880. https://doi.org/10.1109/MICRO50266.2020.00075

[71] Suraj Prabhakaran, Mohsin Iqbal, Sebastian Rinke, Christian Windisch, and Felix Wolf. 2014. A Batch System with Fair Scheduling for Evolving Applications. In *2014 43rd International Conference on Parallel Processing*. 351–360. https://doi.org/10.1109/ICPP.2014.44

[72] Suraj Prabhakaran, Marcel Neumann, Sebastian Rinke, Felix Wolf, Abhishek Gupta, and Laxmikant V. Kale. 2015. A Batch System with Efficient Adaptive Scheduling for Malleable and Evolving Applications. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 429–438. https://doi.org/10.1109/IPDPS.2015.34

[73] Howard Pritchard, Evan Harvey, Sung-Eun Choi, James Swaro, and Zachary Tiffany. 2016. The GNI provider layer for OFI libfabric. In *Proceedings of Cray User Group Meeting, CUG*, Vol. 2016.

[74] Pramod Subba Rao and George Porter. 2016. Is memory disaggregation feasible? A case study with Spark SQL. In *2016 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 75–80. https://doi.org/10.1145/2881025.2881030

[75] A. Raveendran, T. Bicer, and G. Agrawal. 2011. A Framework for Elastic Execution of Existing MPI Programs. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. 940–947. https://doi.org/10.1109/IPDPS.2011.240

[76] Paul K. Romano, Nicholas E. Horelik, Bryan R. Herman, Adam G. Nelson, Benoit Forget, and Kord Smith. 2015. OpenMC: A State-of-the-Art Monte Carlo Code for Research and Development. *Ann. Nucl. Energy* 82 (2015), 90–97.

[77] Aamer Shah, Matthias Müller, and Felix Wolf. 2018. Estimating the Impact of External Interference on Application Performance. In *Euro-Par 2018: Parallel Processing*, Marco Aldinucci, Luca Padovani, and Massimo Torquati (Eds.). Springer International Publishing, Cham, 46–58.

[78] James Shimek, James Swaro, and M Saint Paul. 2016. Dynamic rdma credentials. In *Cray User Group (CUG) Meeting*.

[79] Sergei Shudler, Alexandru Calotoiu, Torsten Hoefler, and Felix Wolf. 2017. Isoefficiency in Practice: Configuring and Understanding the Performance of Task-Based Applications. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Austin, Texas, USA) *(PPoPP '17)*. Association for Computing Machinery, New York, NY, USA, 131–143. https://doi.org/10.1145/3018743.3018770

[80] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. 2020. Prebaking Functions to Warm the Serverless Cold Start. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3423211.3425682

[81] Nikolay A. Simakov, Robert L. DeLeon, Joseph P. White, Thomas R. Furlani, Martins Innus, Steven M. Gallo, Matthew D. Jones, Abani Patra, Benjamin D. Plessinger, Jeanette Sperhac, Thomas Yearke, Ryan Rathsam, and Jeffrey T. Palmer. 2016. A Quantitative Analysis of Node Sharing on HPC Clusters Using XDMoD Application Kernels. In *Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale* (Miami, USA) *(XSEDE16)*. Association for Computing Machinery, New York, NY, USA, Article 32, 8 pages. https://doi.org/10.1145/2949550.2949553

[82] Allan Snavely, Dean M. Tullsen, and Geoff Voelker. 2002. Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (Marina Del Rey, California) *(SIGMETRICS '02)*. Association for Computing Machinery, New York, NY, USA, 66–76. https://doi.org/10.1145/511334.511343

[83] Lingjia Tang, Jason Mars, Wei Wang, Tanima Dey, and Mary Lou Soffa. 2013. ReQoS: Reactive Static/Dynamic Compilation for QoS in Warehouse Scale Computers. *SIGPLAN Not.* 48, 4 (mar 2013), 89–100. https://doi.org/10.1145/2499368.2451126

[84] Xiongchao Tang, Haojie Wang, Xiaosong Ma, Nosayba El-Sayed, Jidong Zhai, Wenguang Chen, and Ashraf Aboulnaga. 2019. Spread-n-share: improving application performance and cluster throughput with resource-aware job placement. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[85] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2021. Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 495–514. https://www.usenix.org/conference/osdi21/presentation/thorpe

[86] Lukas Tobler. 2022. GPUless – Serverless GPU Functions. (2022). https://spcl.inf.ethz.ch/Publications/.pdf/tobler-gpu-thesis.pdf

[87] Feiyi Wang, Sarp Oral, Satyabrata Sen, and Neena Imam. 2019. Learning from Five-year Resource-Utilization Data of Titan System. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. 1–6. https://doi.org/10.1109/CLUSTER.2019.8891001

[88] Jonathan Weinberg and Allan Snavely. 2006. User-Guided Symbiotic Space-Sharing of Real Workloads. In *Proceedings of the 20th Annual International Conference on Supercomputing* (Cairns, Queensland, Australia) *(ICS '06)*. Association for Computing Machinery, New York, NY, USA, 345–352. https://doi.org/10.1145/1183401.1183450

[89] Jonathan Weinberg and Allan Snavely. 2007. Symbiotic Space-Sharing on SDSC's DataStar System. In *Job Scheduling Strategies for Parallel Processing*, Eitan Frachtenberg and Uwe Schwiegelshohn (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 192–209.

[90] Joseph P. White, Robert L. DeLeon, Thomas R. Furlani, Steven M. Gallo, Matthew D. Jones, Amin Ghadersohi, Cynthia D. Cornelius, Abani K. Patra, James C. Browne, William L. Barth, and John Hammond. 2014. An Analysis of Node Sharing on HPC Clusters Using XDMoD/TACC_Stats. In *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment* (Atlanta, GA, USA) *(XSEDE '14)*. Association for Computing Machinery, New York, NY, USA, Article 31, 8 pages. https://doi.org/10.1145/2616498.2616533

[91] Yaocheng Xiang, Chencheng Ye, Xiaolin Wang, Yingwei Luo, and Zhenlin Wang. 2019. EMBA: Efficient Memory Bandwidth Allocation to Improve Performance on Intel Commodity Processor. In *Proceedings of the 48th International Conference on Parallel Processing* (Kyoto, Japan) *(ICPP 2019)*. Association for Computing Machinery, New York, NY, USA, Article 16, 12 pages. https://doi.org/10.1145/3337821.3337863

[92] Qingqing Xiong, Emre Ates, Martin C. Herbordt, and Ayse K. Coskun. 2018. Tangram: Colocating HPC Applications with Oversubscription. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2018.8547644

[93] Di Xu, Chenggang Wu, and Pen-Chung Yew. 2010. On Mitigating Memory Bandwidth Contention through Bandwidth-Aware Scheduling. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques* (Vienna, Austria) *(PACT '10)*. Association for Computing Machinery, New York, NY, USA, 237–248. https://doi.org/10.1145/1854273.1854306

[94] Hao Xu, Shuang Song, and Ze Mao. 2023. Characterizing the Performance of Emerging Deep Learning, Graph, and High Performance Computing Workloads Under Interference. arXiv:2303.15763 [cs.PF]

[95] Haihang You and Hao Zhang. 2013. Comprehensive Workload Analysis and Modeling of a Petascale Supercomputer. In *Job Scheduling Strategies for Parallel Processing*, Walfredo Cirne, Narayan Desai, Eitan Frachtenberg, and Uwe Schwiegelshohn (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 253–271.

[96] Felippe Vieira Zacarias, Paul Carpenter, and Vinicius Petrucci. 2021. Memory Demands in Disaggregated HPC: How Accurate Do We Need to Be?. In *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 1–6. https://doi.org/10.1109/PMBS54543.2021.00006

[97] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. 2010. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, Pennsylvania, USA) *(ASPLOS XV)*. Association for Computing Machinery, New York, NY, USA, 129–142. https://doi.org/10.1145/1736020.1736036

[98] Darko Zivanovic, Milan Pavlovic, Milan Radulovic, Hyunsung Shin, Jongpil Son, Sally A. Mckee, Paul M. Carpenter, Petar Radojković, and Eduard Ayguadé. 2017. Main Memory in HPC: Do We Need More or Could We Live with Less? *ACM Trans. Archit. Code Optim.* 14, 1, Article 3 (mar 2017), 26 pages. https://doi.org/10.1145/3023362

[99] David Álvarez, Kevin Sala, and Vicenç Beltran. 2022. nOS-V: Co-Executing HPC Applications Using System-Wide Task Scheduling. https://doi.org/10.48550/ARXIV.2204.10768