Malleability in Modern HPC Systems: Current Experiences, Challenges, and Future Opportunities

Ahmad Tarraf, Martin Schreiber, Alberto Cascajo, Jean-Baptiste Besnard, Marc-André Vef, Dominik Huber, Sonja Happ, André Brinkmann, David E. Singh, Hans-Christian Hoppe, Alberto Miranda, Antonio J. Peña, Rui Machado, Marta Garcia-Gasulla, Martin Schulz, Paul Carpenter, Simon Pickartz, Tiberiu Rotaru, Sergio Iserte, Victor Lopez, Jorge Ejarque, Heena Sirwani, Jesus Carretero, and Felix Wolf

***Abstract*—With the increase of complex scientific simulations driven by workflows and heterogeneous workload profiles, managing system resources effectively is essential for improving performance and system throughput, especially due to trends like heterogeneous HPC and deeply integrated systems with on-chip accelerators. For optimal resource utilization, dynamic resource allocation can improve productivity across all system and application levels, by adapting the applications' configurations to the system's resources. In this context, malleable jobs, which can change resources at runtime, can increase the system throughput and resource utilization while bringing various advantages for HPC users (e.g., shorter waiting time). Malleability has received much attention recently, even though it has been an active research area for more than two decades. This article presents the state-of-the-art of malleable implementations in HPC systems, targeting mainly malleability in compute and I/O resources. Based on our experiences, we state our current concerns and list future opportunities for research.**

***Index Terms*—Malleability, state-of-the-art, survey, HPC.**

Manuscript received 13 September 2023; revised 7 May 2024; accepted 18 May 2024. Date of publication 29 May 2024; date of current version 18 July 2024. This work was supported by the European Commission, the German Federal Ministry of Education and Research (BMBF), and the Research and Innovation Spanish Ministry (PCI2021-121958 and "NextGeneration EU/PRTR") under the EuroHPC Programmes DEEP-SEA (GA No. 955606 – 16HPC011K – MCIN/AEI/10.13039/501100011033), ADMIRE (GA No. 956748 – 16HPC006K – PCI2021-121966) Time-X (GA No. 955701 – 16HPC46K), REGALE (GA No. 956560 – 16HPC039K), which receive support from the European Union's Horizon 2020 programme and DE, FR, ES, GR, BE, SE, GB, CH (DEEP-SEA); DE, FR, ES, IT, PL, SE (ADMIRE); BE, FR, DE, CH (Time-X); and TR, NO, PT, DE (REGALE). As part of the NHR funding, we further acknowledge the support from BMBF and HMWK. Recommended for acceptance by A. Sussman. *(Corresponding author: Ahmad Tarraf.)*

Ahmad Tarraf and Felix Wolf are with the Department of Computer Science, Technical University of Darmstadt, 64289 Darmstadt, Germany (e-mail: ahmad.tarraf@tu-darmstadt.de).

Martin Schreiber is with the IMAG, Université Grenoble Alpes, 38400 Saint-Martin-d'Hères, France, and also with the Department of Informatics, Technical University of Munich, 80333 München, Germany.

Alberto Cascajo, David E. Singh, and Jesus Carretero are with the Computer Science and Engineering Department, University Carlos III of Madrid, 28911 Leganés, Madrid, Spain.

Jean-Baptiste Besnard is with the France Office, ParaTools SAS, 91680 Bruyères-le-Châtel, France.

Marc-André Vef and André Brinkmann are with the Institute of Computer Science, Johannes Gutenberg University Mainz, 55122 Mainz, Germany.

Dominik Huber and Martin Schulz are with the Department of Informatics, Technical University of Munich, 80333 München, Germany.

Sonja Happ and Simon Pickartz are with the ParTec AG, 81679 Munich, Germany.

Hans-Christian Hoppe is with the Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, 52428 Jülich, Germany.

Alberto Miranda, Antonio J. Peña, Marta Garcia-Gasulla, Paul Carpenter, Sergio Iserte, Victor Lopez, and Jorge Ejarque are with Barcelona Supercomputing Center, 08034 Barcelona, Spain.

Rui Machado, Tiberiu Rotaru, and Heena Sirwani are with the High Performance Computing Department, Fraunhofer ITWM, 67663 Kaiserslautern, Germany.

Digital Object Identifier 10.1109/TPDS.2024.3406764

I. Introduction

MALLEABILITY in HPC has been a research topic for many years [1], [2], [3], [4], [5], [6], [7], [8], [9]. Malleable jobs can be considered the most scheduler-friendly jobs, as they allow for effective system utilization and throughput. For example, these jobs can start whenever the minimal requirements are available and expand during their lifetime when more resources become available. For a domain partitioning application with adaptive mesh (AMR) [6], for instance, the problem size of the job can increase or decrease during its execution and resources added or released accordingly. Malleable jobs can also increase energy efficiency [10], [11] and reduce average power consumption [12], [13], which makes these jobs *green* due to their lower impact on the environment. By considering platforms with different classes of compute nodes, energy minimization, and performance-per-watt maximization as optimization criteria, finding the optimal solution is not limited to finding the most appropriate number of processes but also to determining the classes of compute nodes that must be used [14], [15]. With the share of greenhouse gas emissions produced by digital technologies increasing from 2.5% in 2013 to 4% in 2020 and likely 8% by 2025 [16], malleability can be a game changer, not only for performance optimization and higher efficiency but also for sustainability and greener IT. It allows HPC centers to increase their overall system throughput, while providing them with more flexibility, e.g., regarding their power consumption. At the same time, malleability also brings benefits for HPC users [17], such as reduced response times [18], which usually lead to lower turnaround times [19].

While there are several reasons why malleability should be used on modern HPC systems, the question remains: Why do

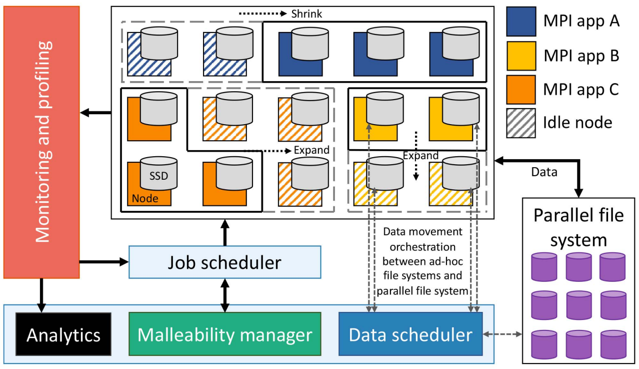

Fig. 1. An example architecture of a malleable HPC system.

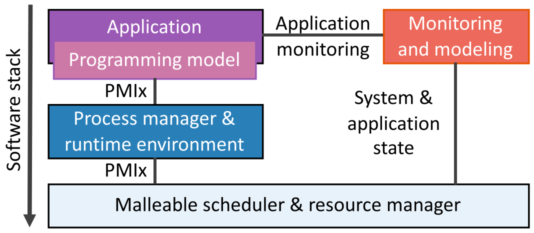
Software stack
Application
Programming model
PMIx
Process manager & runtime environment
PMIx
Malleable scheduler & resource manager
Application monitoring
Monitoring and modeling
System & application state

Fig. 2. A multi-layer model of the system software stack on HPC systems subject to malleability including monitoring.

modern HPC systems lack malleable implementations? Despite the long research history, users do not employ malleability tools and job schedulers in production HPC machines do not exploit malleability [20]. In this survey, we report the current efforts and challenges of malleability in HPC based on our experience across several distinct European projects.

In general, introducing malleability to HPC systems is challenging, as it requires changes in almost the entire HPC software stack (see Fig. 1). Systematically approaching this challenge could facilitate the development of standardized interfaces for malleability among the involved components. One way is organizing malleability into multiple layers, defining tasks and services for each layer of the model, and developing well-defined interfaces between these layers (e.g., based on the PMIx standard [21]). Fig. 2 provides an example of such a multi-layer model.

In accordance with Fig. 2, we structure the first part of this paper targeting computational malleability. As a starting point, we introduce the terminology in Section II. Section III covers malleability from the application perspective and briefly looks into monitoring and modeling malleable applications. Section IV examines existing programming models and their current developments, progress in process and resource managers, and the interaction between these layers. Compared to a recent survey [22], which examines several MPI process malleability solutions and shows possible adaptations in user codes, in this paper, we explore malleability system-wide, handling current efforts and planned future extensions.

The second part of this paper (Section V) examines I/O malleability, focusing on I/O services and storage resources. I/O malleability offers similar benefits as computational malleability and can, if combined, complement each other, significantly improving their efficiency. Moreover, both methods share most of the necessary components (see Fig. 1). Note that other malleable resources, e.g., by directly targeting network communication or main memory, could also be beneficial but are out of the scope of this paper. Finally, based on our experience, we state the major challenges related to malleability in Section VI and guide future research in Section VII.

II. Terminology and Classification

In HPC, malleability refers to the ability of a system or application to dynamically adapt its *resource usage* on demand without necessarily stopping or restarting its activity. More specifically, malleability involves the allocation of resources (e.g., CPU, memory, network, and I/O bandwidth) in response to their availability and changes in the workload. A term often confused with malleability is dynamism, which refers to the ability of a system or application to dynamically adapt its *behavior or performance* to deal with changes in workload, availability of new resources, and the conditions of the system. A dynamic workload manager can adapt the resource allocation for different jobs based on their resource demands and the system's current status. Hence, dynamism and malleability are related concepts in HPC. Malleability can be considered a subcategory of dynamism, as it allows jobs to adapt themselves to changing conditions at runtime. In contrast, dynamism is more general, including more sources of information and mechanisms in the decision-making. The terms have been widely used over the years to describe not only jobs but also systems and resources that support such operations. Considering, e.g., power as a malleable resource, jobs subject to power capping can be considered malleable. Thus, we clarify the existing terminology in this section.

A. Job Classification

In HPC, dynamism is essential for optimizing resource utilization and improving system efficiency. However, not all HPC systems support this aspect. For example, a system that implements a fixed resource allocation for each job is unable to reconfigure the job even when new resources are available. Thus, jobs that leverage dynamism make sense only if the system supports this feature, as the system needs to accommodate job reconfigurations at runtime.

HPC jobs can be categorized as rigid, moldable, malleable, or evolving [23]. *Rigid* jobs are the most basic and common ones. They have a fixed resource allocation, which remains constant during their life cycle. There are many reasons why a job is rigid, including: 1) the nature of the problem (e. g., fixed domain decomposition), 2) the structure of the algorithm (e.g., assigning fixed tasks to nodes), or 3) just for simplicity during development. The expected execution time is usually provided to the batch system when submitting the job. *Moldable* jobs can be executed with various resource allocations. The user specifies several acceptable resource configurations besides individual and tentative execution times for each of them. Moldable jobs, by themselves, do not allow the system to change the allocations for the jobs; they require a moldable system. If the resource manager and the batch system support these features, they become more flexible –and complex–, improving resource utilization compared to rigid platforms.

Malleable and *evolving* jobs can adapt their resource usage during runtime. The difference lies in how and when the resource reconfiguration decision is executed. A job is *evolving* if the *job* decides when and how the resources are reconfigured. Contrary, a job is *malleable* if the *system* makes this decision and the job just applies it. More specifically, evolving jobs investigate their resource usage at specific points followed by deciding the resource allocations. In contrast, for malleable jobs, the resource manager or the system scheduler requests a change in the resource allocation depending on the system status, and the jobs react by reconfiguring themselves. As application resource requests can be forwarded to the system to be handled by the platform, for simplicity, we assume evolving jobs are covered by the same topics as malleable jobs.

As the resources for rigid and moldable jobs are allocated before the application execution, the allocation is referred to as *static allocation*. In contrast, *dynamic allocation* refers to expanding or shrinking resources for evolving and malleable jobs. This paper focuses on topics related to malleable jobs.

B. Malleable Systems and Resources

As *dynamism* in HPC environments is a system's ability to adapt to varying requirements (workloads, resources, etc.), the environment should support some key features: Dynamic resource allocation, load balancing, malleability, fault tolerance, and heterogeneity. *Dynamic resource allocation* allows the system to adapt the resources used by jobs at runtime. The system typically considers the workload and available resources to perform these allocations. Supporting this feature provides additional mechanisms for effective resource usage, increasing the system's performance. *Load balancing* is the system's ability to balance the workload across multiple nodes and processors. As the system can adapt the allocated resources, it should ensure that each job, node, and processor operates at acceptable performance levels. To provide dynamic resource allocation and allow jobs to use these resources at runtime, the system should support malleability. Dynamic systems can scale the malleable jobs up and down, enabling them to react to peak loads and low activity periods of resource usage.

Dynamic systems should be designed to tolerate failures, i.e., continue operating even when a job or component fails. However, *fault tolerance* accounts for new potential sources of failures, e.g., resource re-allocation and job reconfiguration. If the first fails, the system might be left in an unstable state, while if the latter fails, jobs might yield erroneous results, stall, or end abruptly. Finally, *heterogeneity* allows the system to support a wide range of hardware and software. It optimizes different types of workloads and applications, depending on their requirements. However, specific hardware and software might reduce the interoperability between the system and the jobs, forcing developers to adapt their applications and components to the platform requirements.

C. Proactive and Reactive Malleability

Based on the decision-making mechanism in dynamic systems, there are two approaches to deal with resource allocation and job scheduling. A *proactive system* can predict and prevent events before they occur (e.g., potential problems, reconfigurations, interference, and performance degradation). It might involve predictive analytics and workload modeling to anticipate changes in demand and automatically adjust the available resources to match the changing workload. These systems can maintain their performance and reliability based on monitoring techniques, prediction, continuous analysis, and re-schedulings, however, this increases their complexity. In contrast, a *reactive system* responds quickly to events after they occur (e.g., when a job requests more resources, failures, or network congestion). The system detects and evaluates the current status of the resources and jobs, and takes a decision based on the implemented policy (e.g., reduce energy consumption or increase throughput). This approach requires real-time monitoring and alerts to detect the events and take appropriate actions. While both systems have cons and pros, selecting one depends on various factors, such as specific workload requirements, available resources, and performance and reliability goals of the system. Yet, systems that combine both approaches may achieve the best balance of performance, efficiency, and resilience. Note, we distinguish between *active* malleability, where an application investigates its use of resources at specific points in its code and then decides on whether and how to change the resource set, and *passive* malleability, where a resource manager/scheduler offers or requests a resource change from an application at an arbitrary point in its execution, to which the application can react.

III. Malleability in HPC Applications

On the application level, malleability brings various advantages to application owners and HPC centers (Section III-A). Moreover, to fully characterize malleable applications, monitoring and modeling tools will need to adapt (Section III-B).

A. Applications

Traditionally and as a result of the pre-dominant MPI programming model, HPC applications work on a fixed resource set specified by the user. Overly large resource/time allocations can reduce the efficiency of the system usage. Smaller allocations increase execution times and can cause premature termination and loss of results due to timeouts. If the applications are still running when the allocation time expires, the system cancels the job, and important information may be lost. Specifying realistic resource requests is important for computer centers and end-users, yet without detailed application knowledge, is difficult to do.

Modern HPC applications exhibit a high degree of dynamism aligning with the behavior of the simulated physical processes. Besides, they can be affected by external constraints such as dynamic system reconfigurations (e.g., power capping) or global resource contention. Consequently, the ability to steer resources allocated to an application would be an advantage for *both* end-users and system operators. At the heart of this are applications capable of making effective use of dynamically changing resources. This can require redistribution of data or tasks to a changed set of resources, which can be costly operations, before continuing computation without losing previous compute results. Monitoring solutions, for example, could be used here to find the right timing for such operations. Applications supporting advanced forms of checkpoint/restart or dynamic load balancing [24], [25] are good examples, being massively parallel and of an iterative nature with prescribed points where resource changes can be accommodated. Also, certain kinds of applications, e.g., those based on task farming or parallel-in-time codes based on the Parareal algorithm [26], can benefit from malleability without significant code changes.

A malleable solution could benefit end-users by: 1) reduced time-to-machine, as well as 2) improved time-to-solution and more efficient use of resources, which 3) preserves the computing time budget. The users only pay for resources their applications can use *efficiently*. Compute centers, on the other hand, could benefit from: 1) reduced resource idling, which directly 2) increases system utilization and efficiency and provides flexibility in 3) controlling their power consumption to stay inside the narrow power corridors they usually have for large-scale systems. Despite these benefits, a recent survey [17] shows that only 16% of the examined applications could change the number of processes dynamically. Interestingly, 37% of the applications can restart on a different number of processes from an application checkpoint, which means that they are at least moldable. This 21% difference could indicate that the maturity of existing mechanisms for checkpoint/restart and the resulting benefits drove significantly more applications to tackle the problem of dynamic data distribution.

Almost twenty years ago, simulations have shown that already a small percentage of malleable jobs can improve system performance [27]. Examples of applications that currently use malleability in large-scale systems are EpiGraph [28], WaComm++ [29], MPDATA [11], HPG-Aligner [30], and LAMMPS [31]. To make an application passively malleable (see Section II-C), for example, the user-level code must outline the reconfiguration phase and carefully define data-management procedures enabling the remapping of the various operands. Nonetheless, more generic support could, e.g., be devised either inside PETSc [25] that controls the data or task distribution of applications or directly through MPI [24], with a set of extensions for dynamic resources and AMR.

Deciding on resource allocations requires a detailed understanding of application behavior and in particular the application phases. Often, the achievable parallelism varies over time according to the transition between phases. Malleability can serve to adapt allocated resources to the current needs of the applications, yet the overhead of reconfiguration must be considered (see Section III-B). In the real world, perfectly scaling applications and systems do not occur, and the data redistribution effort can scale differently from the computation. Capabilities of HPC systems also impact the available options for adaptive resource changes: shared memory across all application processes can greatly simplify data redistribution and reduce its overhead, while the use of shared resources (such as I/O nodes or subsystems) across different jobs can create interference and substantial overhead. Moreover, HPC system resource management must handle the totality of jobs and in the interest of the compute center usually strives to achieve a global optimum (e.g., maximum throughput) rather than minimizing each job's runtime as handled later in Section IV-C. To support malleable decisions application/system monitoring and modeling can be used as described in the next section.

B. Monitoring and Modeling

Malleable applications pose specific challenges in terms of monitoring. First, since these applications can change the number of processes during their lifespan, the traditional approach (i.e., process-based monitoring) is insufficient and additional monitoring capabilities must be devised to track a job's structure. Second, runtime dynamics are of paramount importance, as being dynamic requires diagnosing the need for a resource change during runtime. Thus, traditional feed-forward tools relying on post-mortem analysis must be complemented with runtime monitoring, and monitoring has to become less transitive on HPC applications.

Job malleability can benefit from performance models determining the worthiness of a change. These models are built from historical data, accounting for several job instances, profiles, or more verbose traces. Each program running on a supercomputer has a complex performance space, often described as how it *scales*. However, evolution in HPC architectures made this space even more convoluted. Indeed, what was previously a single number of MPI processes is now a multi-variadic space, ranging from the number of OpenMP threads versus MPI processes and the potential availability of accelerators. A dynamic system changes application parameters to move applications' configurations in the space of their expression on the machine. Performance models could be used to justify malleable decisions. However, these are difficult to obtain in such a complex space. Yet, all performance data sources should be leveraged to produce higher-level performance models.

While the performance analysis of targeted code segments is common practice for the optimization of an individual problem or scale, capturing the parametric unfolding of application behavior is rare. Still, as Fig. 3 shows, developing such dynamic application models is crucial to support adaptivity in the dynamism workflow. It is the foundation to enable the application to run in a more optimal configuration, i.e., according to the available resources, runtime constraints, and possible collocation dependencies between components. Further, it supports reconfiguration during the program execution, reacting on the fly to new restrictions, such as *urgent* jobs, quality of service (QoS), or maintenance constraints.

Dynamism is a step towards more efficient computing systems thanks to a feedback loop between execution configurations and their resulting performance. Considering the increasing job complexity going from monolithic jobs to workflows, in-situ or even ad-hoc services, there is a need to express launch configurations better. Moreover, what is already difficult on the job

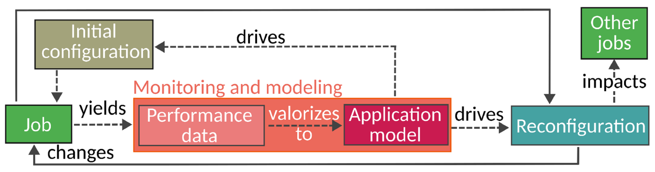

Fig. 3. Dynamism workflow for improved job configuration.

level is even harder at the machine-wide level, requiring reflexive cross-job modeling capabilities. Thus, a malleable system must include a comprehensive monitoring component and infrastructure, coupled with matching analysis and modeling capabilities, typically referred to as Online Data Analytics (ODA) – elevating always-on performance measurements to a first-class citizen in the supercomputer.

Dynamism requires focusing on a whole machine or even center, seeing how jobs and systems interact (i.e., system-wide monitoring), and building a performance model database. This needs comprehensive and highly scalable infrastructures, providing continuous monitoring. Further, the focus should exceed the computing system, including the infrastructure (e.g., power and cooling) and peripheral systems (e.g., network and I/O). For shared resources such as I/O, the ability to determine all the sources of operations is of interest. From an instrumentation point of view, this precludes the enforcement of some relatively intrusive forms of instrumentation, such as recompilation or specialized annotation libraries. The transition to a malleable environment has to be as easy as possible since many applications are reluctant to update their codebase. Instrumentation must be multi-staged, starting from a base of node-level instrumentation and gathering more information from the collaborating binaries, e.g., annotated with additional performance hints, for example, with the Caliper [32] infrastructure. Further, runtimes, such as the ones for MPI or OpenMP, can directly provide monitoring information due to their dedicated tool interfaces [33], [34], [35]. For example, MPI is evolving to provide a multi-instrumentation layer called QMPI [36], enabling concurrent monitoring of the MPI interface. Thanks to this evolution, always-on monitoring of MPI is becoming possible, an approach currently limited to nodes' status (memory, health, temperature, etc.).

Many centers focus on developing such monitoring infrastructures, often with the goal of system tracking and tuning. A prominent example, currently deployed on production systems at LRZ and extended in the projects DEEP-SEA [37] and REGALE [38], is the Data Center Data Base (DCDB) [39], [40], which is capable of routinely tracking millions of sensors on large scale production systems, such as SuperMUC-NG, using technologies from the IoT space combined with a federation of time series databases built in top of Cassandra. Similarly, the ADMIRE project [41] is building an entirely new measurement infrastructure relying on the Prometheus time-series database (TSDB) connected to a node-level aggregating push gateway coupled with LIMITLESS [42] for node-level monitoring and high-speed spatial reduction based on a tree-based overlay network (TBON). Although such comprehensive monitoring capabilities are not yet commonly used in all parallel systems, previous experiments have demonstrated the usefulness of malleability at the application level. For instance, the FlexMPI runtime [43] and TALP [44] collect various performance metrics related to each MPI call, which can be utilized to develop performance models and support decision-making on expanding the number of processes. However, monitoring represents only a fraction of the challenges involved in supporting dynamic resources. As we will elaborate in the subsequent sections, a paradigm shift is necessary in every layer of the computing architecture to facilitate such novel scenarios.

IV. Malleability on HPC Systems

The software stack of HPC systems consists of several layers (see Fig. 2) that require adaptations for malleability. These layers are examined in the following subsections.

A. *Programming Models*

Application developers use programming models to write HPC applications. These models need to provide *standardized, flexible, and practical interfaces* to include malleability into the application-specific workflow. Originally, malleability leveraged advanced checkpoint/restart mechanisms (e.g., SCR [2]), or fault tolerance systems for MPI (e.g., ULFM [45]) to implement automatic resizes. It has been a long way with extensive research from malleability-specific efforts, such as ReSHAPE [3], Elastic MPI [8], and Adaptive MPI (AMPI) on top of Charm++'s runtime system [46] as discussed in Aliaga et al. [22], to approaches currently evolving. Along this way, the standardization of MPI and PMIx has progressed to support the integration of malleability features and interfaces into existing programming models. Table I shows an overview of the programming model implementations for message passing, PGAS, and task-based programming discussed here in regard to their application and system programming interfaces for active and passive malleability and data migration support.

1) MPI Sessions: The MPI Standard 4.0 introduces the MPI Sessions model, which promises more resource flexibility by avoiding the global MPI world communicator and tighter integration of runtime information. It introduces the concept of MPI process sets (PSets) as a mechanism to set up communicators in a more modular and scalable way. PSets can be looked up in a dictionary with URIs referring to particular sets. Beyond the mandatory `mpi://SELF` and `mpi://WORLD` process sets, the MPI standard allows the definition of additional process sets before and during the runtime of an MPI application. Malleability could be already supported by adding new URIs referring to different sets of resources. However, the standard does not define anything beyond that, yet. Two approaches, currently investigated to (fully) support malleability with MPI Sessions, are discussed in Sections IV-A2 and IV-A3.

2) ParaStation MPI: ParaStation MPI [47], [56] is an open source MPICH [57] derivate and especially designed to support systems following the Modular Supercomputing Architecture (MSA) [58]. It is a production-level MPI implementation compatible to MPI-4, including support for MPI Sessions (see Section IV-A1). It has been extended to support Psets provided by the runtime environment via the PMIx interface. Furthermore, MPI extensions have been added to ParaStation MPI as user interfaces for active malleability. These extensions enable the

TABLE I
OVERVIEW OF PROGRAMMING MODEL IMPLEMENTATIONS SUPPORTING MALLEABILITY

	Approach	Model	Application interface	System interface	Active malleability	Passive malleability	Data migration
Message Passing	ParaStation MPI [47]	MPI	MPI Sessions, MPI ext.	PMIx	PMIx process groups, alloc. req., spawn	Active mechanism + relaying system req. via PMIx Psets/ MPI Info	Not covered
	Dynamic MPI Processes [24]	MPI	PSets, MPI-4 ext., MPI-indep. sched. API	PMIx	PMIx-based API using unified resource opt. language		Not covered
	FlexMPI [43]	MPI	MPI-2 & ext., API, planned: MPI Sessions	Custom API	ADMIRE API call	ADMIRE API in coordination with Slurm	Selected cases
	DLB Library [48]	MPI/ OpenMP	OpenMP & custom API	Custom API	Custom API	MPI or DROM	Selected cases (shared mem.)
	DMR API [49]/ DMRlib [13]	OmpSs/ MPI	MPI & custom API	Slurm API & MPI	MPI & custom API		Fully covered/ Selected cases
	Adaptive MPI / Charm++ [46]	MPI	AMPI_Migrate	PM2 API	Via AMPI_Migrate	Not supported	Via checkpoint-restart
PGAS	GASPI [50] and GPI-2 [51]	PGAS	API to add/remove processes	PMIx	Planned via PMIx	Via checkpoint- restart	Not covered
	X10 Invasive Framework [52]	X10 PGAS	Invasive comp. API	Custom API	API using embedded constraint description language		Fully covered
Task	GPI-Space [53]	Task-based	RPCs to runtime add/remove methods	Slurm cmd line API	Workflow-driven, RPCs to runtime system	Not supported	Not covered
	OmpSs-2@Cluster [54]	OmpSs-2	API call to add/remove nodes	Slurm API & MPI	Supported via Nanos6 API call	Future work	Fully covered
	PyCOMPSs/ COMPSs [55]	PyCOMPSs/ COMPSs	Fully transparent to app. (hints in submission cmd)	Slurm cmd line API & spawn new procs.	Runtime ctrl. based on est. workload & overheads	Not supported	Fully covered

exploitation of the re-initialization capability of MPI Sessions for dynamically changing the number of MPI processes. They are based on the PMIx process group, PMIx spawn, and PMIx allocation request APIs. Existing MPI Sessions are finalized and one new MPI Session is initialized to exclude terminated and include additional processes in the application. The MPI extensions also provide interfaces to return unused resources or request additional resources from the resource management system in an asynchronous way. They have been demonstrated successfully for active expansions of MPI applications on a prototype system. In future work, passive malleability can be supported via the active malleability mechanisms plus relaying of resource change requests from outside the application via Psets and/or the MPI extensions.

3) Dynamic Processes With Process Sets (DPP): A recent approach [24], [59], [60] introduced dynamic MPI Process interface extensions and an implementation based on Open MPI [61]. This approach follows the principles further described in [60] allowing, e.g., adding/removing processes to/from the application at runtime. The interface abstracts resources as sets, and resource (re-)assignments as set operations, allowing the description of the applications' resource assignments as a graph. From the application perspective, the interface allows users to query resource change information, apply set operations to create new process sets (PSets), where processes are related to resources, and join with dynamically added PSets without building or destroying entire MPI Sessions. This implementation is based on OpenPMIx, including extensions of it. It will be extended to provide a more flexible interface for complex application workflows and tighter integration with global system optimization. The DPP principles [60] target to cover all required aspects of dynamic resources, e.g., 1) a unified interface covering all kinds of allocation requests, performance hints, and monitoring data, 2) a change of resources to be application-driven, system-driven or both, 3) a flexible approach covering all kinds of applications to avoid an application-specific lock-in, which will be based on 4) set operations describing the particular change (added, removed, split, etc.).

4) FlexMPI: FlexMPI [43] provides malleable capabilities to MPI applications in order to dynamically spawn and shrink the number of processes in runtime. FlexMPI also includes features for balancing the application workload (by means of automatic data redistribution) and monitoring the application performance. The programming model corresponds to the Single-Program-Multiple-Data (SPMD). The malleable code section -which usually corresponds to the iterative part of the code- is delimited by control points introduced as FlexMPI library calls. In the current version of FlexMPI runtime the malleable decisions (increasing or decreasing the application's number of processes or performing an automatic load balancing) are provided by an external controller (for instance, ADMIRE's Intelligent Controller) that is responsible for coordinating the decisions based on a holistic view of the system. Additionally, FlexMPI is integrated with Slurm in order to dynamically allocate or release certain compute nodes depending on the malleable actions that are taken. Besides investigating the use of malleability for improving application performance, FlexMPI has been also used to explore the use of malleability for developing application schedulers that permit application migration, I/O scheduling, and the development of application performance models.

5) Dynamic Load Balancing Library (DLB) for MPI: DLB [62], is a dynamic user-transparent library that improves the load balancing of hybrid applications by managing the number of threads of each MPI process. The library is compatible with MPI, OpenMP, and OmpSs. Since version 3.0, DLB includes three independent and complementary modules: LeWI (Lend When Idle), DROM (Dynamic Resource Ownership Management), and TALP (Tracking Application Live Performance).

The LeWI module [48] is used in hybrid MPI + OpenMP/OmpSs applications to dynamically and transparently change the cores and number of threads assigned to a process. Once a process reaches a state where its threads are idle, it may temporarily yield its cores to another process that would benefit from them. This improves the load distribution by efficiently using computational resources. LeWI realizes malleability through programming models (e.g., OpenMP and OmpSs) to dynamically adjust these resources.

DLB relies on the PMPI interface to intercept MPI-blocking calls. Regarding OpenMP, it uses the public API to change the number of threads or the OMPT interface based on availability. These features enable DLB to work transparently with the application without requiring code recompilation or modification. Moreover, a public API is provided that can be used by applications if the programmer has hints to provide or the application is DLB-aware.

6) Dynamic Management of Resources Library (DMRlib): DMRLib [13], which derives itself from the DMR API [49], is a malleability framework devised to orchestrate job reconfigurations rescaling on-the-fly the number of MPI processes. DMRlib is conceived to facilitate programmability by automating resource reallocation, process handling, and data redistribution. DMRlib implements reconfiguration policies to adjust the number of MPI ranks based on 1) job execution performance metrics with TALP [44] and 2) global cluster metrics such as resource availability and pending jobs.

DMRlib comprises a resource management system (RMS) and an MPI-based parallel distributed runtime. DMRlib provides the communication layer between them and allows seamlessly malleable application execution in a cluster workload. Slurm, the RMS used, monitors the resource utilization and job requests. It has been extended to include the capability of scheduling malleable jobs and managing dynamic resources.

DMRlib enables malleability in a wide range of applications, not only the traditional iterative applications like Jacobi, CG, N-body, and LAMMPS but also producer-consumer bioinformatics applications [30]. Furthermore, DMRlib is the first malleability framework to report a malleable GPU-enabled application [11].

7) GASPI and GPI-2: GPI-2 [51] is a scalable and fault-tolerant open-source implementation of the GASPI [50] standard specification. It implements a PGAS parallel programming model and relies on one-sided asynchronous communication and fine-grained synchronization of accesses to the partitioned global address space for high scalability and flexibility when developing parallel applications. GPI-2 provides the capability to dynamically adjust the number of processes to ensure fault tolerance [63]. To avoid indefinite blocking, remote operations can have a timeout value, and an error state vector is accessible if any issues arise. Through these features, applications can flexibly adjust to function with the remaining healthy processes. Moreover, a failed process can be replaced by implementing a suitable checkpointing scheme [64].

To support the release and request of dynamic resource allocations, GPI-2 is being extended to implement the PMIx client interface. GPI-2 will support malleable and evolving applications by implementing new interfaces to request resource changes at runtime and responding to resource changes initiated by schedulers and resource managers, respectively. The capabilities will support adaptation of inter-process communication and coordination of processes in applications to changed resources. GASPI will be extended to standardize the implementation of new interfaces for adaptations to resource changes by applications initiated by schedulers and resource managers or by applications themselves.

8) GPI-Space: GPI-Space is a task-based workflow management system for parallel applications [53]. It is designed to automatically coordinate scalable, parallel executions in large, complex environments, being currently used as a development and execution platform for various workflow-driven applications from different domains [65], [66], [67], [68], [69]. GPI-Space allows domain developers to build domain-specific workflows using their own parallelization patterns, data management policies, and I/O routines, while relying on the runtime system to address general aspects related to scheduling, distributed memory management, task execution and tolerance to failures. Further, it supports adding or removing workers at runtime without interrupting the application. The event-driven underlying architecture of the runtime system allows the coordinating component (the *Agent*) to quickly react to events such as worker registrations or disconnections and take appropriate decisions with respect to the execution of a workflow (e.g., task assignment, cancellation, and rescheduling) while ensuring its progress. In the case of worker registrations, the new workers are immediately scheduled and served tasks to execute. In case of worker disconnections, all the tasks previously scheduled to them are rescheduled to the remaining workers such that load balance is preserved and fairness respected. This allows running the GPI-Space applications in a malleable way and to be tolerant to worker failures. Currently, the addition or removal of workers can be done interactively by the user at runtime using RPC commands. This can also be done automatically by embedding specialized transitions that invoke specific RPC commands to be executed by the runtime system. The node removal is also automatically handled by the runtime system when some nodes become suddenly unavailable for various reasons, e.g., hardware failure or allocation termination.

9) OmpSs-2@Cluster: OmpSs-2 [70] is a task-based programming model, similar to OpenMP, which supports task nesting, dependencies among tasks and an advanced dependency system. OmpSs-2@Cluster [54] is the task offloading extension of OmpSs-2, which executes tasks across multiple nodes with distributed memory. OmpSs-2@Cluster can be used either on its own or in a hybrid configuration in combination with MPI [71]. Task ordering is inherited from the sequential version of the code, and there is a common virtual address space across workers. OmpSs-2@Cluster benefits from core-level malleability through integration with the Dynamic Load Balancing (DLB) library. OmpSs-2's malleable thread execution model is used to transparently load balance MPI+OmpSs-2 programs across nodes [71].

OmpSs-2@Cluster provides a single API call to dynamically change the number of nodes [72]. The runtime interacts with the resource management system to add or remove compute resources, it automatically redistributes the application's data and subsequently schedules tasks across the new number of nodes. The task-based programming model provides an easy route for an application to support malleability, since the application is written in a way that is generally independent of the underlying compute resources. The current runtime implementation uses MPI_Comm_spawn, but it will be updated to use MPI Sessions (see Section IV-A3), which provides greater flexibility to add and remove resources at arbitrary granularities. In ongoing work, the malleability support will be extended for compatibility with hybrid MPI+OmpSs-2@Cluster.

10) PyCOMPSs/COMPSs: PyCOMPSs/COMPSs [55] is a task-based programming model that tackles large granularity tasks and aims to be executed in distributed environments. PyCOMPSs provides a programming model and a runtime system allowing developers to easily convert a sequential Python script to parallel workflows for distributed computing environments, hiding the complexity of the parallelization and of the execution management. A PyCOMPSs application can be represented as a Direct-Acyclic-Graph (DAG) that stores information about the computational load required by the application at any point of the execution. The runtime has an autoscaling module that is able to scale up and down the computing resources used by the application according to its demands. Combining runtime profiling information with the task dependency graph, the PyCOMPSs runtime periodically estimates the remaining parallel workload and the computing infrastructure capacity. These metrics are useful to determine when to add or remove computing resources. In previous versions of PyCOMPSs [73], the auto-scaling features were applied to scale service executions in cloud environments where the runtime contacts the resource provider API to create and destroy virtual machines. The auto-scaling feature has been extended to auto-scale scientific workflows in HPC clusters, such that the current runtime interacts with Slurm [74].

B. Process Manager / Runtime Environment

The process manager or runtime environment is responsible for executing applications on the computing resources. It interacts with them (via the programming model agnostic abstraction layer) to execute malleability operations and with the global resource management system/scheduler to change the resource allocation. At this layer, application-level monitoring can be placed and a local scheduling component can be located to optimize resource assignments inside and between the managed applications.

1) ParaStation Management: ParaStation Management [75], [76], [77] offers a complete open-source process management system that can be combined with an outer and more generic resource manager, together with a batch queuing system and a job scheduler such as Slurm. With ParaStation Management, processes can be started on remote nodes, communication channels of remotely started processes can be controlled, and signals across node boundaries can be managed. The main component of ParaStation Management is the ParaStation Daemon (psid) running on each compute node and forming a scalable network of daemons.

The functionality of the psid can be extended by plugins. The ParaStation Slurm (psslurm) plugin enables the psid to communicate with the Slurm scheduler and launcher via Slurm Remote Procedure Call messages. To support malleability, this plugin will be extended to enable the coordination of resource allocation requests between the Slurm scheduler and the node-local process management system. The ParaStation PMIx (pspmix) plugin provides a PMIx server in the psid to which the processes of an application connect as PMIx clients. In the context of malleability, the pspmix plugin will serve as the interface between the application and the process management. It will provide the PMIx allocation requests API to clients so that resource changes can be requested by applications and can be provided to them as new psets. Any plugin of ParaStation Management can be interfaced via a hook mechanism in the psid, which enables the forwarding of a resource allocation request from pspmix via psslurm to the Slurm scheduler and vice versa for a response of the scheduler.

2) ADMIRE's Malleability Workflow: The Intelligent Controller (IC) supports the execution of applications for maximizing the usage of the computational and I/O resources as well as the global system performance. To achieve these goals, the IC provides control mechanisms to coordinate the execution of the components of the ADMIRE framework. These components are: the applications, the system and application monitoring tools, the system job scheduler, the resource manager (e.g., Slurm), and the I/O subsystem. By coordinating these components, the IC creates a distributed control infrastructure that provides a single and global view of the system.

The malleability workflow is described by two communication channels: 1) the connection between the IC and 2) the resource manager (Slurm) and the connection between the applications and the IC. Currently, ADMIRE uses Slurm as a resource manager. Via the Slurm connector, the IC can send instructions to Slurm to allocate and deallocate resources for each application. Using ADMIRE's communication library, the applications can exchange information related to malleability with the IC and execute reconfigurations depending on the existing policies implemented in the IC.

3) PMIx Reference Runtime Environment: The PMIx Reference Runtime Environment (PRRTE) [78] is a reference implementation of a PMIx-enabled runtime environment developed by the PMIx community. PRRTE provides general process management and runtime environment services for PMIx-based HPC jobs and is the native runtime environment of Open MPI. It could potentially provide common malleability mechanisms to different programming models, since it is not tailored to a specific one. PRRTE's (v3.0.0) malleability support is currently limited to dynamically spawning new processes on the resources of the current allocation due to the lack of integration with system-level resource managers.

Recently, a prototype was developed, which extends the PRRTE, OpenPMIx, and Open MPI implementations to support a dynamic MPI Sessions interface [24]. These extensions allow applications to request the addition of processes to and removal of processes from the application during runtime. Moreover, applications can query the PRRTE runtime for (potentially system-driven) addition/removal of processes and indicate the successful adaption to such changes. So far, this implementation still relies on the resource allocation initially granted by the system-level resource manager. To this end, the prototype is planned to be extended with a PMIx-based interaction between PRRTE and the system-level resource manager through the usage of the PMIx_Allocation_request, to dynamically change the current resource allocation.

C. Malleable Scheduler and Resource Manager

At the node management layer in HPC, malleable job scheduling refers to the process of dynamically allocating or deallocating resources during job execution to adapt the resource usage dynamically, depending on the system conditions and the application needs. The jobs receive the support to use more or fewer resources in real-time, scaling up or down their performance without the need to terminate and restart. Scheduling malleable jobs efficiently introduces new challenges that do not exist in traditional non-malleable platforms. To support dynamic job reconfiguration, challenges arise including, e.g., data redistribution which usually occurs after a malleable operation (shrink or expansion). Related topics such as the management of large unstructured data or complex data distributions (e.g., with data replication in the boundary areas) must be also considered. Moreover, mechanisms for transferring the application state to newly created processes while keeping the application in a consistent state must be defined. Another challenge is the reconfiguration knowledge required by the scheduler, which needs *a priori* information about how the new configuration impacts the application behavior to determine whether the reconfiguration is worthy or not. This usually needs a precise estimation of the application performance using historical data or application modeling (see Section III-B). This is just the tip of the iceberg, as many more challenges exist, including the interaction between I/O and computational malleability, data dependencies during different phases of the job, and scheduling time windows.

Scheduling malleable jobs is challenging since it requires considering (via monitoring) the complete and updated system and the knowledge about the application scaling performance (via performance models). Traditional job scheduling systems (e.g., batch schedulers) are not suitable for malleable jobs. One suitable approach is to use an intelligent resource manager that has a holistic vision of the system and the running application capable of allocating resources dynamically. This allows the system to allocate the resources for each job in real-time, while considering the current platform status and the requirements of both the executing and queued jobs. Here, simulators that support malleability, such as ElastiSim [79], can facilitate and expedite the development of novel scheduling policies.

V. Malleability on Storage Resources

I/O malleability focuses on the dynamism of I/O services and storage resources. Interestingly, I/O malleability can leverage the compute nodes from a job allocation directly as storage resources to increase I/O performance and mitigate negative side effects on the *parallel file system* (PFS) due to uncoordinated I/O. In fact, I/O malleability pursues similar goals as computational malleability, e.g., improving the job scheduler pipeline, increasing resource usage efficiency, and reducing application runtime. Hence, applying both methods complementary may lead to increased benefits. Moreover, they share most of the necessary architectural components (see Section I), as we discuss in the following.

A common PFS approach to overcome the performance penalty of magnetic disks is to introduce a caching layer that stores frequently accessed data on fast SSD devices. This layer can be directly implemented inside the file system [80], [81], [82], be part of an external burst buffer [83], [84], or can use SSDs inside compute nodes [85], [86], [87]. However, the degradation in I/O performance caused by cross-application interference [88], [89], [90], is harder to overcome. Directly preventing I/O interference from within a malleable scheduler would require the scheduler to include application I/O patterns in its decisions and to restrict the number of simultaneously running I/O-intensive applications. The necessary information for this is often unavailable, and the resulting logic significantly complicates the scheduling and might artificially delay the execution of applications. Nonetheless, a co-design between scheduler and applications can shift non-critical I/O phases of an application (such as check-pointing) in time to avoid potential simultaneous accesses [91], reducing interference.

Further control of application impact on I/O can be achieved with QoS extensions for PFSs [92], [93], such as the token bucket filter in the Lustre file system's network request scheduler [94]. This can limit the number of data and metadata requests to prevent a single application from saturating the storage. However, QoS control alone is often not enough to overcome the resulting interference of access patterns and performance limitations from the underlying storage systems. *Ad-hoc file systems* are a recent storage abstraction [95] that helps reduce I/O interference by deploying application-specific file systems using the node-local storage resources assigned to a job. This allows applications to run effectively isolated from the PFS, which is only accessed to import input datasets for the application (*stage-in*) or to export its results for long-term storage (*stage-out*). Ad-hoc storage systems offer a suitable platform to deploy I/O malleability techniques, which is why we focus on them in the remainder of this section.

Several components are needed to enable large-scale usage of ad-hoc storage. Whereas Fig. 1 presented the overall architecture for a malleable HPC system, Fig. 4 depicts the major five components in detail needed to realize this vision: 1) The ad-hoc file system itself, 2) a data scheduler and management component controlling the ad-hoc file systems and the data flow between them and the PFS, 3) a malleability manager determining when and which malleable aspects of the ad-hoc file system and

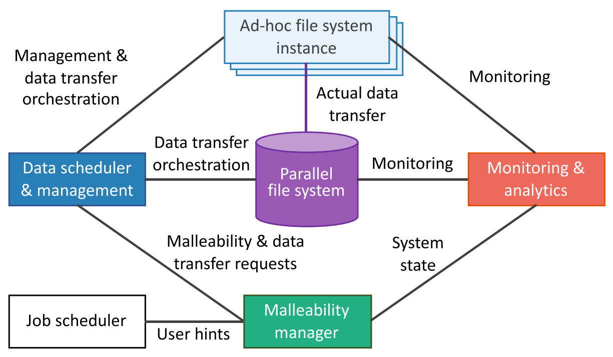
Ad-hoc file system instance
Management & data transfer orchestration
Monitoring
Actual data transfer
Data transfer orchestration
Data scheduler & management
Parallel file system
Monitoring
Monitoring & analytics
Malleability & data transfer requests
System state
Job scheduler
User hints
Malleability manager

Fig. 4. Proposed I/O malleability HPC architecture.

the data transfers are changed, 4) a monitoring and analytics component keeping track of the system's resource usage, and 5) a job scheduler offering an additional interface for optional hints. Next, we discuss the first three components, as the monitoring and analytics component was handled in Section III-B, and the job scheduler was covered in Section IV-C.

A. *Ad-Hoc File Systems*

Several ad-hoc file system implementations exist, such as GekkoFS [96], BurstFS [97], Hercules [98], Expand [99], UnifyFS [100], BeeGFS BeeOND [101], and others [102], [103], each targeting slightly different use cases. These implementations can be dynamically started and stopped within a given context (e.g., a compute job) and provision storage using existing node-local flash-based storage that usually remains unused by applications or even spare memory if available. Since an ad-hoc file system can form a single namespace by combining the devices' capacity and accumulating their I/O performance, ad-hoc file systems can scale performance almost linearly with the number of nodes used as servers [95].

Ad-hoc file systems directly support malleability by adapting to a given use case. Most HPC applications, e.g., do not require strong consistency semantics and rarely use certain I/O operations like `rename()` or `readdir()` [104], [105]. Ad-hoc file systems can thus be configured for these applications by relaxing cache consistency guarantees, increasing the performance of a file system by simplifying its protocols. Other examples of settings for I/O malleability are increasing write-back caching intervals to foster data locality or QoS mechanisms for managing applications and network interference. More interestingly, HPC workflows can dynamically tune ad-hoc file systems for each workflow phase by expanding or reducing the number of compute nodes used as I/O servers. Note, however, that this reconfiguration needs potentially large amounts of data to be redistributed between the new I/O servers. While the scale of this data will be smaller than the PFS', the cost of this redistribution cannot be neglected [106].

Deploying ad-hoc storage systems as fundamental components of an I/O-malleable ecosystem requires a management component and user support: 1) Data staging between different ad-hoc storage instances needs to be orchestrated to prevent uncoordinated competition; 2) Each ad-hoc file system needs to be tuned for its particular use case. The latter typically does not require special permissions, as most ad-hoc file systems run in user space, but it does require prior knowledge of an application's I/O behavior, which HPC users typically do not have; 3) Complex applications like Tensorflow [107] or OpenFoam [108] can exhibit highly complex I/O workflows and require dynamic reconfiguration of the I/O resources that requires support from the scheduling environment.

B. *Data Scheduler and Management*

The input data has to be staged into the ad-hoc file system before the application can access it, and output data must be staged out of it before the ad-hoc instance is terminated. The data scheduler is responsible for interacting with the ad-hoc file systems to maximize the optimization of the overall system's resources. However, the data scheduler cannot work effectively without user input and, at the very minimum, needs to know which paths contain an application's input and where the output should be placed. The data scheduler transforms the user hints captured by the job scheduler upon job submission and deploys ad-hoc file system instances as needed. Once an ad-hoc instance is deployed, the data scheduler orchestrates data transfers from the PFS to feed it with data, or to the PFS to transfer data to persistent long-term storage. These data transfers can also be malleable, i.e., the number of processes that are involved and their QoS limitations are dynamic and controlled at runtime. Since data transfers correspond to actual PFS accesses, properly orchestrating them is crucial to minimize congestion and maximize I/O performance. These tasks are not without challenges because 1) ad-hoc storage space is limited, hence, any scheduling algorithm controlling the transfers should strive to keep enough free space in the ad-hoc file system to ensure applications do not receive an out-of-space error; 2) applications can make progress as long as they have the data they need, which means that data should be staged concurrently with application computations and should ideally be available just before an application accesses the data of interest; and 3) data that is meant to be persistent should be staged out to the PFS as soon as it is no longer needed. Thus, for data staging to happen on time, application I/O models need to be investigated that can accurately predict the frequency and volumes of application I/O phases. Moreover, for effective data scheduling, these models could be extrapolated to different node and process configurations. While several services were proposed to support data staging [109], [110], [111], [112], to the best of our knowledge, none consider these issues.

C. *Malleability Manager*

This component has access to the state of the HPC system's resources and makes malleable decisions both targeting ad-hoc file systems and data transfers. A malleable request is sent to the data scheduler, executed, or forwarded depending on whether it targets an ad-hoc file system or a data transfer. Therefore, such decisions can only be made if the following information is available: 1) The current load of the PFS, 2) an advanced understanding of an application's I/O requirements and phases, and 3) knowledge about the ad-hoc storage system's performance capabilities. Such malleable decisions can target several parts of an ad-hoc file system, e.g., optimizing semantic protocols or applying QoS restrictions. For data transfers, the malleability manager determines when and at which speed data transfers occur. Finally, the malleability manager takes known application phases into account to minimize any I/O interference. For example, using the data scheduler, the staging process can transform random I/O that occurs in bursts on the ad-hoc file system into sequential I/O on the PFS that is spread over a longer time frame during a computation phase.

VI. Current Challenges of Malleability

As presented, the support of malleability for traditional HPC workloads requires adaptations across the entire HPC software stack. The major challenges that need to be solved are:

1) *Resource optimization:* Resource management under malleability poses various new challenges requiring the incorporation of application- and monitoring-driven data, allowing decisions to be made within a reasonable time frame. These decisions can have different targets, e.g., improved application throughput, reduced energy consumption, or QoS for I/O. For instance, data redistribution costs could be reduced by timing these operations strategically.
2) *Monitoring optimization information:* Monitoring provides insight into the application's performance. The challenges we see, are the difficulties in processing the vast amount of monitoring data under the presence of compute and I/O malleability. This provides the basis for improved malleable models used in resource optimization.
3) *Program optimization information:* We see the strong requirement of application developers to aid resource optimization by providing further information. The challenge is lowering the programming efforts to extract performance information for performance optimizations compatible with monitoring information.
4) *Malleability for the masses:* To be successful, we need not only a standardized malleable programming model across the HPC community but also production-ready implementations and a wide adoption in frameworks for semi-transparent usage of malleability.
5) *HPC hard-/software components:* Heterogeneous hard-/software components already pose challenges to cope with them. Malleability is even further complicated when considering, for example, accelerators like GPUs. While this can, in principle, be handled by malleability approaches on the host system or process level, it also introduces new challenges and opportunities, e.g., by taking advantage of active resource partitioning on GPUs [113].

VII. Guiding the Future Research & Conclusion

Looking back at the question raised in Section I, the sluggish progress in making malleability accessible to applications and systems presents itself as a chicken-and-egg problem. Even if it was supported on the system side, application developers, especially those of large and historically grown code bases, would face substantial practical challenges when trying to introduce malleability features. Knowing this, system architects do not see enough incentive to create such support in the first place. The increasing number of distributed AI training jobs, which are malleable by design, starts slightly tipping the balance in favor of malleability – but not yet strongly enough.

Some obstacles application developers need to overcome stem from data redistribution after shrink or expand requests. How difficult this is depends on the design pattern applied to parallelize the program. Task-based programs submit small work units to a task pool, from which processes or threads can retrieve them for completion, potentially submitting new tasks while processing existing ones. As such tasks are usually short-lived and do not require much communication among themselves, they form an elastic collection that can be comfortably redistributed whenever nodes are added or removed. While common on systems with shared memory and supported by standard APIs such as OpenMP, this pattern is rarely found on large-scale systems with distributed memory. PyCOMPSs/COMPSs [55] is an attempt to popularize task-based programming on such systems.

Yet, not all problems are equally suitable for task-based programming. Many require geometric domain decomposition, in which the simulated domain (e.g., the ocean, the atmosphere, a turbine) is divided into subdomains to be mapped onto processing elements. Especially for irregular domains lacking symmetry or homogeneity, this division is non-trivial and requires complex calculations using space-filling curves or sophisticated tools such as ParMETIS. Dynamic redistribution, whether by checkpointing, followed by restart in a different configuration – basically, malleability through the backdoor, for which moldability is sufficient – or by marshaling the data into a new configuration across the network, always has to solve the same *complex division problem*.

An alternative way of looking at this problem, combining aspects of task-based programming with geometric decomposition, is applying the principle of over-decomposition. Over-decomposition is ubiquitous in task-based programming but also beyond. For example, it is one of the cornerstones of Charm++ [114], a processor-independent programming system for large-scale HPC. The idea is to divide the problem into several sub-tasks whose number exceeds the degree of hardware parallelism by far. An adaptive runtime system then distributes these sub-tasks across the available parallel hardware, allowing it to balance the load automatically. This separation of concerns made Charm++ an early adopter of malleability – as a pretty natural extension. Although it already enjoys a very large number of users, it has not yet become mainstream for reasons related to another trade-off. Because most novel HPC programming systems are created and maintained by researchers, and their funding suffers from significant volatility, application developers, whose codes may live on for decades, are highly conservative, trying to stay on trodden paths and not straying away outside the realm of established community standards.

If malleability is to succeed, there is a need for a *standardized interface for dynamic resources*. The necessary prerequisites should, therefore, be fulfilled within the confines of the standard programming approach, which still is MPI+X. While instruments to shrink and expand MPI communicators are already part of the standard, data redistribution has always been regarded as the sole responsibility of the programmer. AMPI [46], an implementation of MPI based on the Charm++ runtime system, addresses this problem by representing MPI ranks as lightweight user-level migratable threads with the option of running multiple of them on a single core. It essentially interprets the idea of over-decomposition as creating an oversized collection of ranks, which can then be flexibly mapped on an arbitrary and potentially changing number of physical cores. While this is an elegant solution to the problem of making MPI jobs malleable, the virtualization of MPI ranks may suffer from undesired communication and context-switching overhead when the number of ranks exceeds the number of cores. However, there might be another way to shift at least a good portion of the redistribution task to the MPI standard - closer to the original design philosophy of MPI, combining the notion of virtual process topologies, expressed as a graph or Cartesian grid, with the concept of over-decomposition. If there was a controlled way of splitting arbitrary and (re-)merging adjacent MPI ranks and their associated data, refining or coarsening neighborhood relationships of the topology along the way, an MPI job could be more easily and automatically adapted to changing resource conditions. Essentially, the redistribution problem would be reduced to a local operation the user still has to implement – but without the need for laborious and cumbersome global repartitioning. Splitting all ranks evenly across the entire job to occupy twice the number of nodes would even retain load balance if it had existed before. While this is still a very early idea that still needs to be proven, we believe its conceptual simplicity makes it at least worth a try. Moreover, based on our experience, we saw that individual efforts barely moved things forward due to the high specialization of what was moved. The development of a standardized interface covering malleability in MPI, but also beyond MPI with converged computing is required for continuous integration and long-term support.

Programmability is the second aspect the success of malleability relies on. Malleability can only be successful if it is widely adopted by, e.g., applications, frameworks, programming models, and DSLs. This requires it to be programmable in an easy way. Here, we envision hiding this complexity in three different ways: 1) Provide malleability support within commonly used parallel software and parallel libraries, which lowers the bar for using malleability (see Huber et al. [24], which uses a dynamic resource extension for p4est) or makes malleability almost transparent to the application developers using these updated software packages. 2) We envision a standardized layer between the application and MPI that provides various functionalities to again lower the bar for utilizing malleability. E.g., the DMRLib [13] development already goes in this direction. 3) To still support all other cases and corner cases, direct access to the dynamic resource MPI layer is still provided [24], [59]; however, likely to be more complex to program. We see programmability as one of the crucial points that is, however, currently investigated insufficiently.

Finally, while several more points can be added, other important aspects include developing components that drive the users towards malleability, which is currently in progress with changes in, e.g., PMIx, MPI, and schedulers like SLURM, but more work is required, particularly on the application side and the scheduler (e.g., FLUX). Moreover, the development of novel scheduling algorithms co-designed with application and monitoring frameworks/libraries could also facilitate the adaption of malleability.

Summing up, various challenges retain malleability implementations in HPC. Based on the state-of-the-art, we listed the main challenges we expect and experienced through recent efforts in this field across different layers of the software stack. These challenges are worth facing due to the advantages malleability brings to the HPC systems and users. To facilitate the process, we listed improvement aspects to guide future research. We expect malleability to be a key component that will lead to a new era of supercomputers.

Acknowledgment

The authors gratefully thank Daniel Milroy and Isaías Comprés for their help and suggestions for this work.

References

[1] S. S. Vadhiyar and J. J. Dongarra, "SRS—A framework for developing malleable and migratable applications for distributed systems," *Parallel Process. Lett.*, vol. 2, pp. 291–312, 2002.
[2] K. El Maghraoui et al., "An architecture for reconfigurable iterative MPI applications in dynamic environments," in *Proc. Int. Conf. Parallel Process. Appl. Math.*, Berlin, Germany, 2006, pp. 258–271.
[3] R. Sudarsan and C. J. Ribbens, "ReSHAPE: A framework for dynamic resizing and scheduling of homogeneous applications in a parallel environment," in *Proc. Int. Conf. Parallel Process.*, 2007, pp. 44–44.
[4] G. Martín et al., "FLEX-MPI: An MPI extension for supporting dynamic load balancing on heterogeneous non-dedicated systems," in *Proc. Eur. Conf. Parallel Process.*, Berlin, Germany, 2013, pp. 138–149.
[5] F. S. Ribeiro et al., "Autonomic malleability in iterative MPI applications," in *Proc. Symp. Comput. Architecture High Perform. Comput.*, 2013, pp. 192–199.
[6] M. Schreiber et al., "Invasive compute balancing for applications with shared and hybrid parallelization," *Int. J. Parallel Program.*, vol. 43, no. 6, pp. 1004–1027, 2015.
[7] S. Prabhakaran, M. Neumann, S. Rinke, F. Wolf, A. Gupta, and L. V. Kale, "A batch system with efficient adaptive scheduling for malleable and evolving applications," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2015, pp. 429–438.
[8] I. Comprés et al., "Infrastructure and API extensions for elastic execution of MPI applications," in *Proc. 23rd Eur. MPI Users' Group Meeting*, 2016, pp. 82–97.
[9] S. Iserte et al., "Efficient scalable computing through flexible applications and adaptive workloads," in *Proc. 46th Int. Conf. Parallel Process. Workshops*, 2017, pp. 180–189.
[10] P. Sanders and J. Speck, "Energy efficient frequency scaling and scheduling for malleable tasks," in *Proc. Int. Conf. Parallel Process.*, C. Kaklamanis et al. Eds. Berlin, Germany: Springer, 2012, pp. 167–178.
[11] S. Iserte and K. Rojek, "A study of the effect of process malleability in the energy efficiency on GPU-based clusters," *J. Supercomputing*, vol. 76, pp. 255–274, Oct. 2019.
[12] B. Dupont et al., "Energy-aware scheduling of malleable HPC applications using a particle swarm optimised greedy algorithm," *Sustain. Comput. Inform. Syst.*, vol. 28, 2020, Art. no. 100447.
[13] S. Iserte, R. Mayo, E. S. Quintana-Ortí, and A. J. Peña, "DMRlib: Easy-coding and efficient resource management for job malleability," *IEEE Trans. Comput.*, vol. 70, no. 9, pp. 1443–1457, Sep. 2021.

[14] J. Carretero et al., "Optimizations to enhance sustainability of MPI applications," in *Proc. 21st Eur. MPI Users' Group Meeting*, 2014, pp. 145–150.
[15] M. Rodríguez-Gonzalo et al., "Improving the energy efficiency of MPI applications by means of malleability," in *Proc. 24th Int. Conf. Parallel Distrib. Netw. Based Process.*, 2016, pp. 627–634.
[16] H. Ferreboeuf, "Lean ICT – towards digital sobriety," The Shift Project, Mar. 2019. [Online]. Available: https://theshiftproject.org/wp-content/uploads/2019/03/Lean-ICT-Report_The-Shift-Project_2019.pdf
[17] D. Bernholdt et al., "A survey of MPI usage in the US exascale computing project," *Concurrency Computation: Pract. Experience*, vol. 32, no. 3, Sep. 2018, Art. no. e4851.
[18] J. Hungershofer, "On the combined scheduling of malleable and rigid jobs," in *Proc. 16th Symp. Comput. Architecture High Perform. Comput.*, 2004, pp. 206–213.
[19] S. Iserte, "High-throughput computation through efficient resource management," PhD dissertation, UJI, Castelló (Spain), Nov. 2018.
[20] M. D'Amico et al., "DROM: Enabling efficient and effortless malleability for resource managers," in *Proc. 47th Int. Conf. Parallel Process. Companion*, Eugene OR USA, 2018, pp. 1–10.
[21] R. H. Castain et al., "PMIx: Process management for exascale environments," *Parallel Comput.*, vol. 79, pp. 9–29, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819118302424
[22] J. I. Aliaga et al., "A survey on malleability solutions for high-performance distributed computing," *Appl. Sci.*, vol. 12, no. 10, May 2022, Art. no. 5231.
[23] D. G. Feitelson, "Packing schemes for gang scheduling," in *Proc. Workshop Job Scheduling Strategies Parallel Process.*, Berlin, Germany: Springer, 1996, pp. 89–110.
[24] D. Huber et al., "Towards dynamic resource management with MPI sessions and PMIx," in *Proc. 29th Eur. MPI Users' Group Meeting*, 2022, pp. 57–67.
[25] S. Balay et al., "PETSc Web page," 2022. [Online]. Available: https://petsc.org/
[26] J.-L. Lions et al., "Résolution d'EDP par un schéma en temps pararéel," *Comptes Rendus de l'Académie des Sci. Ser. I Math.*, vol. 332, no. 7, pp. 661–668, Apr. 2001.
[27] J. Hungershofer, "On the combined scheduling of malleable and rigid jobs," in *Proc. 16th Symp. Comput. Architecture High Perform. Comput.*, 2004, pp. 206–213.
[28] G. Martín et al., "EpiGraph: A scalable simulation tool for epidemiological studies," in *Proc. Int. Conf. Bioinf. Comput. Biol.*, 2012, pp. 529–537.
[29] D. D. Luccio et al., "Coastal marine data crowdsourcing using the internet of floating things: Improving the results of a water quality model," *IEEE Access*, vol. 8, pp. 101 209–101 223, 2020.
[30] S. Iserte et al., "Dynamic reconfiguration of noniterative scientific applications," *Int. J. High Perform. Comput. Appl.*, vol. 33, no. 5, pp. 804–816, Sep. 2018.
[31] R. Sudarsan et al., "Dynamic resizing of parallel scientific simulations: A case study using LAMMPS," in *Proc. Int. Conf. Comput. Sci.*, Berlin, Germany: Springer, 2009, pp. 175–184.
[32] D. Boehme et al., "Caliper: Performance introspection for HPC software stacks," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2016, pp. 550–560.
[33] T. Islam et al., "Exploring the capabilities of the new MPI_T interface," in *Proc. 21st Eur. MPI Users' Group Meeting*, 2014, pp. 91–96.
[34] A. E. Eichenberger et al., "OMPT: An OpenMP tools application programming interface for performance analysis," in *OpenMP in the Era of Low Power Devices and Accelerators*. Berlin, Germany: Springer, 2013, pp. 171–185.
[35] S. Ramesh et al., "MPI performance engineering with the MPI tool interface: The integration of MVAPICH and TAU," in *Proc. 24th Eur. MPI Users' Group Meeting*, 2017, pp. 1–11.
[36] B. Elis et al., "QMPI: A next generation MPI profiling interface for modern HPC platforms," in *Proc. 26th Eur. MPI Users' Group Meeting*, 2019, Art. no. 4.
[37] DEEP-SEA EuroHPC project, "DEEP-SEA official website," Apr. 2021. [Online]. Available: https://www.deep-projects.eu/
[38] REGALE EuroHPC project, "REGALE official website," Apr. 2021. [Online]. Available: https://regale-project.eu/
[39] E. Arima et al., "On the convergence of malleability and the HPC power-stack: Exploiting dynamism in over-provisioned and power-constrained HPC systems," in *Proc. Int. Workshops High Perform. Comput.*, 2023, pp. 206–217.
[40] M. Schulz et al., "On the inevitability of integrated HPC systems and how they will change HPC system operations," in *Proc. 11th Int. Symp. Highly Efficient Accel. Reconfigurable Technol.*, 2021, Art. no. 2.
[41] ADMIRE EuroHPC project, "ADMIRE official website," Apr. 2021. [Online]. Available: http://www.admire-eurohpc.eu/
[42] A. Cascajo et al., "LIMITLESS—Light-weight monitoring tool for large scale systems," *Microprocessors Microsystems*, vol. 93, 2022, Art. no. 104586.
[43] G. Martín et al., "Enhancing the performance of malleable MPI applications by using performance-aware dynamic reconfiguration," *Parallel Comput.*, vol. 46, pp. 60–77, Jul. 2015.
[44] V. Lopez et al., "TALP: A lightweight tool to unveil parallel efficiency of large-scale executions," in *Proc. Perform. Eng. Modelling Anal. Vis. Strategy*, 2021, pp. 3–10.
[45] P. Lemarinier et al., "Architecting malleable MPI applications for priority-driven adaptive scheduling," in *Proc. 23rd Eur. MPI Users' Group Meeting*, 2016, pp. 74–81.
[46] C. Huang et al., "Adaptive MPI," in *Languages and Compilers for Parallel Computing*, L. Rauchwerger, Ed., Berlin, Germany: Springer, 2004, pp. 306–322.
[47] S. M. Pickartz, "Virtualization as an enabler for dynamic resource allocation in HPC; 1. Auflage," Dissertation, RWTH Aachen Univ., Aachen, Germany, 2019. [Online]. Available: https://publications.rwth-aachen.de/record/756085
[48] M. Garcia et al., "Hints to improve automatic load balancing with LeWI for hybrid applications," *J. Parallel Distrib. Comput.*, vol. 74, no. 9, pp. 2781–2794, 2014.
[49] S. Iserte et al., "DMR API: Improving cluster productivity by turning applications into malleable," *Parallel Comput.*, vol. 78, pp. 54–66, Oct. 2018.
[50] GASPI-Forum, "GASPI homepage," Nov. 2022. [Online]. Available: http://www.gaspi.de/
[51] Fraunhofer-Gesellschaft zur Förderung der angewandten Forschung e.V., "GPI-2 official website," Nov. 2022. [Online]. Available: http://www.gpi-site.com
[52] H. J. Bungartz et al., "Invasive computing in HPC with X10," in *Proc. 3rd ACM SIGPLAN X10 Workshop*, 2013, pp. 12–19.
[53] Fraunhofer ITWM, Competence Center High Performance Computing, "GPI-Space official website," Sep. 2020. [Online]. Available: https://www.gpi-space.de
[54] J. Aguilar Mena et al., "OmpSs-2@Cluster: Distributed memory execution of nested OpenMP-style tasks," in *Proc. Eur. Conf. Parallel Process.*, Berlin, Germany: Springer, 2022, pp. 319–334.
[55] E. Tejedor et al., "PyCOMPSs: Parallel computational workflows in Python," *Int. J. High Perform. Comput. Appl.*, vol. 31, pp. 66–82, 2017.
[56] ParTec AG, "ParaStation MPI," ParTec AG, 2023. [Online]. Available: https://github.com/ParaStation/psmpi
[57] W. Gropp, "MPICH2: A new start for MPI implementations," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, D. Kranzlmüller et al. Eds. Berlin, Germany: Springer, 2002.
[58] E. Suarez et al., "White paper: Modular supercomputing architecture: A success story of European R&D," ETP4HPC, 2022. [Online]. Available: https://www.etp4hpc.eu/pujades/files/ETP4HPC_WP_MSA_20220519.pdf
[59] J. Fecht et al., "An emulation layer for dynamic resources with MPI sessions," in *Proc. Malleability Techn. Appl. High Perform. Comput.*, Cham: Springer, 2022, pp. 147–161. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03856702
[60] D. Huber et al., "Design principles of dynamic resource management for high-performance parallel programming models," 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2403.17107
[61] E. Gabriel et al., "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, D. Kranzlmüller et al., Eds. Berlin, Germany: Springer, 2004, pp. 97–104.
[62] M. Garcia-Gasulla et al., "Runtime mechanisms to survive new HPC architectures: A use case in human respiratory simulations," *Int. J. High Perform. Comput. Appl.*, vol. 34, no. 1, pp. 42–56, 2020.
[63] F. Shahzad et al., "Building a fault tolerant application using the GASPI communication layer," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2015, pp. 580–587.
[64] V. Bartsch et al., "GASPI/GPI in-memory checkpointing library," in *Proc. Int. Conf. Eur. Conf. Parallel Process.*, F. F. Rivera et al. Eds. Springer, 2017, pp. 497–508.

[65] D. Grünewald et al., "FRTM—A productive framework for reverse time migration," in *Proc. EAGE Workshop High Perform. Comput. Upstream*, Amsterdam, The Netherlands, Sep. 2014, pp. 44–48, doi: 10.3997/2214-4609.20141914.

[66] D. Merten and F.-J. Pfreundt, "ALOMA, an auto-parallelization tool for seismic processing," in *Proc. 79th EAGE Conf. Exhib. Workshops*, 2017, pp. 399–403.

[67] J. Böhm et al., "Towards massively parallel computations in algebraic geometry," *Found. Comput. Math.*, vol. 21, pp. 1–40, 2020.

[68] N. Weber et al., "Fed-DART and FACT: A solution for federated learning in a production environment," 2022, *arXiv:2205.11267*.

[69] K. Dolag et al., "Visualizing 10^{11} particles from cosmological simulations," *GCS Inside*, vol. 13, no. 2, pp. 29–31, 2015.

[70] Barcelona Supercomputing Center, "OmpSs-2 specification," Barcelona Supercomputing Center, 2021. [Online]. Available: https://pm.bsc.es/ftp/ompss-2/doc/spec/

[71] J. Aguilar Mena et al., "Transparent load balancing of MPI programs using OmpSs-2@Cluster and DLB," in *Proc. 51st Int. Conf. Parallel Process.*, 2022, Art. no. 55.

[72] J. Aguilar Mena, "Methodology for malleable applications on distributed memory systems," PhD dissertation, Dept. Comput. Architecture, Universitat Politècnica de Catalunya, Nov. 2022.

[73] F. Lordan et al., "ServiceSs: An interoperable programming framework for the Cloud," *J. Grid Comput.*, vol. 12, no. 1, pp. 67–91, Mar. 2014. [Online]. Available: https://digital.csic.es/handle/10261/132141

[74] J. Ejarque et al., "The BioExcel methodology for developing dynamic, scalable, reliable and portable computational biomolecular workflows," in *Proc. 18th Int. Conf. eScience*, 2022, pp. 357–366.

[75] C. Clauss et al., "Dynamic process management with allocation-internal co-scheduling towards interactive supercomputing," in *Proc. 1st COSH Workshop Co-Scheduling HPC Appl.*, Prague, 2016, Art. no. 13.

[76] C. Clauss et al., "Allocation-internal co-scheduling – Interaction and orchestration of multiple concurrent MPI sessions," in *Co-Scheduling of HPC Applications*, vol. 28. Amsterdam, The Netherlands: IOS Press, 2017, pp. 46–68. [Online]. Available: http://ebooks.iospress.nl/volume/co-scheduling-of-hpc-applications

[77] ParTec AG, "ParaStation management," ParTec AG, 2023. [Online]. Available: https://github.com/ParaStation/psmgmt

[78] OpenPMIx Developers, "PMIx reference runtime environment (PRRTE)," OpenPMIx Developers, 2023. [Online]. Available: https://github.com/openpmix/prrte

[79] T. Özden et al., "ElastiSim: A batch-system simulator for malleable workloads," in *Proc. 51st Int. Conf. Parallel Process.*, 2023, Art. no. 40.

[80] X. Zhang, K. Liu, K. Davis, and S. Jiang, "iBridge: Improving unaligned parallel file access with solid-state drives," in *Proc. 27th IEEE Int. Symp. Parallel Distrib. Process.*, 2013, pp. 381–392.

[81] D. Koo et al., "Adaptive hybrid storage systems leveraging SSDs and HDDs in HPC cloud environments," *Cluster Comput.*, vol. 20, no. 3, pp. 2119–2131, 2017.

[82] D. Abramson et al., "A BeeGFS-based caching file system for data-intensive parallel computing," in *Proc. 6th Asian Conf. Supercomputing Front.*, 2020, pp. 3–22.

[83] N. Liu et al., "On the role of burst buffers in leadership-class storage systems," in *Proc. IEEE 28th Symp. Mass Storage Syst. Technol.*, 2012, pp. 1–11.

[84] S. He et al., "S4D-Cache: Smart selective SSD cache for parallel I/O systems," in *Proc. IEEE 34th Int. Conf. Distrib. Comput. Syst.*, 2014, pp. 514–523.

[85] D. Zhao, K. Qiao, and I. Raicu, "HyCache+: Towards scalable high-performance caching middleware for parallel file systems," in *Proc. 14th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2014, pp. 267–276.

[86] G. Congiu, S. Narasimhamurthy, T. Süß, and A. Brinkmann, "Improving collective I/O performance using non-volatile memory devices," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2016, pp. 120–129.

[87] Y. Qian et al., "LPCC: Hierarchical persistent client caching for lustre," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2019, Art. no. 88.

[88] J. F. Lofstead et al., "Managing variability in the IO performance of petascale storage systems," in *Proc. Conf. High Perform. Comput. Netw. Storage Anal.*, 2010, pp. 1–12.

[89] B. Xie et al., "Characterizing output bottlenecks in a supercomputer," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, pp. 1–11.

[90] C. Kuo, A. Shah, A. Nomura, S. Matsuoka, and F. Wolf, "How file access patterns influence interference among cluster applications," in *Proc. IEEE Int. Conf. Cluster Comput.*, Madrid, Spain, 2014, pp. 185–193.

[91] D. E. Singh and J. Carretero, "Combining malleability and I/O control mechanisms to enhance the execution of multiple applications," *J. Syst. Softw.*, vol. 148, pp. 21–36, 2019.

[92] A. Gulati et al., "pClock: An arrival curve based approach for QoS guarantees in shared storage systems," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Model. Comput. Syst.*, 2007, pp. 13–24.

[93] X. Zhang et al., "QoS support for end users of I/O-intensive applications using shared storage systems," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, pp. 18:1–18:12.

[94] Y. Qian et al., "A configurable rule based classful token bucket filter network request scheduler for the lustre file system," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2017, Art. no. 6.

[95] A. Brinkmann et al., "Ad hoc file systems for high-performance computing," *J. Comput. Sci. Technol.*, vol. 35, no. 1, pp. 4–26, 2020.

[96] M. Vef et al., "GekkoFS—A temporary distributed file system for HPC applications," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2018, pp. 319–324.

[97] T. Wang et al., "An ephemeral burst-buffer file system for scientific applications," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2016, pp. 807–818.

[98] F. J. Rodrigo Duro et al., "Evaluating data caching techniques in DMCF workflows using hercules," in *Proc. 2nd Int. Workshop Sustain. Ultrascale Comput. Syst.*, Spain, 2015, pp. 95–106.

[99] F. Garcia-Carballeira et al., "The design of the expand parallel file system," *Int. J. High Perform. Comput. Appl.*, vol. 17, no. 1, pp. 21–37, 2003.

[100] Lawrence Livermore National Laboratory, "UnifyFS," Lawrence Livermore National Laboratory, 2019. [Online]. Available: https://github.com/LLNL/UnifyFS

[101] BeeGFS, "BeeOND: BeeGFS on demand," BeeGFS Wiki, 2018. [Online]. Available: https://www.beegfs.io/wiki/BeeOND

[102] O. Tatebe et al., "CHFS: Parallel consistent hashing file system for node-local persistent memory," in *Proc. Int. Conf. High Perform. Comput. Asia-Pacific Region*, 2022, pp. 115–124.

[103] J. Garcia-Blas et al., "IMSS: In-memory storage system for data intensive applications," in *Proc. Int. Conf. High Perform. Comput.*, Cham, 2022, pp. 190–205.

[104] P. H. Lensing et al., "Direct lookup and hash-based metadata placement for local file systems," in *Proc. 6th Annu. Int. Syst. Storage Conf.*, 2013, pp. 5:1–5:11.

[105] C. Wang et al., "File system semantics requirements of HPC applications," in *Proc. 30th Int. Symp. High Perform. Parallel Distrib. Comput.*, E. Laure et al. Eds. 2021, pp. 19–30.

[106] A. Miranda et al., "Reliable and randomized data distribution strategies for large scale storage systems," in *Proc. 18th Int. Conf. High Perform. Comput.*, 2011, pp. 1–10.

[107] M. Abadi et al., "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, K. Keeton and T. Roscoe, Eds. Savannah, GA, USA, 2016, pp. 265–283.

[108] H. Jasak et al., "OpenFOAM: A C++ library for complex physics simulations," in *Proc. Int. Workshop Coupled Methods Numer. Dyn.*, Dubrovnik, Croatia, 2007, pp. 1–20.

[109] H. Abbasi et al., "DataStager: Scalable data staging services for petascale applications," in *Proc. 18th ACM Int. Symp. High Perform. Distrib. Comput.*, 2009, pp. 39–48.

[110] B. Dong et al., "Data elevator: Low-contention data movement in hierarchical storage system," in *Proc. IEEE 23rd Int. Conf. High Perform. Comput.*, 2016, pp. 152–161.

[111] P. Subedi et al., "Stacker: An autonomic data movement engine for extreme-scale data staging-based in-situ workflows," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2018, pp. 73:1–73:11.

[112] A. Miranda, A. Jackson, T. Tocci, I. Panourgias, and R. Nou, "NORNS: Extending Slurm to support data-driven workflows through asynchronous data staging," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2019, pp. 1–12.

[113] U. Saroliya, E. Arima, D. Liu, and M. Schulz, "Hierarchical resource partitioning on modern GPUs: A reinforcement learning approach," in *Proc. IEEE Int. Conf. Cluster Comput.*, Santa Fe, NM, USA, 2023, pp. 185–196.

[114] Charm++ Developers, "Charm++ documentation," 2023. [Online]. Available: https://charm.readthedocs.io/en/latest/charm++/manual.html