Evolution of HPC Software Development and Accompanying Changes in Performance Tools

Area Exam Report

Srinivasan Ramesh*
*University of Oregon {sramesh}@cs.uoregon.edu

***Abstract*—High-Performance Computing (HPC) software is rapidly evolving to support a wide variety of heterogeneous applications. Traditionally, HPC applications are built using the message-passing interface (MPI) and operate in a bulk-synchronous manner. The complexity of scientific software development and emerging classes of workloads have driven the HPC community to adopt increasingly modular software development frameworks. On the one hand, the modularization of HPC software makes programming these systems more manageable. On the other hand, HPC performance tools have had to be constantly updated to keep up with how HPC software is built and deployed. This paper presents an overview of the fundamental driving forces and technologies that have resulted in the evolution of HPC software development over the last three decades. The techniques that performance tools have implemented to keep up with these changes are also discussed. Finally, this paper presents some avenues for future work, highlighting the critical areas that performance tools must address to remain relevant.**

***Index Terms*—modules, components, services, coupled applications, data management**

I. Introduction

Over the past three decades, there has been a constant evolution in how HPC distributed software is conceptualized, implemented, and deployed. Traditional HPC software development has been centered around the message-passing programming model. In particular, the message-passing interface (MPI) has been the de-facto programming model of choice for developing distributed HPC applications. In response to the recent explosion of data-centric and machine learning (ML) workloads in scientific computing [1], HPC systems and software are rapidly evolving to meet the demands of diversified applications. These new applications do not fit into the MPI programming model [1], [2], thus necessitating a change in the fundamental methodologies for distributed HPC software development. In particular, the emergence of coupled applications, ensembles, and in-situ software services running alongside traditional HPC simulations [3] are the key indicators of such change. Scientific workflows are beginning to move away from traditional MPI monoliths to resemble a mix of several different pieces of specialized distributed software working in concert to achieve some larger goal [4].

Within a process running inside the broader distributed application, increasing software complexity and the need to perform ever-more-realistic simulations have been the driving forces behind the componentization of HPC software [5]. Parallels can be drawn between adopting componentization in the industry [6] and the subsequent push to componentize HPC software to manage complexity. At the same time, HPC performance tools have also been updated to reflect this change [7]. Over the last 20 years, the push to componentize software has resulted in evolving a service-oriented architecture in the industry. The HPC community has recently been actively looking into similar software architectures to support heterogeneous, data-centric workloads.

The key aspect that sets HPC applications from other forms of distributed software is the need to achieve high performance, high efficiency, and a high degree of scalability on exotic HPC hardware. In such environments, performance measurement and analysis tools play a critical role in identifying sources of performance inefficiencies. State-of-the-art HPC performance tools such as TAU, HPCToolkit, and Caliper [8]–[10] excel at the performance analysis of monolithic MPI applications. However, when faced with the task of holistically analyzing the performance of coupled multi-physics codes or distributed HPC data services, applying these performance tools without change finds limited application because these tools implicitly rely on the existence of an MPI library to bootstrap their measurement frameworks. Studying the evolution of HPC performance tools in this context is necessary to identify opportunities and future tool design requirements.

This area exam explores the evolution of HPC software and performance tool development. Starting with a collection of source files built into one monolithic MPI executable, HPC software has evolved to support coupled applications, distributed data services, in-situ ML, visualization, and analysis modules running together on a single machine allocation. A novel narrative of the tension between the need to manage software complexity while simultaneously achieving high performance is presented. Wherever appropriate, notable trends from the general computing industry are cited as key technology enablers of such change. The parallel timeline and evolution of performance measurement, analysis, and online monitoring tools and techniques are also presented in this research document.

II. Background

To familiarize the reader with standard HPC programming practices and common terminology, a brief overview of the state-of-the-art in HPC system architectures, applications, and performance analysis software is necessary. HPC machines,

also known as *supercomputers*, represent the largest networked computers designated for scientific computing. Although official figures of the cost of such machines are rarely released, speculations [11] suggest that the hardware cost alone is several hundreds of millions of dollars. Also, the operating costs of running these machines are in the order of tens of millions of dollars. Thus, the applications that run on these machines must do so at the highest efficiency possible to maximize scientific output, maximize machine occupancy, and minimize cost. Today's typical HPC machine architecture consists of a heterogeneous mix of general-purpose CPUs and accelerator architectures such as the graphics processing unit (GPU) [12]. These computing elements are connected through a high-bandwidth, low-latency interconnect such as Infiniband [13]. Further, all the computing and networking elements are typically situated within the same IT infrastructure or building. These key characteristics separate HPC architectures from more general distributed grid computing architectures.

A. MPI: The Dominant Distributed Programming Model

The MPI programming model [14] has dominated the HPC software development landscape for a large portion of the past three decades. An MPI application is launched as a set of *N* communicating processes. Traditional MPI applications [15], [16] divide an application domain, such as a computational grid into several logical sub-domains. Each MPI process is assigned one or more sub-domains on which they perform local computation. When necessary, these MPI processes communicate to exchange or aggregate intermediate results. This communication can either be point-to-point or collective. A typical scientific application [15] contains a discretized time domain and a computational grid (spatial domain) and runs for a certain number of fixed timesteps. Communication and computation proceed in phases within a timestep, with periodic synchronization between the different processes. Such a model of parallel computation is referred to as the bulk-synchronous parallel model (BSP).

Given the importance of MPI, there have been several large-scale, ongoing efforts to implement high-performance MPI implementations that are portable as well [17], [18]. Communication requires processes to synchronize with each other. Besides, communication over the network can significantly slow down a parallel program. At a large scale, synchronous, collective communication can degrade the application's overall performance and quickly limit the application's scalability. Therefore, several HPC performance engineering efforts have been centered around improving MPI library communication performance.

B. Shared-Memory Programming Models

Multi-core and many-core CPU architectures such as the Intel Xeon Phi [19] and accelerator architectures such as the GPU have become commonplace on leadership-class HPC systems [12]. HPC applications have evolved to support and extract performance from the increased on-node parallelism. Specifically, shared-memory parallel programming has been a key focus area for performance optimizations. Notable programming models offering shared-memory parallel programming capabilities include OpenMP [20], TBB [21], pthreads [22], and OpenACC [23]. NVIDIA CUDA [24] is arguably the most popular GPU programming model, followed by OpenCL [25].

These shared-memory programming models expose their functionality either through a library-based API or through compiler *pragmas* or hints to aid with the automatic identification and generation of parallel code. Invariably, shared-memory parallel programming involves the generation of *parallel threads* of execution. These threads share the same process address space, may have their local stacks, and communicate through shared memory regions.

C. Other Programming Models

While many conventional HPC applications employ a distributed model such as MPI combined with a shared-memory model such as OpenMP, other applications employ hybrid programming models and runtimes. Notable examples include Charm++ [26], a machine-agnostic task-based programming approach, and partitioned global address space (PGAS) programming models such as UPC [27] and Chapel [28].

D. Performance Analysis Tools

This section discusses the state-of-art in the performance analysis of traditional HPC applications.

1) MPI Performance Analysis: Performance tools for HPC applications have primarily catered to those applications that employ the MPI programming model. Typically, the parallel profiling and tracing tools build on the presence of an MPI library to bootstrap their measurement frameworks [8]–[10]. The PMPI-based library interposition technique has successfully enabled performance tools to intercept MPI calls to perform timing measurements and capture other relevant performance data such as message sizes. The PMPI approach is often the first step in analyzing the performance of MPI applications. Key performance metrics include the sizes of MPI messages, contributions of MPI collective routines, and the contributions of MPI synchronization operations to the overall execution time.

2) Shared-Memory Performance Analysis: Regarding the capture of application-level performance information (function-level timers), HPC performance tools follow one of two schools of thought. Instrumentation-based tools [8], [10] rely on intrusive instrumentation to elicit the exact measurements of events. Tools based on statistical sampling [8], [9] rely on lightweight sampling and call stack unwinding to capture statistical features of the performance data. Hardware counters are commonly used to track the efficiency of various routines based on their hardware resource usage characteristics. The performance API (PAPI) [29] has grown into a standard and portable way of exposing hardware performance data.

Shared-memory programming libraries expose their profiling APIs to allow insight into their operation and performance.

Notably, the OMPT interface [30] allows performance tools to register callbacks for several events defined by the OpenMP specification. Likewise, the CUDA CUPTI API allows insight into the operation of the CUDA API. After the node-level performance data from various sources is collected, performance tools typically orient and aggregate this data around the MPI processes involved in the particular execution instance.

III. Definitions

This section defines the various terms that are used in the sections that follow. Unless specified otherwise, any usage of these terms pertains to the following definitions.

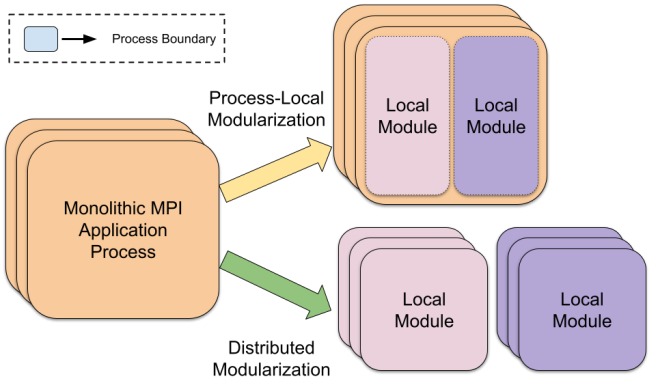

Fig. 1: Modularization: A Conceptual Illustration

A. Module

A *module* is *any* piece of software or code entity with well-defined boundaries used as a general building block for higher-level functionality. A module can be a library, a class object, a file, a service, or a component. Throughout this document, the term "module" is used in the broadest context possible, i.e., it does not refer to any specific software, implementation, technique, or specification. Therefore, it follows that modularization is the process by which a piece of software is divided into separate, independent entities by following the general software design principle of "separation of concerns". *Process-local modularization* results in software modules that run within the same address space (process). *Distributed modularization* results in software modules that are separated by different address spaces (processes). Figure 1 depicts a conceptual illustration of modularization of a monolithic MPI-based application.

B. Component

The term *component* is interchangeably used with the term module everywhere *except* in Section V. In Section V, the term "component" has a special meaning and refers to modules that adhere to a particular type of component architecture and interface specification. Componentization, as used in Section V, is converting a piece of software into components that follow the component architecture.

C. Service

Services are regarded as loosely connected software modules running on *separate* processes with well-defined public interfaces specifying access to the service functionality. Usually (but not always), the interaction between two service entities involves calls over the network. Again, every service is a module (or a component), but not every module is a service.

D. Composition

Composition is the process by which two or more modules are connected (coupled) to form a larger entity that functions as a whole. By this definition, the composed modules can be within the same process, within different processes on the same computing node, or inside processes running on separate computing nodes. Composition is a natural outcome of the modularization of software.

IV. Application Complexity and Modularization

This section presents the fundamental claim regarding the evolution of HPC software along with the reasoning supporting this claim.

A. Claim

HPC software development approaches have become increasingly modular to manage software complexity. As a direct consequence of this modularization, performance analysis tools have had to reinvent themselves to stay relevant and practical.

B. Reasoning

By modularizing software, the complexity of software is compartmentalized [5], and modules become re-usable. Individual teams or developers can focus on building just a few specialized modules with clearly defined interfaces instead of having to deal with a massive, complicated codebase representing the entire application. Individual components can be portable by having multiple "backend" implementations. Componentization can also enable fine-grained resource allocation and management. Besides, modularization is attractive because it allows for the rapid composition of modules to create many composed applications targeting different usage scenarios. The complexity in developing HPC software primarily arises from the following sources.

1) Simulation Scale and Fidelity: There are two challenges to programming at large scale. First, parallel programming is inherently hard to get right. There are several classes of software bugs related to "program correctness" that show up only on highly concurrent systems. The one constant in high-end computing has been the need to simulate natural systems with ever-increasing fidelity and at a larger scale. A study of the largest HPC systems globally over the past 25 years [12] supports this claim. Second, with a billion-way parallelism available on modern machines, the software must be constantly re-written and updated to reflect and utilize new sources of parallelism. An analysis of popular large-scale applications such as CESM [31], LAMMPS [32], HACC [33] reveals that

each of these applications has consistently been updated with additional modules to simulate individual physical phenomena with increased fidelity.

2) Range of Applications and Platforms: In the past decade, the range and diversity of applications requiring high-end computing capabilities have exploded. The US Department of Energy develops and publishes *mini-applications* called CORAL benchmarks that represent the core computations within applications of national interest. Performance optimization of these benchmarks would likely result in an improvement in the performance of the larger scientific applications they represent. An analysis of the CORAL-1 benchmark suite [34] released in 2014 and the CORAL-2 benchmark suite [35] released in 2020 reveals a telling story. In 6 years, data science and machine learning (ML) workloads have become Tier-1 applications. These data science workloads are *not* regular MPI applications. This explosion in application variety has resulted in the search for a broader set of programming models and supporting services to accommodate the newer applications.

3) Structure of Modern Scientific Research Teams: Due to the number of different components involved in modern scientific software development, it has become impossible for one person or team to develop all the software components [5], [36].

With an increase in the number of interacting components or modules that must simultaneously run at high efficiency, performance data exchange with analysis and monitoring tools has become more complex. When modularization results in black-box software components, it is challenging for performance tools to instrument and extract the necessary performance data. There is tension between the need to manage software complexity and the simultaneous requirement of running software components at their optimal efficiency. The rest of this paper attempts to present evidence in support of this reasoning.

V. Process-Local Modularization

This section presents an overview of the development of component-based HPC software. An in-depth account of the techniques implemented by HPC performance tools to analyze component-based software is also discussed.

A. Important Trends in the Computing Industry

Arguably, the need for designing modular software can be traced to the popularization of object-oriented methodologies in the 1970s and early 1980s [37]. Software complexity had exploded, and the computing industry was beginning to realize the importance of *software architectural patterns* as a way to implement and manage software.

Eventually, the focus on separating concerns led to the development of *component-based software engineering* (CBSE) [38]. CBSE aimed to go beyond object or module re-use by defining *components* as "executable units of independent production, acquisition, and deployment that can be composed into a functioning whole". The fundamental notions of independent deployment, re-usability, and composition are the recurring themes underlying major revolutions in software architecture over the past two decades.

In the early 1990s, the industry began to adopt component-based software engineering frameworks such as the Common Object Request Broker Architecture (CORBA) [39], the Java Remote Method Invocation (RMI), and the Component Object Model (COM) [40]. Component models aimed to address the shortcomings of object-oriented methodologies. Specifically, components were designed to be modular, re-usable, and language-independent, allowing for their rapid composition to build higher-level functionality.

B. Component Software For HPC

In the late 1990s, the HPC research community had realized that the ballooning scientific software complexity had to be controlled and managed. Scientific software developers were beginning to develop coupled, multi-physics models for plasma simulations, and nuclear fusion codes. There was a need to employ high-performance, re-usable, plug-and-play components that were language-agnostic.

The Common Component Architecture (CCA) [41]–[45] sought to address several shortcomings of object-oriented methodologies, libraries, and commercial component frameworks [39], [40]. First, while object-oriented frameworks had done well to encourage software reuse within a project, they offered little to no cross-project reuse. Second, object-oriented frameworks were limited in their ability to form the basis of component software as they were applicable only in compile-time coupling scenarios. HPC component frameworks required that components expose a way for compatible implementations to be "swapped" at runtime. Third, component frameworks enabled language independence by relying on meta-language interfaces, a feature that was not typically available with object-oriented frameworks at the time. Fourth, the main issue with employing commercial component frameworks in HPC was the exorbitant performance overheads for "local" inter-component interactions. Lastly, multiple component implementations with the same interface could co-exist within a framework. Doing so was not possible with libraries. After initial attempts to develop independent, disparate component frameworks, the HPC community came together to form the CCA Forum, mainly consisting of members from various academic institutions and US Department of Energy laboratories.

1) CCA Model: The objectives of the CCA specification are found in [41]. Specifically, the CCA was designed within the context of single-program-multiple-data (SPMD) or multiple-program-multiple-data (MPMD) codes. The CCA-MPI marriage was destined given the popularity of the MPI programming model and the integration objectives of the CCA specification.

The following list defines some of the critical elements of the CCA model.

- Components: In the CCA model, a component is an encapsulated piece of software that exposes a well-defined public interface to its internal functionality. Notably, this

definition allows components to be composed together to form more complex software.

- Local and Remote Components: Components that live within the same address space are local components. Interactions among local components are ideally no more expensive than regular function calls. A process boundary separates remote components. Although the CCA specification supported remote component interactions, it implicitly incentivized components to perform a bulk of the interactions locally, if possible.
- Scientific Interface Definition Language (SIDL): The CCA specification introduced the scientific interface definition language (SIDL) as a means of enabling composition and interaction amongst components written in different languages. Given the prevalence of "legacy" HPC codes written in C and Fortran and the growing use of Python for scripting and analysis tasks, language interoperability was an essential CCA requirement. The SIDL is a meta-language that is used to describe component interfaces. Notably, it supported complex data types, a feature that commercial component frameworks did not support at that time. Other tools such as Babel [46] read in the SIDL specification to generate glue code allowing components written in different languages to interact.
- Frameworks: Frameworks are the software that manage component interactions. They are responsible for connecting components through the use of ports. The notion of a framework implies a certain level of orchestration necessary for the functioning of CCA components.
- CCA Ports: The CCA specification described two types of ports — *provides* and *uses* ports. A component allows access to its functionality through the provides port, and it registers its intent to interact with other components through the uses ports. The framework is ultimately responsible for actuating the interaction by connecting the provides and uses ports.
- CCA Services: Every CCA-compliant framework provides the registered components with a set of essential services. The components access these framework services similar to how they interact with other components — through ports. One of the most important functionalities of the services object is to provide methods for the components to register their uses and provides ports.
- CCA Repository: The CCA specification included a public CCA repository for components. The key idea was to enable the rapid, "plug-and-play" design of scientific applications using off-the-shelf components available in the CCA repository. The other motivation behind providing a repository was to encourage large-scale community reuse of software components and widespread adoption of the CCA specification.
- Cohort: A collection of components of the same type (running within different address spaces) is referred to as a cohort.
- Direct-Connected Framework: As Figure 2 depicts, there are two ways in which CCA components can be composed. In a direct-connected framework, each process consists of the same set of parallel components. A notable feature of such a framework is that it does not allow *diagonal* interactions among components, i.e., inter-component interactions are limited to function calls within a process. Direct-connected frameworks only support the SPMD model of parallelism. Parallel components within a cohort can interact through any distributed communication library available. This latter type of communication was outside the scope of the CCA specification.
- Distributed Framework: Distributed frameworks, depicted by Figure 2, allow diagonal interactions and a more general MPMD model of parallelism. Specifically, inter-component interaction can occur between components belonging to different processes. In addition to providing a remote-method-invocation (RMI) interface, distributed frameworks need to address data distribution between coupled applications. This problem shall be revisited in Section VI.

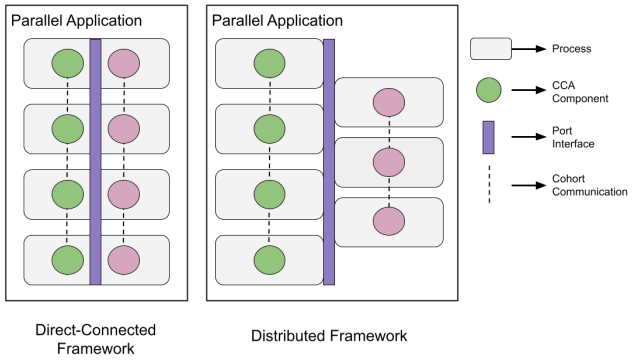

Fig. 2: CCA: Framework Types (inspired from DCA [47])

2) CCA: Performance Measurement: Given the composition model of CCA applications, it was imperative to generate a performance model of the component assembly and judge the efficacy of the instantiation [48], [49]. Specifically, the community found it necessary to measure local component performance and inter-component interactions in a non-intrusive, cohesive way. These measurements would then be used to generate performance models of individual components and their interactions. Finally, the generated performance models would be employed to select an optimal set of components for the particular application context. The research on performance measurement and analysis of high-performance CCA applications can be divided into two categories: (1) Intra-component performance measurement and (2) Inter-component performance measurement.

Intra-component performance measurement entails capturing performance data about the execution of functions and routines within a component. Literature [7] presents two ways in which intra-component interactions can be captured:

- Direct Instrumentation: This is the most straightforward way to extract measurement data. Existing HPC performance tools could instrument component routines directly

(either manually or through automatic instrumentation). The component invokes the measurement library to generate the necessary performance data.
- Performance Component: Direct instrumentation techniques suffer from the disadvantage of being tightly coupled with component implementations. The use of an abstract measurement component interface and a performance component that implements the measurement interface circumvents this problem of tight-coupling, and it allows for a more flexible approach to performance measurement of intra-component interactions. Specifically, any compliant performance tool (TAU being just one example) can implement the performance component interface. Moreover, the use of an abstract measurement interface ensures that the overheads of performance instrumentation are effectively zero when no performance component is connected.

Inter-component performance measurement is necessary to study the interactions between components. Specifically, components are connected via provides and uses ports. The interactions between components occur on these ports and contain valuable information such as data transfer sizes and source and destination identifiers used in message-passing routines. These interactions are not visible to an external entity, and thus, unique instrumentation is required to capture them.

The instrumentation and measurement techniques used for inter-component performance analysis can be categorized as either (1) direct instrumentation, (2) instrumentation during interface definition and generation, or (3) Proxy-based instrumentation and measurement.

- Direct Instrumentation: A few CCA frameworks, such as CCAFFEINE [50], are based entirely on C++ as the language for implementing component interfaces. In such scenarios (often not the case), direct instrumentation techniques can measure and observe inter-component interactions.
- Instrumentation of Interface Generation Code: When component interfaces are specified using an interface definition language such as SIDL [51], direct instrumentation can be applied only once the interface language compiler (such as Babel) has generated the language-specific component interface glue code. Another approach is to build the instrumentation directly into the process of generating the glue code. Both approaches are feasible. However, the latter technique is likely to yield more optimal code [7].
- Proxy-based Instrumentation and Measurement: Arguably, the most popular technique to instrument and measure component interactions involves component proxies to snoop for invoked methods on the provides and uses ports [48], [52]. Component proxies are stub component implementations presenting the same interface as the components they represent. Component proxies placed "in-front" of "caller" components trap method calls on the uses ports to enable performance measurement. A "Mastermind" component invokes the measurement API of a backend performance component (such as the TAU component) and is responsible for storing and exposing the performance data for external analysis and query.

3) CCA: Performance Monitoring and Optimization: Aside from managing software complexity, component frameworks also present *logical* boundaries for performance optimization. Recall that, unlike standard libraries, the CCA component specification allowed multiple component implementations presenting the same interface to coexist within the application. In a CCA-enabled application, a sub-optimal component can be dynamically replaced with a more optimal component. For example, in a scientific application composed of *solver* components performing a linear-algebra calculation, the solver component implementation can be switched at runtime depending on how well the component performs on traditional metrics such as execution time as well as *functional* metrics such as solver residual.

Literature [5], [53], [54] describes *computational quality of service* (CQoS) as a general methodology for optimizing CCA-enabled application. CQoS is the "automatic selection and configuration of components to suit a particular computational need". Essentially, the selection of an optimal set of components involves a trade-off between accuracy, performance, stability, and efficiency [54]. The cycle of performance measurement and optimization of CCA applications has four distinct parts: (1) performance measurement, (2) performance analysis, (3) performance model generation, and (4) a control system to implement optimizations.

Performance measurement has been discussed in Section V-B2. Thus, here we discuss performance analysis, performance model generation, and control systems for CCA optimization. The application is instrumented to report traditional performance metrics such as the execution time and component-specific *functional* performance metrics such as the solver residual. This performance information is used to train analytical and empirical models of component performance. Specifically, the performance information is written to a performance database component [55]. The analysis tools query the database component to generate *component performance models*. These component models are written into a "substitution assertion database" that acts as the link between the analysis and control infrastructure.

The control system is driven by *control laws* that dictate the actions of the control infrastructure. Control laws are essentially the "rules" that drive dynamic adaptation of CCA components. A control law executes by combining the application's state information with the appropriate model information within the substitution database to output a recommendation for optimization. The control infrastructure is ultimately responsible for implementing the recommendation.

The control infrastructure consists of the *reparameterization decision service* and the *replacement service* as the critical pieces. CQoS control is accessed seamlessly via proxy components. The use of a proxy allows applications to benefit from CQoS with minimal addition of intrusive instrumentation.

Further, CQoS can be dynamically turned on or off. When CQoS is disabled, proxy components function as gateways to CCA performance measurement. When CQoS is enabled, the proxy component is connected to the optimization components that inform the proxy of the optimal provides port to use (among many candidate component ports). Effectively, the proxy component functions as a switch that connects the caller (application component) with the optimal implementation of the callee component.

C. Other HPC Component Frameworks

Although the CCA specification formed the majority of efforts to componentize HPC software, there were other similar projects that had related goals.

1) High-Performance Grid Component Frameworks: Component-based grid scientific computing infrastructures explore methodologies for optimization [56] that share similarities with CCA. First, like CCA, grid component compositions are indicated in the component metadata. CXML is a markup-based composition specification that is similar to the SIDL language used in CCA frameworks. Such a specification enables the component framework to generate and analyze static call graphs. Second, the "application mapper" is an optimization component that functions as the control system within the framework. The application mapper takes the abstract component composition (known as an application description document) and generates a runtime representation of the composition. It takes system resource metadata as input from the grid deployment services and combines the existing component performance models to form an optimal *execution plan*. If there is a change in grid resources, the grid application can contact the application mapper at runtime to generate a new execution plan.

Aside from the hardware on which they operate, there are two crucial differences between high-performance grid component frameworks and traditional high-performance computing frameworks such as CCA. First, although both frameworks operate on distributed systems, inter-component interactions in grid frameworks typically involve the network. In direct-connected CCA frameworks, most inter-component interactions are reduced to a sequence of regular function calls. As a result, grid component frameworks are designed more like distributed "services", and traditional HPC frameworks operate more like libraries. Second, grid component frameworks assume that the component repository contains performance model metadata that allows the application mapper to make reconfiguration decisions. In other words, *dynamic* component reconfiguration is treated as a first-class design requirement, given that the fluctuation in available resources is a common occurrence. The CCA specification as such does not pay special attention to ensuring the dynamic reconfiguration of components. Instead, CCA treats dynamic component reconfiguration as an activity to be performed on a per-application need basis. Despite this, the HPC community has invented clever ways of seamlessly and incrementally integrating control capabilities through the use of proxy components.

2) Low-Level Component Framework (L2C): The L2C [57] attempts to address the issue of *performance portability* on HPC systems. The authors observe that multi-platform support for large applications is usually achieved through means that offer little code reuse (conditional compilation, component software, runtime switches). They attempt to resolve this problem through a low-level component model that is (1) composable, (2) offers portable performance, and (3) offers a high degree of code reuse. Components are implemented as *annotated objects* with well-defined entry points, resembling a plugin architecture.

The components are written in C++, Fortran, or Charm++. Multiple instances of a particular component type can co-exist within the application. The composition is specified through an L2C assembly descriptor file. A small L2C runtime is responsible for managing the component interactions. Notably, there is no support for multi-language component compositions (and associated glue code generation). The key idea lies in breaking down the application into several fine-grained components. Doing so allows for a high degree of code reuse between performance-portable implementations designed for different platforms. This design choice also leads to an explosion in the number of components required to implement even a relatively simple application such as a Jacobi solver.

3) directMOD Component Framework: HPC applications such as adaptive mesh refinement (AMR) codes form a particular category of applications whose structure (data and communication distribution) changes over time. As the application structure informs the component assembly, AMR applications require a component framework that supports dynamic reconfiguration. Not only this, multiple dynamic reconfigurations co-occur inside different application processes, introducing synchronization and consistency issues among them. The directMOD [58] component framework addresses these problems by offering a component model that introduces two new concepts: domains and transformations. Domains are components that lock specific portions of the application and ensure safety. Transformations are ports that connect a transformation to its target sub-assembly. However, directMOD is not broadly applicable to any general application.

D. Comparing Component Frameworks

A comparison of the various component frameworks is presented in Table I. Several of these comply with the CCA architecture. CCAFFEINE [50] was intended to be a model implementation of the CCA specification. Notably, it is the only major HPC-optimized CCA implementation that does not support the MPMD model. In other words, all CCAFFEINE-enabled applications are limited to SPMD parallelism, and peer components interact through direct connections only. Some component frameworks such as MOCCA [59], VGE-CCA [60], CCAT [61], LegionCCA [62], and XCAT3 [63] are not HPC-optimized. These frameworks are geared primarily to operate in grid environments, and thus, the distributed communication libraries they employ are not HPC-aware.

Among the HPC-optimized frameworks, SCIrun2 [64] and Uintah [65] are the only CCA frameworks that have some form of in-built performance optimization support for dynamic reconfiguration of their components. However, the performance analysis and optimization techniques discussed in Section V-B2 and Section V-B3 can be generally employed in CCA frameworks that do not have built-in support. SCIRun2 is the only HPC-optimized framework that supports cross-framework compatibility. That is, SCIRun2 is a *meta-framework* that allows interoperability of components adhering to different specifications. For example, SCIRun2 allows the composition of a CCA component with a CORBA component. It is also worth mentioning that most CCA frameworks support some form of data distribution among components that run within a coupled, MPMD-style architecture. A detailed discussion of this support is presented in SectionVI-E1.

E. Scientific Computing Frameworks

Component architectures have helped manage the complexity of HPC software development and increase developer productivity. At the same time, there have been other noteworthy, smaller-scale efforts in this direction. Specifically, scientific computing frameworks such as POOMA [66], PETSc [67], HYPRE [68], Grace [69], and OVERTURE [70] have enabled the rapid development of scientific software from "building blocks". These frameworks are built into libraries and export an interface in either C, C++, or Fortran (the three most commonly used languages to develop HPC software). Except for HYPRE, the building blocks used to compose higher-level functionality are explicitly implemented as objects.

While these frameworks share with component architectures the general principle of composition, they target a different user and application space. Table II enlists the similarities and differences between scientific computing frameworks and the CCA component framework. Most importantly, scientific frameworks are tailored for a specific, narrow domain and are not applicable to build arbitrary HPC applications. HYPRE, for example, is a library of pre-conditioners for use in linear algebra calculations. On the other hand, PETSc is a library designed specifically for matrix operations. Given that scientific frameworks are implemented as libraries, traditional performance analysis techniques such as library interposition, sampling, and compiler instrumentation can be employed directly without any special modifications. Compared to CCA, scientific frameworks typically offer an unmatched speed of development, productivity, and out-of-the-box performance for applications that fit the particular domain supported by the framework. However, scientific frameworks generally offer little to no support for adaptivity. It is generally assumed to be the responsibility of the application developer for fine-tuning the performance of the library on a novel platform.

F. Modularization of MPI Libraries

MPI libraries were among the first to adopt a modular architecture. The rising complexity of MPI library implementations, the need to be portable across HPC platforms, and the scale of development teams were the primary motivating factors behind the push to modularize MPI libraries. Three prominent MPI libraries are discussed, compared, and contrasted based on how they choose to implement modular architectures.

1) LAM/MPI: The LAM/MPI project [71] was the first production-ready MPI implementation to implement a component architecture explicitly. The LAM project was initially structured as an extensive collection of source files and directories. However, it was observed that new developers found it increasingly hard to understand the source code, contribute to the project, and experiment with novel optimization strategies. As a result, the LAM project adopted a component architecture focused on being lightweight, high-performance, and domain-specific, as opposed to more general frameworks such as the CCA [44] architecture.

LAM/MPI supports four types of components — the RPI (Request Progression Interface) component, the COLL (COLLector) component, the CR (Checkpoint-Restart) component, and the BOOT (BOOTstrapping) component. LAM supports multiple implementations of the same component type (through a plugin framework) to coexist within an MPI process. Doing so allows for a dynamic selection of component implementation to optimize runtime behavior. Notably, the re-implementation of LAM using a component architecture improved MPI communication performance by a small margin.

2) OpenMPI: The OpenMPI project [72] is a successor to the LAM/MPI library. The OpenMPI community recognized the need for an MPI library that explicitly supports and encourages third-party developer contributions. When OpenMPI was being developed, algorithms for process control, fault tolerance, checkpoint-restart, and collective communication were beginning to form separate research areas in their own right. Thus, there was a need to support several different versions of these algorithms within a single larger framework.

At the heart of the OpenMPI implementation is a component architecture that aims to resolve both of these challenges. The design element that sets OpenMPI apart from previous implementations is a *multi-level* component architecture. The principal, higher-level component framework ("meta-framework") supports several component frameworks underneath. Each of these lower-level component frameworks targets one specific function, such as collective communication or checkpoint-restart. Further, these lower-level component frameworks manage one or more modules. It is the responsibility of the individual component frameworks to load, discover, and manage the life-cycle of their respective modules. Like LAM/MPI, OpenMPI modules are implemented as plugins. They are integrated into the MPI library statically or as shared libraries, allowing for compile-time or runtime module discovery and initialization.

3) MVAPICH2: MVAPICH2 [18] is a state-of-the-art, high-performance MPI implementation that does not explicitly follow a component architecture. However, due to the same factors described in Section IV-B, the design of the library has become increasingly modular over time. Figure 3 depicts this modularization of MVAPICH2. The current version of the

TABLE I: Comparing Component Frameworks

Property	CCaffeine	DCA	XCAT3	Uintah	SCIRun2	MOCCA	VGE-CCA	LegionCCA	L2C	directMOD	CCAT
CCA Compliant?	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes
Optimized for HPC?	Yes	Yes	No	Yes	Yes	No	No	No	Yes	Yes	No
General Purpose?	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	No	Yes
Distributed Communication Support	N/A	RMI	Multiple	RMI	RMI	Multiple	SOAP-RPC	RMI	MPI	MPI	RMI
Built-In Performance Optimization?	No	No	No	Yes	Yes	No	Yes	No	No	Yes	No
Cross-Framework Compatibility?	No	No	No	No	Yes	Yes	No	No	No	No	No
Cross-Language Support?	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes
Code Reuse	Moderate	Moderate	Moderate	Moderate	Moderate	Moderate	Moderate	Moderate	High	Moderate	Moderate

TABLE II: Comparing Scientific Frameworks and CCA

Property	Scientific Frameworks	CCA
Domain Specific?	Yes	No
Distributed Computing Support?	No	Yes
Cross-language Support?	Partial	Yes
Traditional Performance Tools Applicable?	Yes	No
Support for Performance Portability?	Moderate	High
Speed of Development of Higher-Level Functionality?	High	Moderate
Support for Adaptivity?	Moderate	High

library delineates the same set of *logically separate* modules as OpenMPI — fault tolerance, job startup, and collective algorithms. However, these separate modules are not explicitly managed as independent units. Therefore, the primary method by which MVAPICH2 allows an external user to control its behavior is through the use of environment variables and compile-time configuration flags. Notably, MVAPICH2 does not support custom implementations of these modules, nor does it offer an easy way to replace them at runtime dynamically. Arguably, its monolithic architecture makes it more difficult for external contributors to make changes to the library source code.

G. Tools for Performance Data Exchange

A side-effect of the increasing software complexity and scale is an update in how performance tools instrument the software to measure performance. While modularization can be considered a good engineering practice and a necessity when considering the sizeable cross-institutional nature of HPC software development, modularization hinders the exchange of necessary performance essential information between software layers. At the same time, the use of modular software on large, high-end computing systems has (1) necessitated their dynamic, online adaptation and (2) enabled fine-grained optimization of the various modules and algorithms that comprise the software. Traditionally, HPC performance tools have been passive participants in the optimization of HPC applications. They are primarily employed for offline analysis of performance data. The various costs associated with running applications at exceedingly large scales have motivated the tighter integration of performance tools into the software stack.

1) MPI Tools: There have been various attempts to design and implement techniques that enable closer interaction between a performance tool and the MPI library. PERUSE [73] was an initial attempt at using callbacks to gain access to significant events that occur inside the MPI library. The performance tool installs callbacks into its code for events that it is interested in measuring. When these events occur, the tool callback is invoked, allowing the tool to gather and analyze the pertinent performance data. Ultimately, PERUSE failed to gain traction in the MPI community due to a mismatch between the proposed events and the capabilities of existing MPI implementations.

The MPI Tools Information Interface (MPI_T), introduced as a part of the MPI 3.0 standard, has received significantly more attention from tool developers than previous efforts. MPI_T defines two variable types — performance variables (PVARs) and control variables (CVARs). Tools need to query the MPI_T interface to access the list of PVARs and CVARs that the MPI library wishes to export. PVARs represent counters and resource usage levels within the MPI library, while CVARs represent the "control knobs" that can affect dynamic MPI library reconfiguration. Several tools have been developed [74]–[76] to take advantage of the MPI_T interface to gather performance data, while only one previous work [75] implements a tool architecture that enables the dynamic control of the MPI library through CVARs. MPI libraries such as MVAPICH2 and OpenMPI export a plethora of PVARs to be queried at runtime, but they currently lack support for CVARs that control *online* behavior. As a result, the effective use of MPI_T for dynamic reconfiguration of MPI libraries remains an open problem. More recently, callback-driven event support through MPI_T is once again gaining traction within the MPI community [77].

2) OpenMP Tools: The first notable effort to enable profiling of the OpenMP runtime is the POMP [78] profiling interface. POMP was designed as an OpenMP equivalent of the PMPI interface for MPI applications. POMP allows for seamless, portable profiling of OpenMP sections to gather context information using the OPARI source-to-source instrumentor. However, it can impose noticeable runtime overheads for short-running loops. More importantly, POMP does not allow access to internal OpenMP runtime information and thus has limited application in gathering internal event data. The Sun (Oracle) profiling interface [79] implements a callback-driven model to gain partial access to OpenMP runtime state through the asynchronous sampling of call stacks. However, a lack of support for static executables and gathering of context information resulted in the interface not gaining traction within the community.

Like MPI, these various efforts to enable low-overhead profiling and tracing of OpenMP applications have culminated in developing the OpenMP Tools Interface specification (OMPT) [30]. OMPT is a *standard* that defines how tools

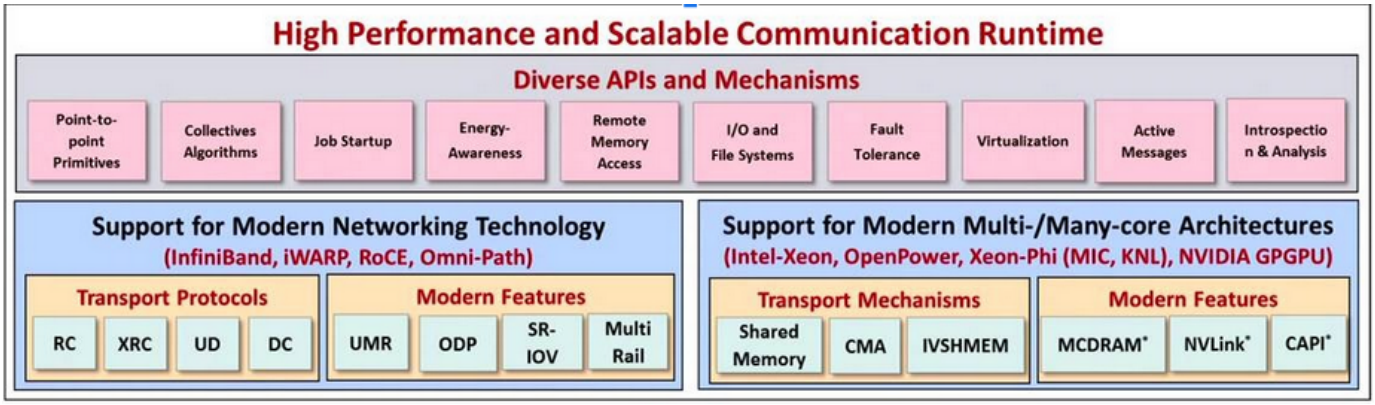

Fig. 3: Modularization of MVAPICH2 (image credits: Dr. D.K. Panda, Network Based Computing Lab, The Ohio State University)

and OpenMP runtimes should interact to enable profiling and tracing of OpenMP applications. It borrows ideas from past efforts to present a callback-driven interface that supports several *mandatory* and *optional* events. Each supported event is associated with a specific data structure provided to the tool for generating context information. Additionally, the OpenMP runtime manages state information on a per-thread basis. Since its introduction into the OpenMP standard, the OMPT interface has grown to support callback-driven profiling of accelerators such as GPUs.

3) PAPI SDE: While the MPI and OpenMP tool interfaces are limited to enabling performance data exchange within their respective domains, the PAPI Software-Defined-Events (SDE) [80] is an attempt to standardize the exchange of software performance counters between *any* two software layers within a process. The PAPI SDE project recognizes that library-specific approaches such as MPI_T, albeit standardized, are not widely applicable directly. Through the existing PAPI API, software modules can export software performance metrics of interest to other libraries or modules running within the process. There are three ways in which SDEs can be created and used. One, a library can declare an internal variable as an SDE to be read directly by other modules. Two, the library can register a callback that returns the value of the variable. Three, the library can create and update a variable that lives inside the PAPI library.

4) Comparing Techniques for Performance Data Exchange: Table III compares various tools on the basis of how they enable performance data exchange. Notably, the PAPI SDE effort is unique in its ability to be generally applicable to *any* type of software module. Over the past fifteen years, the HPC community has iterated on various designs for performance introspection, and it can be argued that event-based callbacks are generally favored over instrumentation. Moreover, the push to include performance introspection capabilities as a part of the standards specification of major communication libraries such as OpenMP and MPI has resulted in the widespread adoption and support of performance tools. In other words, performance tools are increasingly viewed as first-class citizens as opposed to an afterthought within the performance optimization process.

A notable limitation of the MPI_T effort is a lack of tool portability. Specifically, the MPI_T standard allows the MPI implementation the freedom to export any counter or gauge. The standard does not require any mandatory counters or events to be exported. As a result, performance tools need to discover the specific list and names of PVARs and CVARs exported by an MPI library. These names and types are not portable between MPI implementations, and thus, there is little reuse for tool logic that generates performance recommendations or optimizations. OMPT, on the other hand, resolves this problem by separating salient events into mandatory and optional events. This approach can facilitate tool portability across different library implementations.

VI. Distributed Modularization

This section presents how distributed modularization of HPC software has resulted in the formation of coupled application ("in-situ") workflows, ensembles, and services. At the same time, an overview of the accompanying changes within the performance analysis and monitoring tools landscape is also discussed.

A. Overview

Since the late 1990s, there have been several efforts to support *task-coupling* on HPC systems. Specifically in this context, task-coupling is defined as the simultaneous execution of two or more distributed entities in an inter-dependent manner. The shift towards a coupled distributed architecture began in the late 1990s and early 2000s. This process has been accelerated in the last decade by several factors described in Section IV-B. Logan et al. [81] define three categories of task-coupling that capture how emerging HPC software architectures are being designed. In a *strongly coupled* architecture, the coupled entities are tightly integrated and intimately dependent on one another to make progress. Most multi-physics applications such as the XGC-GENE [82] coupled code operate with an assumption of strong coupling.

In a *weakly coupled* architecture, the producer of data can proceed without being concerned about how the data is being

TABLE III: Comparing Techniques for Performance Data Exchange

Property	PERUSE	MPI_T	POMP	Sun OpenMP Interface	OMPT	PAPI SDE
Data Exchange Strategy	Event Callbacks	Counters, Event Callbacks	Source-to-Source Instrumentation	Event Callbacks	Event Callbacks	Counters
Generally Applicable?	No; MPI-only	No; MPI-only	No; OpenMP-only	No; OpenMP-only	No; OpenMP-only	Yes
Access to Fine-Grained Events?	Yes	Yes	No	Partial	Yes	Yes
Standardized Technique?	No	Yes	No	No	Yes	No
Widespread Support?	No	Yes	Yes	No	Yes	Yes
Direct Support for Control?	No	Yes	No	No	No	No
Level of Insight into Module Internals?	High	High	Low	Moderate	High	Low
Tool Portability?	Moderate	Low	High	Moderate	High	Low
Support for Accelerators?	No	Yes	No	No	Yes	Yes

consumed. This interaction model is the *modus operandi* for most in-situ data analysis, visualization, monitoring, and ML services. These services run alongside a "primary" application, typically an MPI-based simulation. Sarkar et al [83] allure to this type of software architecture by giving it the moniker "new-era weak-scaling".

A third emerging type of a coupled architecture is *ensemble computing*. Ensembles find application in domains such as weather modeling, molecular biology [2], and the training of ML models [1]. Ensembles involve simultaneously executing collections of parallel tasks (each of which may be an MPI application) within a single HPC node allocation. Deelman et al. [84] refer to this kind of architecture as "in-situ workflows", distinguishing them from "distributed workflows" that span multiple HPC platforms and scientific instruments. Unless specified, the primary focus of this document is to delineate the critical questions surrounding in-situ workflows.

Distributed modularization has several benefits. By breaking up a large, monolithic code into several smaller distributed modules, the user has increased control on scaling individual entities. In doing so, the application can be executed on a larger node count as compared to a traditional monolithic MPI-based executable. Two, when software modules are deployed as separate parallel executables, it allows larger software development teams to collaborate without worrying about the logistical issues associated with a mammoth code base. Third, specific capabilities such as data-processing and ML-based analysis tasks can not be leveraged directly within the constraints of the MPI programming model. Thus, their integration requires a software architecture that allocates a set of dedicated computing resources for their operation. Four, when modules run inside separate processes, they can be implemented in the language that best suits their specific need. The development of a separate, intermediate language-interoperability tool such as Babel [46] is not required — this job is usually performed by the communication library.

However, several challenges need to be addressed when considering a distributed, modular architecture. First, it is necessary to identify, among the available options, the correct way of splitting up the monolithic application into the constituent (parallel) modules. Second, even when the modularization itself is straightforward, it is not always clear how to allocate the appropriate computing resources to each parallel module. An optimal configuration can be orders of magnitude more performant than a haphazardly configured setup. Third, distributed modularization can result in vast amounts of data traversing the network between the parallel modules. Thus, an efficient middleware or data-transfer mechanism assumes vital importance. Four, when dealing with multiple simultaneously executing components and transient services, performance monitoring and analysis challenges are notably different from those posed by traditional monolithic MPI executables. Distributed components require performance data to be extracted, exported, aggregated, and analyzed online from multiple sources. Given the transient nature of these distributed components and the scale of operation, it is often infeasible for this data to be written out to disk and analyzed offline. We touch upon these opportunities and challenges in the sections that follow.

B. Important Trends in the Computing Industry

A key observation that can be made when surveying the origins of several defining shifts in HPC software development methodologies is that they are usually predated by similar changes within the broader computing industry. Specifically, two computing industry trends bear importance in the context of distributed modularization. The first of these is the notion of a *service-oriented-architecture* [85]. This reference model for SOA defines "services" as to how needs and capabilities are brought together. Fundamentally, SOA embodies the principle of "separation of concerns". Further, the services within an SOA are assumed to have potentially different owners (software development teams) and are developed and deployed independently of each other. SOA architectures directly reflect the structure of software development teams within a larger organization. Arguably, this bears a resemblance to the emerging methodologies for designing and deploying coupled HPC applications.

The other relevant trend is that of the "enterprise service bus" (ESB) [86]. The ESB is the mechanism by which different services within a framework discover and connect. The ESB assumes the job of connecting a service requestor with a service provider, transporting the message requests correctly, implementing load-balancing, and making the necessary protocol conversions. In other words, the ESB functions within the confines of a "publish-subscribe" model of distributed communication. It orchestrates the interaction between the different entities in the system. Arguably, data transfer and staging software such as ADIOS [87] and DataSpaces [88] perform the same duties within a coupled HPC workflow.

C. Composition Model

This section presents a discussion of the various types of distributed HPC frameworks and compares them based on their composition models, coupling strategies, and distributed communication protocols. A *framework* in this context is defined as any software that either (1) functions as a standalone distributed component offering a distinct functionality or (2) enables the development of specialized distributed components through a programming library or platform. This study does not regard I/O libraries such as ADIOS and Decaf [89] as "frameworks". Instead, they are a part of a larger body of work addressing the problem of distributed data management and are discussed separately in Section VI-E2. At the same time, it is worth mentioning that there is significant overlap between in-situ analysis tools, data services, and data management libraries. This overlap is depicted by Figure 4.

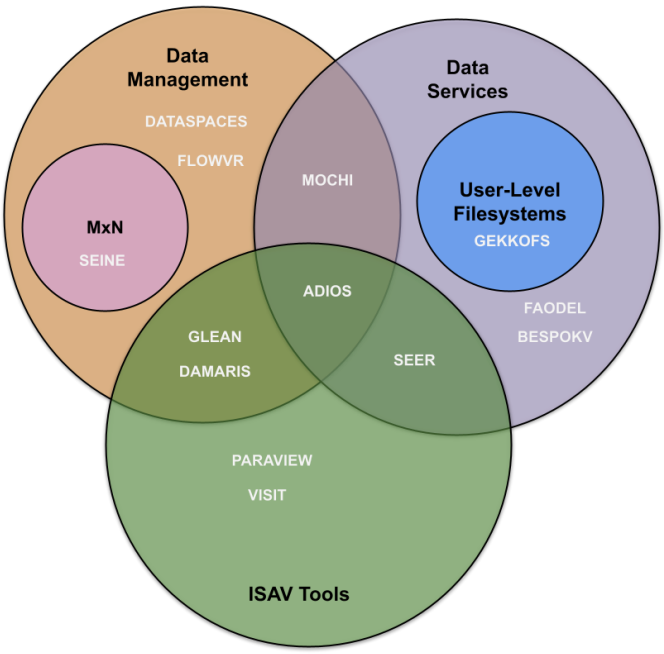

Fig. 4: Overlap Between In-Situ Analysis and Visualization (ISAV) Tools, Data Management Libraries, and Data Services

Composition models define the functional relationship between coupled modules or tasks. Here, we follow and extend the definitions for composition models provided by Logan et al. [81]:

- Strongly-coupled: Two or more coupled modules are tightly integrated and interact in a back-and-forth manner to exchange data during execution.
- Weakly-coupled: Distinct sets of producers and consumers characterize this composition model. Notably, the execution of producer logic does not depend on the consumer's execution, performance, or failure state.
- Hybrid: In a hybrid model, the coupled modules can either be strongly-coupled or weakly-coupled depending on how the system is set up.
- Ensemble: An ensemble represents a distinct type of distributed coupling between tasks, and it is characterized by the execution of a large number of concurrent tasks (each of which may be an MPI application). The tasks may or may not depend on one another. When the tasks are completely independent, the ensemble is "fully decoupled". Such a model resembles "embarrassingly parallel" computation, except that it occurs at a higher level of task granularity.

A coupling strategy determines how the framework functionality is accessed from an external component. For example, the remote procedure call (RPC) is a popular coupling strategy for HPC data services, while most CCA component frameworks are limited to using the parallel remote method invocation (PRMI) model for distributed, inter-component interaction. Table IV presents a list of popular distributed HPC frameworks covering a broad spectrum, a brief description of the functionality enabled by each of these frameworks, and their composition models. Table V is a list of the coupling strategies and the internal communication protocols used by these particular frameworks.

1) Distributed CCA Frameworks: Distributed CCA frameworks were among the first HPC software to develop a solution to enable communication between distributed CCA components. Several efforts such as DCA [47] and PAWS [90] recognized that the CCA specification primarily targeted SPMD parallelism within direct-connected frameworks and had little provision or advice for enabling distributed frameworks. However, when CCA frameworks were beginning to be integrated into production HPC applications, the community realized the need to enable CCA-based componentization of multi-physics codes such as the XGC-GENE [82] code, molecular dynamics applications such as LAMMPS [32], and fusion codes.

Each module in a coupled code is typically implemented as a separate MPI application consisting of one or more CCA components. Within a purely direct-connected framework, enabling collective port invocation for a subset of components is straightforward — the set of communicating processes can make use of a separate MPI communicator. The CCA framework need not be involved in this collective communication process (refer to Figure 2). However, when the collective port invocation needs to happen between two parallel, distributed components belonging to different MPI programs (as in a coupled code), several challenges arise. First, the use of MPI communicators is meaningless when passed across different MPI programs. Second, the CCA framework needs to know and decide which set of processes (or components) participate in a collective, distributed port invocation. Third, there needs to be an agreed-upon synchronization strategy for components participating in the port invocation. Fourth, the framework needs to know how to distribute the data between the callee and the caller components involved in the collective port invocation call. The CCA forum recognized these issues and drafted a communication model called the parallel remote method invocation (PRMI).

In the PRMI model, the callee component blocks until the

caller has completed the method invocation. When this requirement is imposed in a collective communication routine involving the participation of multiple components in the RMI port call, it categorizes distributed CCA frameworks as implementing a strongly-coupled composition model. Specifically, the PRMI call serves as a potentially unwanted synchronization point and reduces the system's effective concurrency. At the same time, the coupled codes that were built using distributed CCA frameworks *required* such patterns of communication between *M* processes of one component and *N* processes of another. The data redistribution issues that arise from this communication pattern are broadly referred to as the *MxN* problem and shall be discussed in depth in Section VI-E2.

Among the CCA frameworks described in Table I, CCAFFEINE [50] is a purely direct-connected framework. MOCCA [59], VGE-CCA [60], XCAT [63], CCAT [61], and LegionCCA [62] support distributed component interactions through RMI, but these frameworks are optimized for scientific applications in the grid as opposed to those employed on HPC platforms. The individual components in these frameworks are not MPI programs, and as a result, the restrictions of the PRMI model do not apply. The grid frameworks employ RMI over SOAP/HTTP or another web-services protocol for distributed interaction. The two notable general-purpose, HPC-optimized distributed CCA frameworks are DCA [47] and SCIRun2 [64]. As depicted in Table V, both DCA and SCIrun2 employ MPI for internal communication within a component cohort. DCA allows collective PRMI communication among a subset of caller components by re-using the MPI communicator support but stipulates that all the caller components take part in the communication. SCIrun2, on the other hand, provides two types of PRMI calls: (1) independent calls that involve one component on both sides of the caller-callee cohort pair, and (2) collective calls that involve every component on both sides of the caller-callee cohort pair [91].

In a more recent work that bears a resemblance to distributed CCA, Peng et al. [92] propose a new strategy to decouple MPI applications into sets of custom process groups. This research stems from the need to address the scalability limitations of the BSP programming model, particularly concerning load imbalance. Instead of building BSP programs where each process is essentially a replica of the same executable (SPMD), the authors propose to break down the application's functionality into a set of specialized operations. The process space is divided into groups of processes implementing these specialized operations. These process groups are organized into a cohesive distributed processing system through a data stream pipeline. An evaluation of this methodology on a Map-Reduce application at a large scale improved the performance of the application by over four times as compared to a standard BSP-style implementation. Arguably, this distributed architecture is one step toward a services-style coupled model that is commonly used in the broader computing industry.

2) HPC Data Services: Two distinct trends have given rise to a class of applications broadly categorized as "data services". The first noteworthy trend is that the performance of traditional file-based parallel HPC I/O storage systems has not been able to keep up with the increase in the concurrency available on the platform. In other words, the total computational performance is growing faster than the total storage performance of the HPC system. As a result, these systems are forced to integrate faster storage technologies such as burst buffers, non-volatile storage-class memories, and NVMe technology to provide a cost-efficient, performant storage stack.

TABLE IV: Distributed HPC Frameworks: Composition Models

Framework	Short Description	Composition Model
DCA	Distributed CCA framework	Strongly-coupled
SCIRun2	Distributed CCA framework	Strongly-coupled
Mochi	HPC data service	Hybrid
Faodel	HPC data service	Hybrid
BESPOKV	HPC data service	Hybrid
ParaView Catalyst	In-situ viz. and analysis	Weakly-coupled
VisIt Libsim	In-situ viz. and analysis	Weakly-coupled
SENSEI	In-situ viz. and analysis	Weakly-coupled
Ascent	In-situ viz. and analysis	Weakly-coupled
TINS	In-situ analysis	Weakly-coupled
Henson	In-situ analysis	Weakly-coupled
Damaris-viz	In-situ viz. and analysis	Weakly-coupled
Seer	In-situ steering	Hybrid
Swift/T	HPC dataflow programming	Ensemble
RADICAL-PILOT	HPC task-based ensembles	Ensemble
Merlin	ML-ready HPC ensembles	Ensemble

The second trend is the broadening of the variety of HPC applications and accompanying I/O access patterns that need to be supported on these platforms. The traditional interaction between MPI-based HPC applications and the storage system is characterized by an input read phase and one or more large, parallel bulk-synchronous writes of structured data for check-pointing purposes. Machine-learning and data-intensive applications such as CANDLE [1] are characterized by irregular read access and the writing of a large number of small files [93]. Further, these new HPC applications require various data types such as key-value (KV) stores and document stores.

Although there are areas of overlap, data services are distinct from data management libraries. Data services offer transient, high-performance data storage and notably richer functionality than just helping move data between different components in the workflow. Broadly, there are two classes of data services that are of interest — those that function as user-level, distributed file systems, and more general programmable data services that can be employed to serve various application needs. The latter class of data services is of primary interest to this study. General programmable data services such as Mochi [93] can be used to build custom distributed file systems.

There have been attempts to leverage the portability and performance offered by MPI for building HPC file systems and storage services [94], [95]. These studies conclude that while MPI is sufficiently capable of serving as the platform upon which these services can be built. However, there are several missing features ("wish-lists"), if implemented, would

give MPI the best chance of widespread adoption. Specifically, these features include extended support for non-blocking calls, one-sided communication, and the flexibility to continue operating in a situation of failure.

User-level distributed file systems have been developed primarily to improve the application performance on platforms that support burst-buffers or node-local, fast storage hardware. Examples of the state-of-the-art, user-level distributed file systems include FusionFS [96], GekkoFS [97], and UnifyFS [98]. These file systems can be seamlessly integrated into any HPC application by specifying a mount point for storage operations. The user-level file system intercepts regular POSIX I/O calls and routes them to the burst-buffer if the file path matches the mount point. GekkoFS implements relaxed semantics for POSIX I/O calls. GekkoFS and UnifyFS employ a background daemon to serve local client requests and store file metadata.

General programmable data services are fundamentally different from user-level file systems in two ways. First, they are designed to support various functionality in addition to simply improving application storage performance. Second, they employ the principle of composability to enable higher-level functionality to be developed from relatively simpler building blocks. The three general programmable data services that we consider here are BESPOKV [99], Faodel [100], and Mochi [93]. A fourth, composable storage service, Malocology [101] employs the principle of composition to decompose Ceph [102] to make it more programmable [93]. However, Malocology operates more like a storage service than a data service and is not considered here. Specifically, it targets the composition of the storage stack that is typically out of the end-users control and within the purview of a system administrator. As a result, Malocology can not be used to build transient user-level services.

BESPOKV [99] is a high-performance distributed KV store. By recognizing the growing importance of KV stores in HPC for coupling, analysis, and visualization purposes, BESPOKV introduces a flexible design for a distributed service based on the decoupling of the data plane and the control plane. The fundamental unit of the control plane is referred to as a *controlet*. The control plane receives client requests and forwards them to one of the distributed *datalets* in the system. Each user-supplied datalet implements a standard KV store API and manages a customizable "backend". Further, the user has complete control over the number and topology of datalets and controlets in the system, thus making the BESPOKV service customizable, flexible, and extensible. However, the evaluation of the BESPOKV system was performed on a virtualized cloud-based system. Thus, its performance when coupled with HPC applications is unknown.

Faodel [100] is a composable data service that aims to serve the general data storage and analysis needs of in-situ workflow components. There are three Faodel components — Kelpie, a distributed KV blob store, OpBox, a library offering primitives for distributed communication patterns, and Lunasa, a memory-management library for network operations. Faodel is intended to be a sink for data from bulk-synchronous applications, asynchronous many-task runtimes, and other ML-based services running inside the workflow. At the same time, Faodel also acts as a source for in-situ visualization and analysis (ISAV) tools that run either within the workflow node allocation or remotely.

The Mochi project [93] arguably represents the largest-scale effort to build customizable, high-performance data services. The fundamental premise behind this effort is the observation that each member of an ever-broadening set of HPC applications has unique data storage requirements and access patterns. Thus, a one-size-fits-all approach is not a good strategy for developing data services. Instead, the Mochi project relies on the composition of microservice building blocks to enable the rapid development of higher-level functionality and customized data services.

The term "microservice" is a concept that originated in the cloud computing industry and is defined by Fowler [103] to be a "building block that implements a set of specific, cohesive, minimal functionality and can be updated and scaled independently". The works by Dragoni et al. [37] and Zimmermann [104] elucidate the various tenets surrounding the development of microservices in the cloud industry. Essentially, a microservice represents an end of the spectrum of distributed services and encapsulates minimal functionality (separation of concerns). The explosion in the number of cloud-computing services such as Amazon, Netflix, and Facebook that have adopted this architecture has given rise to the debate regarding whether microservices represent an "evolutionary" or "revolutionary" step in distributed software development. Jamshidi et al. [105] present both sides of this argument and conclude that the consensus among industry experts is that microservices are "SOA done right", i.e., they are an evolutionary trend in distributed service architectures.

The Mochi framework offers a set of microservices such as the SDS KV store (SDSKV), the BAKE object store, the REMI resource migration service, the SONATA document store (to name a few). An HPC application can compose these microservices in any manner to serve its custom needs. Mochi depends on the Mercury [106] RPC library for communication and the Argobots [107] library for managing concurrency on the server. Notably, Mochi is a multi-institution effort involving five primary organizations contributing to the software's core development. Several other organizations and research teams across national research laboratories in the US have utilized the Mochi framework to develop a wide range of data services. Examples of these data services include the HEPnOS [108] data store for high-energy physics applications, the FlameStore [93] service for storing the results of ML-trained models in a distributed manner, the UnifyFS [98] user-level distributed file system, and the Mobject [93] object-store.

Table IV and Table V present a comparison of the different HPC data services based on their composition model and coupling strategies. HPC data services fall under a category of components that are composed with HPC applications in a hybrid manner. This categorization is due to the flexibility offered by data services. For example, applications could use

the data service purely for storing some partial results during execution. In this first case, data services are weakly-coupled with the HPC application. However, the application could also be simultaneously reading and writing from the data service. In other words, the application depends on the stored results in order to proceed with its computation. In this second case, the data services are strongly-coupled with the HPC application.

The fundamental unit of composition within the Mochi framework is a microservice, while Faodel offers three fixed components. BESPOKV offers customizable units called "controlets" and "datalets" that bear some resemblance to Mochi microservices in their design. Both Mochi and BESPOKV support multiple database backends, while Faodel doesn't appear to do so. Mochi is unique as it recursively uses RPC to compose operations internal to the service and the operations exposed for external application use. BESPOKV and Faodel employ a relatively flat structure and use shared-memory for internal communication within data service components. Mochi is also unique in the sense that it is the only framework that offers more than just key-value store capabilities. Mochi microservices span a more comprehensive range of functionality and thus are more broadly applicable.

3) In-situ Visualization and Analysis: Over the past decade, the flourishing research within the in-situ visualization and analysis (ISAV) research community has resulted in several loose definitions for the term "in-situ". To reduce the confusion over the use of this term, the ISAV community got together to define and categorize ISAV tools under six unique axes. Thus, unless specified otherwise, any reference to the methodologies surrounding ISAV tools in this section follows the definitions laid out by the community [109].

In-situ refers to the processing of analyzing the simulation data as it is being generated. This type of analysis is distinctly different from data processing after it has been written out to a storage medium. The motivation to perform in-situ analysis primarily arises from the inability of traditional HPC I/O systems to absorb the large volumes of data being generated by HPC applications [110]. Specifically, computation capabilities are growing faster than the storage I/O bandwidth. As a result, it is simply infeasible to write the entire simulation data to long-term storage for offline analysis.

In-situ methods enable the online, parallel processing of large amounts of simulation data to result in significantly smaller volumes of "interesting" simulation features written to disk. All the ISAV tools considered here have either (1) on-node proximity or (2) off-node proximity. This study does not consider an in-depth study of the third variety of ISAV tools that run on "distinct computing resources", *except* under the circumstance that this distinct computing resource happens to be a remote monitoring client (human-in-the-loop interaction). Specifically, the tools that utilize distributed computing resources spanning multiple geographical sites to perform computation fall under the category of grid computing systems, and thus, they are not a primary focus of our study.

As depicted by Table IV, most ISAV frameworks are composed along with the application in a weakly-coupled manner. In other words, the application is the producer, and the ISAV tool is the consumer. The application does not depend on the result of the analysis to proceed with its computation. The general assumption made by ISAV tools is that the analysis to be run is pre-determined (automatic, adaptive, or non-adaptive). One exception to this rule is the Seer [111] in-situ steering framework. In Seer, the simulation takes input from a human-in-the-loop in a non-blocking way, i.e., there is a back-and-forth interaction between the simulation, ISAV tool, and the human user. Thus, Seer is composed along with the application in a hybrid manner. Notably, Seer uses the Mochi [93] data service as an intermediate communication module to enable this interaction.

Concerning the coupling strategy (a combination of proximity, access, and integration type), Paraview Catalyst [112], VisIt Libsim [113], SENSEI [114], TINS, [115], Ascent [116], HENSON [117], Damaris-viz [118], and Seer [111] primarily interact with the application through shared memory. Except for Ascent, they all implement a dedicated API. Ascent supports multiple backends and hence implements a multi-purpose API. Typically, ISAV tool integration with an MPI application occurs through the use of a client library. The simulation invokes the ISAV routine locally on each MPI process, and the ISAV tool client converts the simulation data into a format suitable for analysis and visualization. Some tools such as SENSEI support the off-node transfer of this ISAV data to other components running within the in-situ workflow. Further, VisIt and ParaView support remote visualization of this data. Except for Damaris-viz and SENSEI, all the ISAV tools considered here support only time division between the application and the ISAV routines. Damaris-viz is implemented using the Damaris [119] I/O framework that allocates a dedicated compute core for the execution of ISAV routines. Damaris-viz communicates with the simulation through shared-memory belonging to the operating system, and thus, it can support space division and time division.

4) HPC Ensemble Frameworks: Among the many changes in the HPC landscape over the past few years, an important one is the emergence of a new class of scientific workloads referred to as *HPC ensembles*. Traditional workloads on HPC clusters are characterized by a small number of large, long-running jobs. The push towards uncertainty quantification (UQ) [120] has resulted in ensemble workloads that consist of a large number of small, short-running jobs [121]. These ensembles are also referred to as "in-situ workflows" [84]. An analysis of the batch job submissions on Lawrence Livermore National Laboratory (LLNL)'s Sierra machine reveals that 48.5% of all submitted jobs reveal a pattern that typifies ensembles [121].

Individual jobs or tasks within an ensemble can be fully decoupled (such as the UQ pipeline [120]), or their coupling can be represented by a directed acyclic graph (DAG) or dataflow graph. The latter form represents the more general case. Notably, it is not uncommon for these tasks to represent a collection of different executables. There are a few fundamental capabilities that ensemble frameworks must provide — an ensemble programming system that includes a way to specify

TABLE V: Distributed HPC Frameworks: Coupling Strategies and Communication Protocols

Framework Name	Basic Computation Unit	Coupling Strategy	Internal Communication Protocol
DCA	CCA component	PRMI	MPI
SCIRun2	CCA component	PRMI	MPI
Mochi	Microservice	RPC	Mercury RPC
Faodel	OpBox, Lunasa, Kelpie	RPC	Shared-memory
BESPOKV	Datalet, Controlet	N/A	Shared-memory
ParaView Catalyst	MPI process	Shared-memory	MPI
VisIt Libsim	MPI process	Shared-memory	MPI
SENSEI	MPI process	Shared-memory	MPI/ADIOS
TINS	Task	Shared-memory	MPI+Intel TBB
Henson	Henson Puppet	Shared-memory	MPI
Ascent	MPI process	Shared-memory	MPI/ADIOS
Damaris-viz	MPI process	Shared-memory	MPI
Seer	MPI process/Microservice	Shared-memory/RPC	RPC
Swift/T	Turbine Task	Dataflow	MPI
RADICAL-PILOT	Compute Unit	Task DAG	ZeroMQ
Merlin	Celery Worker	Task DAG	RabbitMQ

task dependence, support for inter-task communication, and hardware resource management (scheduling).

Swift/T [122] is a programming language and runtime for in-situ ensemble workflows. Swift is the scripting language used to specify the composition of workflow tasks, and Turbine [123] is the runtime used to manage the execution of tasks on an HPC cluster. Swift is a naturally concurrent language that uses a dataflow graph to infer dependencies between tasks, each of which can, in turn, be an MPI program itself. Swift is a scripting language that can natively invoke code written in C, C++, or Fortran and scripts in Python or Tcl.

Notably, the dataflow specification is not a static graph but dynamically discovered as the program executes. Internally, the Swift/T program is converted into an MPI program that runs multiple copies of the Turbine runtime on a machine allocation. These Turbine instances (MPI processes) manage the execution of Swift/T tasks by balancing the load between the available resources. When the Swift/T task is itself a parallel MPI program, Turbine creates a separate MPI communicator group to represent the MPI program. MPI is also used by Swift/T tasks to communicate with one another. Swift/T and UQ Pipeline [120] share the distinction of being single-cluster ensemble frameworks. These frameworks cannot be used to run tasks across multiple HPC clusters. At the same time, by bootstrapping on top of MPI, they do not need to deal with the complexities of interacting with a job scheduler and working around the security limitations posed by traditional HPC batch systems.

Merlin [124] and RADICAL-PILOT (RP) [125] are ensemble workflow frameworks that blur the line between HPC and grid computing. While Merlin is a workflow framework tailored for HPC ensembles that result in data being used for training ML models, RP is a general-purpose workflow framework applicable to any system. These two frameworks share several common design elements. One, they both support task execution across multiple HPC clusters. Two, task dependence is inferred through an internal task DAG. Three, both these frameworks employ a script-based programming "frontend". Four, there is an *external* centralized service or node that hosts the task queue. The workers that are launched on compute nodes within the batch job allocation (Celery tasks in Merlin, Agents in RP) pull from this external task queue in what resembles a producer-consumer model. Five, there is an entity that manages the worker instances within a batch job allocation and performs the work of a scheduler. This entity is referred to as a "Pilot" in the RP framework and is the Flux [121] component within Merlin.

However, there are some differences between these two frameworks. One, while Merlin supports inter-task communication through a data management library called Conduit [126], RP appears to use the file system to perform this action. Two, while Merlin uses RabbitMQ for internal communication between its components, RP uses the ZeroMQ [127] messaging platform. While there exist mature, general-purpose, distributed workflow management systems such as Pegasus [128], they are generally not applicable for ensemble HPC workflows. Three reasons are provided by Peterson et al. [124] to support this claim: (1) they often have a considerable upfront user training cost, (2) they do not support accelerators such as GPUs and FPGAs, and (3) they do not work well under the security constraints imposed by HPC data centers.

D. Resource Allocation and Elasticity

When two or more components are coupled together, one of the most fundamental questions that need to be addressed is how they share resources. *Resource allocation* refers to the methodology by which computing resources are divided among a set of simultaneously executing components. A related problem is the ability to change an existing resource allocation scheme dynamically. *Resource elasticity* refers to the ability of a framework to dynamically shrink or expand the number of resources being utilized in response to an internal change in application requirements or external factors such as performance variability and power constraints.

1) Resource Allocation: Table VI lists the resource allocation schemes for the set of distributed HPC frameworks introduced in Section VI-C. Broadly, the common resource

TABLE VI: Distributed HPC Frameworks: Resource Allocation Scheme

Framework	Resource Allocation Scheme
DCA	Distinct nodes, same machine
SCIRun2	Distinct nodes, same machine
Mochi	Hybrid
Faodel	Distinct nodes, same machine
BESPOKV	Distinct nodes, same machine
ParaView Catalyst	Local node, same process
VisIt Libsim	Local node, same process
SENSEI	Hybrid
Ascent	Local node, same process
TINS	Local node, separate processing core
Henson	Local node, same process
Damaris-viz	Local node, separate processing core
Seer	Hybrid
Swift/T	Distinct nodes, same machine
RADICAL-PILOT	Distinct nodes, different machines
Merlin	Distinct nodes, different machines

allocation schemes can be divided into five categories. Starting from the schemes that involve the highest degree of resource sharing to the ones that involve the lowest degree of resource sharing, these categories are:

1) Local node, same process: The framework and the "application" that utilizes the framework live in the same address space, interact through regular function calls and share the same computing resources. They may or may not share processing threads.
2) Local node, separate processing core: The coupled components live on the same computing node and share the computing resources. However, they do not live in the same address space, and thus they do not share processing threads.
3) Distinct nodes, same machine: The coupled components simultaneously execute on distinct computing nodes within the batch job allocation. The only resources they share are network resources (switches and routers) and the parallel I/O filesystem.
4) Hybrid: In a hybrid resource allocation scheme, there is significant flexibility in how the resources are divided up among the coupled components. Specifically, any one of the schemes (1), (2), or (3) can be employed.
5) Distinct nodes, different machines: The coupled components share a minimal amount of resources. They can run across multiple HPC machines and communicate through a centralized messaging or communication framework.

Most distributed CCA component frameworks such as DCA [47] and SCIRun2 [64] employ a resource allocation scheme in which the individual components are executed on distinct computing nodes within the same machine. Typically, these components are deployed as independent MPI programs within the same batch job allocation. Faodel [100] and BESPOKV [99] also employ this same type of resource allocation strategy. Within the class of frameworks referred to as "data services", the Mochi [93] infrastructure is unique because it offers a hybrid resource allocation scheme. Mochi microservices can be configured to run inside the same process as the "client" (MPI simulation), on different processes running on the same node as the client, or distinct computing nodes within the batch job allocation. Notably, the Mercury RPC framework [106] employed by Mochi offers an RPC API that abstracts the specific resource allocation scheme in use.

Typically, ISAV tools are tightly integrated with the MPI application and employ a time-division or space division scheme to share data within the application. Their proximity to the application ensures that the data transfer overheads are minimized. ParaView [112] VisIt [113], and Ascent [116] employ the resource allocation scheme of type (1) and run inside the same process as the MPI application. However, they support an external remote-monitoring client (such as a Jupyter notebook) to monitor the results of the in-situ analysis.

SENSEI [114] functions as a generic bridge between an HPC application and several in-situ implementations such as ParaView and VisIt. Additionally, SENSEI can be coupled to an external component running on the system through ADIOS. Thus, SENSEI implements a hybrid resource allocation scheme. TINS is built upon IntelTBB [21], and the analysis routines are launched within separate threads sharing the local computing resources with the application. Among ISAV tools, Damaris-viz [118] and TINS [115] are unique because they run within a dedicated processing core on each computing node. Damaris-viz is itself an MPI application that is launched beside the simulation. Each Damaris-viz MPI process communicates *only* with the MPI processes belonging to the simulation that runs on the same computing node. Seer [111] employs a hybrid scheme wherein the simulation is coupled with an external Mochi SDSKV service. Seer offers the user the flexibility to determine the exact resource allocation for the Mochi service.

Swift/T [122] tasks are scheduled onto processes belonging to a single MPI application. Here, we consider the resource allocation scheme employed at the task level. Since multiple tasks can run simultaneously on distinct nodes (where each task is itself an MPI program), Swift/T employs a scheme of type (3). In contrast, Merlin [124] and RP [125] are unique as their tasks can span multiple HPC machines. These two ensemble frameworks employ a resource allocation scheme of type (5).

2) Resource Elasticity: Resource elasticity is the ability to *dynamically* expand or shrink the number of resources being utilized *in response* to external stimuli or a change in application requirements. Note that resource elasticity is just one method by which a system can adapt itself, and it does not have the same meaning as dynamic adaptivity. Table VII lists the current support for elasticity within different distributed HPC frameworks.

Most distributed HPC frameworks do not natively support resource elasticity. As pointed out by Dorier et al. [129], one of the factors is the dependence on MPI as a bootstrapping

mechanism. Although the MPI standard has provisions to support the dynamic addition of new processes, most widely used MPI implementations do not support this feature [130]. The ones that do support elasticity (such as Adaptive MPI [131]) require a significant modifications to the application code. Supporting elasticity within a traditional HPC cluster requires changes to the scheduler and cost model as well. Specifically, *over-provisioning* is not a feature supported by most existing HPC cluster schedulers. Motivated by the pay-as-you-go cost model and the elasticity supported on cloud platforms, previous efforts [130], [132] explore the elastic execution of MPI programs within the cloud. Typically, the techniques implemented by these efforts require some form of checkpoint-restart combined with a monitoring and decision-making framework.

However, none of these efforts have successfully demonstrated the elastic execution of MPI programs within the context of a traditional HPC cluster. Therefore, it is safe to say that currently, MPI programs do not support elasticity. ISAV tools such as ParaView and VisIt run within the context of an MPI process and employ time-division coupling to share computing resources. Therefore, any ISAV tool that depends on MPI is limited in its support for elasticity. TINS is an ISAV tool wherein the analysis routines run within a separate TBB thread associated with a dedicated "helper core". When analytics routines are available to run, this core is used exclusively to execute the analytics routines to prevent interference with the simulation. Otherwise, the helper core is utilized as a common core for processing simulation tasks.

The only class of frameworks that support elasticity are data services. Specifically, BESPOKE supports "scale-out" resource elasticity through the dynamic addition and removal of its core components — controlets and datalets. The support for elasticity has recently been added to the Mochi framework. Specifically, the BEDROCK microservice functions as a bootstrapping mechanism through which other microservice instances can be dynamically instantiated. Note that although these frameworks support resource elasticity in some form, none provide the ability to do so *automatically*.

Further, none of the ensemble frameworks surveyed here support resource elasticity. Each of these frameworks requires the user to either specify a *fixed* number of MPI tasks (Swift/T) or a fixed batch allocation size for a given set of tasks (Merlin and RP). Once these tasks are mapped onto the computing resources, there is no way for them to request more (or less) resources dynamically should the need arise. Note, however, that the *resource utilization* levels within a batch job allocation can naturally wax and wane depending on the particular sequence of task execution. Arguably, this is not the same as the ability of a framework to enable resource elasticity.

TABLE VII: Distributed HPC Frameworks: Resource Elasticity

Framework	Supports Elasticity?	Unit of Elasticity
DCA	No	N/A
SCIRun2	No	N/A
Mochi	Yes	Microservice
Faodel	No	N/A
BESPOKV	Yes	Controlet, Datalet
ParaView Catalyst	No	N/A
VisIt Libsim	No	N/A
SENSEI	No	N/A
Ascent	No	N/A
TINS	Yes	Task
Henson	No	N/A
Damaris-viz	No	N/A
Seer	Partially	Microservice
Swift/T	No	N/A
RADICAL-PILOT	No	N/A
Merlin	No	N/A

E. Data Management Strategy

A fundamental question that arises when coupling two or more distributed HPC frameworks or applications is how to transfer and stage data between them efficiently. Within the context of our study, HPC data management frameworks are differentiated from more general HPC data services. From a functional standpoint, HPC data management frameworks exist solely to transfer data between coupled components, while data services offer a more broad set of capabilities.

1) MxN Problem: Multi-physics applications launched as two or more strongly-coupled, separate MPI programs need some way to communicate and exchange data with each other. The general problem of redistributing data from an application launched with *M* processes to an application launched with *N* processes came to be known as the *MxN* problem [91]. The MxN problem was recognized as a major research area within the general field of distributed CCA framework design. As elucidated by Zhao and Jarvis [133], the MxN communication typically involves the following steps:

- Data translation: Data stored in one format (for example, row-major form) may need to be translated into another form (column-major form).
- Data redistribution: The sender and the receiver must be aware of the exact set of elements they expect to communicate with each other.
- Computing a communication schedule: Once the set of elements to be sent are identified, each sender needs to identify the portions of these data elements to be sent to specific receivers and accordingly compute a communication schedule.
- Data transfer: After the communication schedule has been computed, the last step involves the actual data transfer itself.

Table VIII uses these steps to list out and differentiate a set of popular MxN frameworks. Importantly, MxN frameworks use one of two methodologies to enable MxN data redistribution — PRMI or a component-based implementation. Distributed CCA frameworks such as DCA [47] and SCIrun2 [64] employ the PRMI model to perform complete MxN data redistribution. DCA is built upon MPI and allows a subset of components on the sender side to redistribute

TABLE VIII: MxN Coupling Frameworks

Framework	Conceptual Technique	General Framework?	Data Redistribution	Communication Schedule Calculation	Data Transfer
DDB	Component-based	Yes	MxN	Centralized	Parallel
CUMULVS	Component-based	No	Mx1	Centralized	Parallel
Seine	Component-based	Yes	MxN	Centralized	Parallel
MCT	Component-based	No	MxN	Distributed	Coupler
PAWS	Component-based	Yes	MxN	Centralized	Parallel
InterComm	Component-based	Yes	MxN	Distributed	Parallel
DCA	PRMI	Yes	MxN	Distributed	Parallel
SCIRun2	PRMI	Yes	MxN	Distributed	Parallel

data. However, it requires all receiving components to take part in the data redistribution process. SCIrun2 allows two forms of data redistribution — collective and point-to-point. In collective data redistribution, all the components across the sender and receiver side must necessarily be involved in the communication. All the other frameworks presented here — DDB [134], CUMULVS [135], Seine [136], MCT [137], PAWS [90], and InterComm [138] employ a separate CCA component to perform the data redistribution.

Among the component-based frameworks, all of them are generally applicable to any distributed CCA application except for CUMULVS and MCT. CUMULVS is designed as a distributed component that allows a human user to visualize, interact with, and steer the MPI-based simulation as it is running. Specifically, this visualization component is implemented as a serial application that generates a set of requests to "pull in" the necessary portions of the domain (multi-dimensional array) from multiple MPI processes while they are running. As a result, CUMULVS only supports an Mx1 data redistribution scheme. Further, the communication schedule is calculated in a centralized manner within the serial visualization application. The MPI processes transfer their portions of the requested domain in a parallel fashion. On the other hand, the MCT component is explicitly designed to work with earth science applications such as CESM [31]. Although MCT supports an MxN data distribution scheme and the calculation of communication schedules locally within every sender process, the data transfer mechanism is unique. Each of the M sender processes routes their individual data elements through a "coupler" module that performs the data redistribution and forwards them to N receiver processes.

DDB, InterComm, Seine, and PAWS are general-purpose component-based MxN frameworks that support full MxN data distribution. Among these, InterComm is unique concerning the methodology by which the communication schedule is calculated. Specifically, InterComm identifies a set of "responsible" processes that compute the communication schedule after gathering the source and destination data structure representations. Once this task is complete, these "responsible" processes transfer the schedules to the processes that need them. Finally, each sender process proceeds to transfer the data in a parallel manner.

On the other hand, DDB, Seine, and PAWS employ a dedicated centralized component to calculate the communication schedule. DDB employs a two-level scheme where each coupled application nominates a control process (CP) to communicate domain information and data layout to a single registration broker (RB) during the registration phase. Once the RB receives all the information from each CP, the RB performs the matching between data producers and consumers and calculates the communication schedule. This communication schedule is broadcast to all other processes when they finish their registration phase.

PAWS and Seine are similar because they introduce a distinct distributed component to orchestrate the data redistribution. Both these frameworks introduce the notion of a "shared virtual space" used to inform the communication schedule. Notably, the calculation of the communication schedule is performed during the registration phase, making this phase the most expensive routine in terms of execution time. The notion of a shared virtual space and the introduction of a dedicated central component to perform data redistribution allows these frameworks to support process dynamism elegantly. New processes register themselves and their portion of the domain (multi-dimensional array) with the Seine framework, after which they can seamlessly take part in the MxN communication without being aware of who the actual senders are.

TABLE IX: Data Staging and I/O Frameworks

Framework Name	Data Staging Nodes?	Asynchronous Data Transfers?	Allows Process Dynamism?
ADIOS	Flexible	Yes	Yes
DataSpaces	Yes	Yes	Yes
FlexPath	No	Yes	Yes
GLEAN	Yes	Yes	Yes
Decaf	Yes	No	No
FlowVR	No	Yes	No
Damaris	No	Yes	No

2) Data Staging and I/O Frameworks: While MxN frameworks address a critical problem that arises when coupling two MPI applications, it is not difficult to see that the class of applications (and hence the coupling methodologies) they support are limited. Specifically, most, if not all MxN frameworks assume that the data needs to be transferred (1) immediately, (2) synchronously, and (3) *only* between two MPI applications. In other words, MxN frameworks are generally associated with strongly-coupled applications.

The advent of transient, in-situ visualization and analysis tasks together with the growing disparity between computing and parallel storage performance has given rise to a class of frameworks that offer *general* data staging and I/O manage-

ment capabilities. Specifically, these data staging frameworks offer the ability to transfer and stage data asynchronously and between more than two coupled components. Moreover, they allow significant flexibility in describing the data to be staged and potentially also support process dynamism, i.e., the ability to continue operating when processes enter or leave the system.

ADIOS [87] is arguably the most widely supported data staging and I/O framework for HPC applications. ADIOS originally started as an API that offered seamless asynchronous I/O capabilities. The growing disparity between compute and storage performance meant that applications could no longer afford to synchronously write massive amounts of data to disk without severely damaging their overall performance. ADIOS offered a way out of this problem by staging the data locally and performing the write operation only during the application's compute phase. The ADIOS API was initially designed to be POSIX-like while decoupling the *action* of performing a parallel write with *when* and *where* the data is written out. Since its initial release, ADIOS has become synonymous with an I/O API that offers various data staging and in-situ analysis capabilities. Specifically, ADIOS employs several backend "transport methods" that effectively determine the sink for the data. Examples of transport methods include specialized data staging software such as DataSpaces [88] and FlexPath [139]. ADIOS' modular design has ensured widespread adoption as a "high-level" I/O API across various classes of HPC applications.

GLEAN [140] is a data staging and in-situ analysis library that offers asynchronous data staging and offloading capabilities. GLEAN can be leveraged either directly through its API or by the transparent library interposition of HDF5 routines. Note that the latter technique has the benefit of not requiring any application code changes. Data from GLEAN-instrumented applications is asynchronously transferred to a dedicated set of staging nodes, effectively functioning as an in-memory "burst-buffer". After the data is transferred to the staging nodes, the data is available to other components such as in-situ analysis tools. As such, GLEAN does not offer any special features to transform application data into different formats. Because of the passive, decoupled way in which it operates, GLEAN is indifferent to process dynamism within the source application.

DataSpaces [88] is a project that leverages the "shared virtual space" concept first introduced by the Seine [136] MxN coupling framework. DataSpaces builds upon the multi-dimensional data representation and linearization scheme in Seine and offers data coupling via a separate *dataspaces* distributed component. This MPI-based dataspaces component runs on a dedicated set of computing nodes that asynchronously stage data from multiple coupled applications. Further, the dataspaces component offers an API that allows in-situ analysis to be performed and an API to install a monitor that checks for updates on a region of interest. A benefit of having a separate data staging component is the implicit ability to support process dynamism.

The FlexPath [139] system offers a typed publish-subscribe mechanism for connecting data producers with data consumers within a coupled HPC application. The data producers effectively define and generate a data stream to be consumed by any distributed component running on the system or remotely. Notably, FlexPath employs a direct-connect scheme to transfer data objects directly between a publisher and subscriber. This scheme is different from a traditional brokered architecture employed by other data staging software such as DataSpaces. A direct-connect design choice implies that data is staged locally within every publisher process. FlexPath hooks into application I/O routines through the ADIOS I/O API. FlexPath utilizes the EVPath [141] communication substrate to transfer data between different components. Every new process entering the system communicates the data objects of interest to a local message coordinator that, in turn, calculates the publishers from which it must fetch data. This decoupled approach enables FlexPath to support process dynamism.

FlowVR [142] and Decaf [89] represent data staging software that depend on the concept of *dataflows* to transfer data between coupled components ("nodes"). Decaf is a data staging software that depends on MPI. Specifically, the Decaf system takes as input a JSON file representing the coupled applications and splits the global MPI communicator among the various "nodes" (coupled MPI applications) and "links" (data staging processes). Note that a link separates two nodes. The producer node transfers data to the link, and the link can optionally transform the data or perform specialized analysis on the data before forwarding it to the consumer node. In FlowVR, however, the data transfer is performed in the background by a FlowVR daemon process on every computing node, with limited support to stage data. However, FlowVR data in transit can be acted upon by a set of pre-defined "filters" to transform it before it is passed on to the next component in the flow. Both FlowVR and Decaf need to be informed of the task graph before execution, and as a result, they cannot handle process dynamism.

Damaris [119] is an I/O management framework that relies on a dedicated processing core on each computing node to perform asynchronous I/O. The global MPI communicator of the application is split into two — one for the main application itself and the other for the Damaris component. Processes within a computing node asynchronously communicate their I/O data with the local Damaris process through shared memory. The Damaris process optionally hosts a set of plugins that act upon this data to compress, analyze, and finally commit it to a long-term storage medium. Due to its reliance on MPI, Damaris is limited in serving as a general distributed data staging software. Instead, it can serve as a data source for in-situ analysis tools such as Damaris-viz [118].

F. Performance Tools

As the number of simultaneously executing components within a distributed, in-situ workflow continues to rise along with the scale of the HPC machine, the application of performance tools to ensure the proper and optimal operation

of the workflow is growing in importance as well. Traditionally, HPC performance tools are employed for the *offline* performance analysis of monolithic MPI applications. State-of-the-art performance tools such as Score-P [143], TAU [8], CALIPER [10], and HPCToolkit [9] collect a rich profile and trace that is ultimately written out to disk for offline performance analysis.

With the advent of coupled applications and in-situ workflows, a different approach is needed for practical performance analysis at scale. Specifically, several challenges must be addressed within each of the three classical performance engineering categories — performance measurement, performance monitoring and analysis, and performance control and adaptivity. This section considers these challenges in detail. Table X summarizes the level of tool support currently available within each of these categories for the different types of coupled applications and in-situ workflows previously introduced.

1) Performance Measurement: Performance instrumentation, measurement, and sampling often represent the first steps in the performance engineering of an application. The tool API instrumentation is added either explicitly (source instrumentation) or implicitly (library interposition). Measurements are then made when the application executes. Traditionally, these measurements are gathered and written out to disk when the application finishes executing.

An important observation to be made about in-situ workflows is that many depend on one or more MPI-based components. Specifically, this is the case for commonly used strongly-coupled MPI applications (XGC-GENE and LAMMPS) and ISAV tools such as ParaView [112], SENSEI [114], and Ascent [116]. In such a situation, the rich support for measurement in existing HPC performance tools can be leveraged and extended as needed. Specifically, there are two *types* of measurements to be made. One, the measurements that represent internal function execution times and metrics for each coupled component. Second, the measurements that correspond to the interactions between the components. Wolf et al. [144] identify key ADIOS [87] routines to instrument for capturing data movement between coupled applications. Given its widespread use, instrumenting high-level ADIOS routines automatically enables insight into any transport method utilized underneath.

Malony et al. [145] demonstrate a methodology by which Ascent routines are instrumented and analyzed using TAU's plugin architecture [146]. ISAV tools are typically tightly integrated with the MPI application, and the ISAV tool routines are invoked synchronously by each MPI process. This design presents an opportunity for TAU plugins to "hook into" these synchronous executions to dynamically calculate the contributions of the Ascent routines to the captured performance events. A key observation here is that plugin architecture offers a doorway to both the performance event data and the tool measurement API.

Traditional HPC performance analysis tools built for MPI-based applications cannot be generally applied to gather performance measurements from HPC data services such as Mochi [93] and BESPOKV [99]. HPC performance tools implicitly assume that control is not passed between two different distributed applications. Data services break this assumption through an RPC-based client-server communication model. Thus, the HPC community needs to look to the general cloud computing industry for answers to measuring data service performance. Sambasivan et al. [147] summarize the extensive body of research on a class of distributed tracing tools that implement request metadata propagation. Briefly, this technique involves generating a unique "request ID" and the subsequent propagation of this request ID through the system by RPC invocations. The request ID is then used to tie together events that are *causally related* and thus, this technique can be used to capture distributed callpaths, request structures, and also be used to compare request flows [148]. Industry tracing tools such as Dapper [149] and Jaeger [150] employ request metadata propagation on production-scale cloud computing systems.

Performance measurement of HPC ensembles is an open area that is yet to be targeted by the HPC tools community. Table X enlists the current level of performance measurement support for HPC ensembles as "partial". While traditional measurement tools can be used to capture the execution time of individual ensemble tasks that happen to be MPI-based applications (or serial tasks), there is no existing tool to provide a holistic picture of the task execution in conjunction with the dynamic task interactions and resource utilization measurements. An integrated approach is required to capture and correlate all three types of performance measurements.

TABLE X: Performance Tools for Coupled Applications and Workflows

Application Type	Measurement Tools Exist?	Monitoring & Analysis Tools Exist?	Control & Adaptivity Tools Exist?
Strongly-coupled MPI	Yes	Yes	Partial
ISAV Tools	Yes	Yes	Partial
Data Services	No	Partial	Partial
Ensembles	Partial	Partial	No

2) Performance Monitoring and Analysis: Arguably, the bulk of performance solutions for coupled in-situ workflows fall into the category of monitoring and analysis tools. Partly, this is due to the observation that several existing performance measurement tools can be leveraged directly for *most* components within the coupled in-situ workflow. Therefore, the existing research focuses on monitoring and exporting this data to an external entity for aggregation and analysis.

WOWMON [151] is a monitoring and analysis infrastructure for in-situ workflows. WOWMON instruments coupled applications using traditional HPC performance tools to generate performance measurements. These performance measurements are buffered and sent over EVPath [141] to a central workflow manager. The performance data is analyzed to gather the end-to-end latency of the workflow. Further, this performance data is passed through a machine learning profiler to rank the instrumented metrics according to their correlation with the end-to-end latency of the workflow.

SOS [152] is a distributed monitoring tool that offers the ability to collect, aggregate, and analyze performance data from multiple simultaneously executing coupled applications. The SOS client interfaces with the application to collect performance data which it then forwards to a collector daemon running on the same computing node. The daemon processes are organized into an overlay network that aggregates local performance data. The Lightweight Distributed Messaging System (LDMS) [153] and MRNet [154] tools also employ an overlay network to aggregate performance metrics within an HPC cluster. LDMS, together with Ganglia [155] represent a class of monitoring tools that can be employed for system resource monitoring. Unlike SOS, they do not offer a client instrumentation library, and thus, they can not be used to capture application performance data directly.

As the level of concurrency on modern HPC systems continues to rise, the volume of performance monitoring data produced can significantly perturb application performance [81], [144]. Thus, there is a growing interest in sub-sampling and analyzing performance data *in-situ* to reduce trace sizes before a global aggregation is carried out. The MOnitoring Analytics (MONA) [36], [144] approach speaks to this kind of a technique. Specifically, MONA employs the SOS [152] monitoring tool to aggregate and analyze TAU performance data from a coupled MPI application. The TAU performance data is piped to SOS through a TAU plugin. The aggregated data is analyzed and visualized on an interactive dashboard.

Chimbuko [156] is a workflow-level in-situ trace analysis tool. Chimbuko analyzes the performance data from a coupled application workflow to generate *performance anomalies*. Specifically, the TAU plugin infrastructure is utilized to export performance traces to a process-local anomaly detection (AD) module. The AD module periodically communicates with a central AD parameter server to update its internal anomaly thresholds based on a *global* view of statistical outlier information. Finally, when Chimbuko detects an anomaly, it captures and stores provenance information that helps identify the context in which the anomalous value was recorded.

While the design of distributed tools such as SOS can be generally applied to monitor HPC data services, the data model used to capture performance information must be carefully studied. HPC data services that run in highly concurrent environments handle thousands of requests per second. Thus, the instrumentation library must be able to operate efficiently under a high degree of concurrency. Not only this, the accompanying in-situ analysis needs to be able to track changes *across time* to be able to observe poor service performance. Thus, a time-series monitoring approach combined with sophisticated node-local analysis for trace data reduction may be a viable strategy. There are several state-of-the-art cloud-based tools such as Prometheus [157] and Graphite [158] that implement a time-series database. However, these tools operate within the constraints of a commodity hardware and stack, and thus, they need to be appropriately modified to suit HPC service requirements.

3) Control and Adaptivity: Several tools implement adaptive algorithms within the context of individual applications. The MPI_T interface is a notable effort to enable tool integration for control and adaptivity of MPI applications. Specifically, performance tools can use control variables (CVARs) to effect dynamic adaptation and control. The APEX [159] monitoring system exposes a set of *listeners* that external tools can use to implement control policies. The TINS [115] in-situ framework implements a naturally elastic threading model that enables the sharing of computing resources between simulation and analysis routines.

However, fewer tools enable control and adaptivity resulting from data analysis of a coupled execution. The MONA [144] project studies the cross-application interactions resulting from an XGC-GENE coupling to determine a more optimal task placement for both applications. However, this more optimal task placement cannot be implemented immediately. Instead, it is a valuable starting point for subsequent coupled executions. The Seer [111] in-situ steering framework enables a human user to interact with a running simulation to execute custom, dynamic in-situ analysis routines. Older monitoring tools such as Falcon [160] also enable user-interactive simulation steering of traditional monolithic executables. Pufferscale [161] is a multi-objective optimization framework that can simultaneously and dynamically balance load *and* data across a set of distributed Mochi [93] microservices. This re-balancing is enabled through the REMI resource migration microservice. However, Pufferscale cannot enable an online re-scaling (or resizing) of the number of distributed microservices.

VII. Trends and Open Areas

This section describes the important trends and open areas that inform future work for HPC performance tools.

A. Trends

This section describes the major trends and open areas that inform future work for HPC performance tools.

There are several significant trends concerning the evolution of modern HPC applications. First, the number of distinct components coupled together has been steadily increasing over the past two decades. As distributed CCA frameworks became popular, strongly-coupled MPI applications were developed. ISAV tools and data management frameworks arrived on the scene, increasing the number of coupled, distributed components. HPC ensembles push the barrier even further, resulting in hundreds to thousands of small, short-running tasks.

Second, the types of applications that require high-performance capabilities have exploded. HPC platforms that were once strictly the domain of bulk-synchronous parallel applications now share the space with transient data analysis tools and ML tasks that were traditionally executed on desktop-class single-node machines. On the one hand, tools based on statistical analysis extract helpful knowledge from the large amounts of data generated by HPC applications, and their integration with traditional BSP-style codes requires careful thought. As a result, the HPC community has borrowed

ideas and techniques from the general cloud-computing and artificial intelligence (AI) communities. On the other hand, some of these ML and AI tools are large-scale distributed applications in their own right. Their special needs are driving the decisions behind the procurement of these multi-million dollar HPC machines [35].

Third, the relatively slow growth of traditional file-storage performance on HPC machines compared to the computational performance is the single most significant hardware factor contributing to the emergence of several new classes of distributed frameworks described in this document. The resulting storage heterogeneity and the inclusion of faster, storage-class memories is only one part of the solution to this problem. The second part consists of the development of appropriate software abstractions such as data services and I/O frameworks. This latter part is particularly challenging to get right — the integration of these new services: (1) should ensure high performance of the resulting coupled application, (2) should present a unified interface that hides the complexity of programming the storage hardware, (3) should not hamper developer productivity, and (4) should not adversely affect the operation of existing legacy codes.

Fourth, performance tools are always the last to be updated. In other words, their evolution has always *followed* the applications they measure, instead of *co-evolving* with the application itself. For example, the CCA specification in its initial form did not appear to have any special provision for the design of CCA-capable performance tools. Instead, the tool community invented clever ways to integrate performance measurements through component proxies seamlessly. Aside from SCIRun2 [64] and Uintah [65], no other CCA framework considered performance optimization as a first-class design requirement. However, there are indications that this is changing. The MPI and OpenMP communities have recognized the need to tightly integrate performance tools by including a "tools section" in their respective specifications.

A fifth, bold prediction can be made by comparing the evolution of HPC software architectures with the respective changes within the general computing industry. The industry has shifted from an ESB-style tightly-coupled model to a more loosely-coupled services model where each service is highly cohesive in terms of functionality and can be independently updated from other services or components. This paradigm shift has increased the scalability of the overall application and allowed for faster, dynamic updates to service functionality without hampering other distributed components. If the initial signs [92], [93] are anything to go by, HPC software architectures are likely to resemble their industry counterparts in the future.

B. Open Areas

The trends discussed in this section naturally point to some critical open areas for future tool development. Table X presents a brief overview of the level of tool support for the different classes of distributed HPC frameworks discussed in this document.

1) Performance Instrumentation & Measurement: As discussed in Section VI-F, relatively few techniques exist to instrument and measure HPC data service performance. Their key interactions cannot be captured using existing techniques such as compiler instrumentation or PMPI-based library interposition. There are two reasons for this. One, these services typically do not use MPI for communication. Instead, the data services considered in this study primarily rely on RPC for passing control between coupled components. Traditional HPC performance tools are not designed to handle a client-server architecture. Second, and more importantly, control is passed *between* two or more distributed components. In the case of a microservice architecture such as Mochi [93], RPC calls can span microservices running on different computing nodes. Tracking these microservice interactions ("callpaths") through the system requires HPC tools to borrow some ideas from the cloud-based performance tools. At the same time, these microservice callpaths need to be annotated with *context* to associate performance inefficiencies occurring at lower levels in the software stack with higher-level interactions. Thus, a careful application of a combination of ideas borrowed from decades of HPC tool research with novel distributed request-tracing techniques can be a practical approach.

Regarding performance measurement, HPC ensembles only partially succumb to the application of existing HPC performance tools. For example, applying traditional PMPI-based measurement techniques to a Swift/T [122] workflow execution can yield information about the data transfers between individual Swift/T tasks. However, little information can be gathered this way about the execution details of individual Swift/T tasks. Individual tasks need some way of exchanging their identity with the performance tool so that the tool's measurement infrastructure can separate the performance events belonging to the task from the performance events belonging to the underlying Turbine [123] runtime.

2) Performance Monitoring & Analysis: While several robust performance monitoring tools such as LDMS [153] exist, these tools primarily target the monitoring of hardware resources. Recently, tools such as SOS [152] have been developed to monitor and aggregate performance data simultaneously from multiple data sources. They can be broadly applied to monitor any distributed application. However, most existing monitoring tools collect the data, aggregate this data in a central location (database), and then optionally provide the ability for a user to analyze the data from within this central location.

While this technique can scale well when the volume of data aggregated is minimal, it may not work for newer types of applications such as data services. Specifically, HPC data services operate in a highly concurrent environment. Thus, they require monitoring, aggregation, and analysis of large volumes of event traces to detect performance inefficiencies. Given the large storage footprint of event tracing, there is a need to analyze data at the source, *before* the aggregation is performed. Monitoring tools need to be flexible enough to support this type of analysis.

3) Control & Adaptivity: Few tools exist that can dynamically control and guide the execution of a coupled application. Fewer (if any) tools offer the ability to do so automatically. While adaptive algorithms have been studied and developed for individual modules in isolation, the guided execution of a coupled application requires performance data to be captured from multiple sources, aggregated, and finally analyzed to result in a control decision.

Further, the question of *who* actuates the control mechanism is also essential. Currently, the power to make these control decisions rests with a human user [111], [144]. This solution may work well when the number of coupled modules is relatively small, and the timescales involved in the control loop are large. However, HPC ensembles and transient data services involve tens or hundreds of individual modules and tasks that complete in a short span. Thus, they may require an automatic control system that relies on predefined policies. Further, as depicted in Table VII, many existing frameworks need to be updated to support resource elasticity before they can reap the full benefits of a control infrastructure.

VIII. Conclusion

This document presented a novel narrative of the evolution of HPC software development methodologies and the accompanying changes in HPC performance tools. Modularization was identified as the recurring theme underlying major revolutions in HPC software development. This document categorized various emerging types of HPC frameworks and applications based on their composition model, resource allocation scheme, and data management strategies. A discussion of the various techniques that HPC performance tools have implemented to stay relevant was also presented. Finally, the document touched upon some trends and open areas informing future work.

References

[1] J. M. Wozniak, R. Jain, P. Balaprakash, J. Ozik, N. T. Collier, J. Bauer, F. Xia, T. Brettin, R. Stevens, J. Mohd-Yusof *et al.*, "Candle/supervisor: A workflow framework for machine learning applied to cancer research," *BMC bioinformatics*, vol. 19, no. 18, pp. 59–69, 2018.

[2] P. M. Kasson and S. Jha, "Adaptive ensemble simulations of biomolecules," *Current opinion in structural biology*, vol. 52, pp. 87–94, 2018.

[3] A. Malony, "Performance understanding and analysis for exascale data management workflows," Univ. of Oregon, Eugene, OR (United States), Tech. Rep., 2019.

[4] T. Ben-Nun, T. Gamblin, D. Hollman, H. Krishnan, and C. J. Newburn, "Workflows are the new applications: Challenges in performance, portability, and productivity," in *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 2020, pp. 57–69.

[5] B. Norris, J. Ray, R. Armstrong, L. C. McInnes, D. E. Bernholdt, W. R. Elwasif, A. D. Malony, and S. Shende, "Computational quality of service for scientific components," in *International Symposium on Component-Based Software Engineering*. Springer, 2004, pp. 264–271.

[6] W. Emmerich and N. Kaveh, "Component technologies: Java beans, com, corba, rmi, ejb and the corba component model," in *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, 2001, pp. 311–312.

[7] A. Malony, S. Shende, N. Trebon, J. Ray, R. Armstrong, C. Rasmussen, and M. Sottile, "Performance technology for parallel and distributed component software," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 2-4, pp. 117–141, 2005.

[8] S. S. Shende and A. D. Malony, "The tau parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[9] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[10] D. Boehme, T. Gamblin, D. Beckingsale, P.-T. Bremer, A. Gimenez, M. LeGendre, O. Pearce, and M. Schulz, "Caliper: performance introspection for hpc software stacks," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 550–560.

[11] "Price of hpc systems," https://qz.com/1301510.

[12] "Top500 list," https://www.top500.org/.

[13] G. F. Pfister, "An introduction to the infiniband architecture," *High performance mass storage and parallel I/O*, vol. 42, no. 617-632, p. 102, 2001.

[14] D. W. Walker and J. J. Dongarra, "Mpi: a standard message passing interface," *Supercomputer*, vol. 12, pp. 56–68, 1996.

[15] "Ecp proxy suite," https://proxyapps.exascaleproject.org/ecp-proxy-apps-suite.

[16] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow, "The nas parallel benchmarks 2.0," Technical Report NAS-95-020, NASA Ames Research Center, Tech. Rep., 1995.

[17] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel computing*, vol. 22, no. 6, pp. 789–828, 1996.

[18] J. Liu, J. Wu, and D. K. Panda, "High performance rdma-based mpi implementation over infiniband," *International Journal of Parallel Programming*, vol. 32, no. 3, pp. 167–198, 2004.

[19] G. Chrysos, "Intel® xeon phi™ coprocessor-the architecture," *Intel Whitepaper*, vol. 176, p. 43, 2014.

[20] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[21] A. Kukanov and M. J. Voss, "The foundations for scalable multi-core software in intel threading building blocks." *Intel Technology Journal*, vol. 11, no. 4, 2007.

[22] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads programming*. O'Reilly & Associates, Inc., 1996.

[23] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "Openacc—first experiences with real-world applications," in *European Conference on Parallel Processing*. Springer, 2012, pp. 859–870.

[24] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[25] A. Munshi, "The opencl specification," in *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, 2009, pp. 1–314.

[26] L. V. Kale and S. Krishnan, "Charm++ a portable concurrent object oriented system based on c++," in *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, 1993, pp. 91–108.

[27] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to upc and language specification," Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, Tech. Rep., 1999.

[28] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[29] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *Proceedings of the department of defense HPCMP users group conference*, vol. 710. Citeseer, 1999.

[30] A. E. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, "Ompt: An openmp tools application programming interface for performance analysis," in *International Workshop on OpenMP*. Springer, 2013, pp. 171–185.

[31] J. E. Kay, C. Deser, A. Phillips, A. Mai, C. Hannay, G. Strand, J. M. Arblaster, S. Bates, G. Danabasoglu, J. Edwards *et al.*, "The community earth system model (cesm) large ensemble project: A community resource for studying climate change in the presence of internal climate variability," *Bulletin of the American Meteorological Society*, vol. 96, no. 8, pp. 1333–1349, 2015.

[32] S. Plimpton, P. Crozier, and A. Thompson, "Lammps-large-scale atomic/molecular massively parallel simulator," *Sandia National Laboratories*, vol. 18, p. 43, 2007.

[33] S. Habib, A. Pope, H. Finkel, N. Frontiere, K. Heitmann, D. Daniel, P. Fasel, V. Morozov, G. Zagaris, T. Peterka *et al.*, "Hacc: Simulating sky surveys on state-of-the-art supercomputing architectures," *New Astronomy*, vol. 42, pp. 49–65, 2016.

[34] "Coral-1 benchmarks," https://asc.llnl.gov/coral-benchmarks.

[35] "Coral-2 benchmarks," https://asc.llnl.gov/coral-2-benchmarks.

[36] J. Y. Choi, C.-S. Chang, J. Dominski, S. Klasky, G. Merlo, E. Suchyta, M. Ainsworth, B. Allen, F. Cappello, M. Churchill *et al.*, "Coupling exascale multiphysics applications: Methods and lessons learned," in *2018 IEEE 14th International Conference on e-Science (e-Science)*. IEEE, 2018, pp. 442–452.

[37] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," *Present and ulterior software engineering*, pp. 195–216, 2017.

[38] C. Szyperski, D. Gruntz, and S. Murer, *Component software: beyond object-oriented programming*. Pearson Education, 2002.

[39] A. L. Pope, *The CORBA reference guide: understanding the common object request broker architecture*. Addison-Wesley Longman Publishing Co., Inc., 1998.

[40] R. Sessions, *COM and DCOM: Microsoft's vision for distributed objects*. John Wiley & Sons, Inc., 1997.

[41] D. E. Bernholdt, B. A. Allan, R. Armstrong, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. G. Epperly, M. Govindaraju *et al.*, "A component architecture for high-performance scientific computing," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 163–202, 2006.

[42] Y. Alexeev *et al.*, "Component-based software for high-performance scientific computing," *Journal of Physics: Conference Series*, vol. 16, no. 1, 2005.

[43] S. G. Parker, "A component-based architecture for parallel multi-physics pde simulation," *Future Generation Computer Systems*, vol. 22, no. 1-2, pp. 204–216, 2006.

[44] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski, "Toward a common component architecture for high-performance scientific computing," in *Proceedings. The Eighth International Symposium on High Performance Distributed Computing (Cat. No. 99TH8469)*. IEEE, 1999, pp. 115–124.

[45] R. Armstrong, G. Kumfert, L. C. McInnes, S. Parker, B. Allan, M. Sottile, T. Epperly, and T. Dahlgren, "The cca component model for high-performance scientific computing," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 2, pp. 215–229, 2006.

[46] T. G. Epperly, G. Kumfert, T. Dahlgren, D. Ebner, J. Leek, A. Prantl, and S. Kohn, "High-performance language interoperability for scientific computing through babel," *The International Journal of High Performance Computing Applications*, vol. 26, no. 3, pp. 260–274, 2012.

[47] F. Bertrand and R. Bramley, "Dca: A distributed cca framework based on mpi," in *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings.* IEEE, 2004, pp. 80–89.

[48] J. Ray, N. Trebon, R. C. Armstrong, S. Shende, and A. Malony, "Performance measurement and modeling of component applications in a high performance computing environment: A case study," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* IEEE, 2004, p. 95.

[49] N. Trebon, A. Morris, J. Ray, S. Shende, and A. Malony, "Performance modeling of component assemblies with tau," in *Compframe 2005 workshop, Atlanta*, 2005.

[50] B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt, and J. A. Kohl, "The cca core specification in a distributed memory spmd framework," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 5, pp. 323–345, 2002.

[51] A. Cleary, S. Kohn, S. G. Smith, and B. Smolinski, "Language interoperability mechanisms for high-performance scientific applications," in *Proceedings of the 1998 SIAM Workshop on Object-Oriented Methods for Interoperable Scientific and Engineering Computing*, vol. 99, 1999, pp. 30–39.

[52] N. Trebon, A. Morris, J. Ray, S. Shende, and A. D. Malony, "Performance modeling of component assemblies," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 5, pp. 685–696, 2007.

[53] P. Hovland, K. Keahey, L. McInnes, B. Norris, L. Diachin, and P. Raghavan, "A quality of service approach for high-performance numerical components," in *Proceedings of Workshop on QoS in Component-Based Software Engineering, Software Technologies Conference, Toulouse, France*, vol. 20, 2003.

[54] L. C. McInnes, J. Ray, R. Armstrong, T. L. Dahlgren, A. Malony, B. Norris, S. Shende, J. P. Kenny, and J. Steensland, "Computational quality of service for scientific cca applications: Composition, substitution, and reconfiguration," *Preprint ANL/MCS-P1326-0206, Argonne National Laboratory, Feb*, 2006.

[55] V. Bui, B. Norris, K. Huck, L. C. McInnes, L. Li, O. Hernandez, and B. Chapman, "A component infrastructure for performance and power modeling of parallel scientific applications," in *Proceedings of the 2008 compFrame/HPC-GECO workshop on Component based high performance*, 2008, pp. 1–11.

[56] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington, "Optimisation of component-based applications within a grid environment," in *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, 2001, pp. 30–30.

[57] J. Bigot, Z. Hou, C. Pérez, and V. Pichon, "A low level component model easing performance portability of hpc applications," *Computing*, vol. 96, no. 12, pp. 1115–1130, 2014.

[58] C. Perez and V. Lanore, "Towards reconfigurable hpc component models," in *2018 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2018, pp. 151–152.

[59] M. Malawski, D. Kurzyniec, and V. Sunderam, "Mocca-towards a distributed cca framework for metacomputing," in *19th IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2005, pp. 8–pp.

[60] R. Schmidt, S. Benkner, and M. Lucka, "A component plugin mechanism and framework for application web services," in *Towards next generation grids: proceedings of the CoreGRID Symposium 2007, August 27-28, Rennes, France*. Springer, 2007, p. 107.

[61] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, and M. Yechuri, "A component based services architecture for building distributed applications," in *Proceedings the Ninth International Symposium on High-Performance Distributed Computing*. IEEE, 2000, pp. 51–59.

[62] M. Govindaraju, M. J. Lewis, and K. Chiu, "Design and implementation issues for distributed cca framework interoperability," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 5, pp. 651–666, 2007.

[63] S. Krishnan and D. Gannon, "Xcat3: A framework for cca components as ogsa services," in *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings.* IEEE, 2004, pp. 90–97.

[64] K. Zhang, K. Damevski, V. Venkatachalapathy, and S. G. Parker, "Scirun2: A cca framework for high performance computing," in *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings.* IEEE, 2004, pp. 72–79.

[65] J. D. d. S. Germain, J. McCorquodale, S. G. Parker, and C. R. Johnson, "Uintah: A massively parallel problem solving environment," in *Proceedings the Ninth International Symposium on High-Performance Distributed Computing*. IEEE, 2000, pp. 33–41.

[66] J. V. Reynders Iii, J. Cummings, and P. F. Dubois, "The pooma framework," *Computers in Physics*, vol. 12, no. 5, pp. 453–459, 1998.

[67] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, "Petsc," *See http://www. mcs. anl. gov/petsc*, 2001.

[68] R. D. Falgout and U. M. Yang, "hypre: A library of high performance preconditioners," in *International Conference on Computational Science*. Springer, 2002, pp. 632–641.

[69] M. Parashar and J. C. Browne, "Systems engineering for high performance computing software: The hdda/dagh infrastructure for implementation of parallel structured adaptive mesh," in *Structured adaptive mesh refinement (SAMR) grid methods*. Springer, 2000, pp. 1–18.

[70] D. L. Brown, G. S. Chesshire, W. D. Henshaw, and D. J. Quinlan, "Overture: An object-oriented software system for solving partial

differential equations in serial and parallel environments," Los Alamos National Lab., NM (United States), Tech. Rep., 1997.

[71] J. M. Squyres and A. Lumsdaine, "A component architecture for lam/mpi," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2003, pp. 379–387.

[72] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine *et al.*, "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2004, pp. 97–104.

[73] R. Keller, G. Bosilca, G. Fagg, M. Resch, and J. J. Dongarra, "Implementation and usage of the peruse-interface in open mpi," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2006, pp. 347–355.

[74] T. Islam, K. Mohror, and M. Schulz, "Exploring the capabilities of the new mpi_t interface," in *Proceedings of the 21st European MPI Users' Group Meeting*, 2014, pp. 91–96.

[75] S. Ramesh, A. Mahéo, S. Shende, A. D. Malony, H. Subramoni, A. Ruhela, and D. K. D. Panda, "Mpi performance engineering with the mpi tool interface: the integration of mvapich and tau," *Parallel Computing*, vol. 77, pp. 19–37, 2018.

[76] E. Gallardo, J. Vienne, L. Fialho, P. Teller, and J. Browne, "Employing mpi_t in mpi advisor to optimize application performance," *The International Journal of High Performance Computing Applications*, vol. 32, no. 6, pp. 882–896, 2018.

[77] M.-A. Hermanns, N. T. Hjelm, M. Knobloch, K. Mohror, and M. Schulz, "The mpi_t events interface: An early evaluation and overview of the interface," *Parallel computing*, vol. 85, pp. 119–130, 2019.

[78] B. Mohr, A. D. Malony, S. Shende, and F. Wolf, "Design and prototype of a performance tool interface for openmp," *The Journal of Supercomputing*, vol. 23, no. 1, pp. 105–128, 2002.

[79] M. Itzkowitz and Y. Maruyama, "Hpc profiling with the sun studio™ performance tools," in *Tools for high performance computing 2009*. Springer, 2010, pp. 67–93.

[80] H. Jagode, A. Danalis, H. Anzt, and J. Dongarra, "Papi software-defined events for in-depth performance analysis," *The International Journal of High Performance Computing Applications*, vol. 33, no. 6, pp. 1113–1127, 2019.

[81] J. Logan, M. Ainsworth, C. Atkins, J. Chen, J. Y. Choi, J. Gu, J. M. Kress, G. Eisenhauer, B. Geveci, W. Godoy *et al.*, "Extending the publish/subscribe abstraction for high-performance i/o and data management at extreme scale," *Bulletin of the IEEE Technical Committee on Data Engineering*, vol. 43, no. 1, 2020.

[82] G. Merlo, S. Janhunen, F. Jenko, A. Bhattacharjee, C. Chang, J. Cheng, P. Davis, J. Dominski, K. Germaschewski, R. Hager *et al.*, "First coupled gene–xgc microturbulence simulations," *Physics of Plasmas*, vol. 28, no. 1, p. 012303, 2021.

[83] V. Sarkar, W. Harrod, and A. E. Snavely, "Software challenges in extreme scale systems," in *Journal of Physics: Conference Series*, vol. 180, no. 1. IOP Publishing, 2009, p. 012045.

[84] E. Deelman, T. Peterka, I. Altintas, C. D. Carothers, K. K. van Dam, K. Moreland, M. Parashar, L. Ramakrishnan, M. Taufer, and J. Vetter, "The future of scientific workflows," *The International Journal of High Performance Computing Applications*, vol. 32, no. 1, pp. 159–175, 2018.

[85] C. M. MacKenzie, K. Laskey, F. McCabe, P. F. Brown, R. Metz, and B. A. Hamilton, "Reference model for service oriented architecture 1.0," *OASIS standard*, vol. 12, no. S 18, 2006.

[86] D. A. Chappell, *Enterprise service bus*. " O'Reilly Media, Inc.", 2004.

[87] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible io and integration for scientific codes through the adaptable io system (adios)," in *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, 2008, pp. 15–24.

[88] C. Docan, M. Parashar, and S. Klasky, "Dataspaces: an interaction and coordination framework for coupled simulation workflows," *Cluster Computing*, vol. 15, no. 2, pp. 163–181, 2012.

[89] M. Dreher and T. Peterka, "Decaf: Decoupled dataflows for in situ high-performance workflows," Argonne National Lab.(ANL), Argonne, IL (United States), Tech. Rep., 2017.

[90] P. H. Beckman, P. K. Fasel, W. E. Humphrey, and S. M. Mniszewski, "Efficient coupling of parallel applications using paws," in *Proceedings. The Seventh International Symposium on High Performance Distributed Computing (Cat. No. 98TB100244)*. IEEE, 1998, pp. 215–222.

[91] D. Bernholdt, F. Bertrand, R. Bramley, K. Damevski, J. Kohl, S. Parker, and A. Sussman, ""mxn" parallel data redistribution research in the common component architecture (cca)."

[92] I. B. Peng, R. Gioiosa, G. Kestor, E. Laure, and S. Markidis, "Preparing hpc applications for the exascale era: A decoupling strategy," in *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, 2017, pp. 1–10.

[93] R. B. Ross, G. Amvrosiadis, P. Carns, C. D. Cranor, M. Dorier, K. Harms, G. Ganger, G. Gibson, S. K. Gutierrez, R. Latham *et al.*, "Mochi: Composing data services for high-performance computing environments," *Journal of Computer Science and Technology*, vol. 35, no. 1, pp. 121–144, 2020.

[94] R. Latham, R. Ross, and R. Thakur, "Can mpi be used for persistent parallel services?" in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2006, pp. 275–284.

[95] J. A. Zounmevo, D. Kimpe, R. Ross, and A. Afsahi, "Using mpi in high-performance computing services," in *Proceedings of the 20th European MPI Users' Group Meeting*, 2013, pp. 43–48.

[96] D. Zhao, Z. Zhang, X. Zhou, T. Li, K. Wang, D. Kimpe, P. Carns, R. Ross, and I. Raicu, "Fusionfs: Toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems," in *2014 IEEE international conference on big data (Big Data)*. IEEE, 2014, pp. 61–70.

[97] M.-A. Vef, N. Moti, T. Süß, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann, "Gekkofs-a temporary distributed file system for hpc applications," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 319–324.

[98] A. Moody, D. Sikich, N. Bass, M. J. Brim, C. Stanavige, H. Sim, J. Moore, T. Hutter, S. Boehm, K. Mohror *et al.*, "Unifyfs: A distributed burst buffer file system-0.1. 0," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2017.

[99] A. Anwar, Y. Cheng, H. Huang, J. Han, H. Sim, D. Lee, F. Douglis, and A. R. Butt, "Bespokv: Application tailored scale-out key-value stores," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 14–29.

[100] C. Ulmer, S. Mukherjee, G. Templet, S. Levy, J. Lofstead, P. Widener, T. Kordenbrock, and M. Lawson, "Faodel: Data management for next-generation application workflows," in *Proceedings of the 9th Workshop on Scientific Cloud Computing*, 2018, pp. 1–6.

[101] M. A. Sevilla, N. Watkins, I. Jimenez, P. Alvaro, S. Finkelstein, J. LeFevre, and C. Maltzahn, "Malacology: A programmable storage system," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 175–190.

[102] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 307–320.

[103] Martin Fowler, "Microservices," https://martinfowler.com/articles/microservices.html.

[104] O. Zimmermann, "Microservices tenets," *Computer Science-Research and Development*, vol. 32, no. 3, pp. 301–310, 2017.

[105] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018.

[106] J. Soumagne, D. Kimpe, J. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. Ross, "Mercury: Enabling remote procedure call for high-performance computing," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2013, pp. 1–8.

[107] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault *et al.*, "Argobots: A lightweight low-level threading and tasking framework," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 512–526, 2017.

[108] N. Buchanan, M. Dorier, D. Doyle, C. Green, J. Kowalkowski, R. Latham, A. Norman, M. Paterno, R. Ross, S. Sehrish *et al.*, "Supporting hep data as the exascale era approaches," Argonne National Lab.(ANL), Argonne, IL (United States); Fermi National ..., Tech. Rep., 2018.

[109] H. Childs, S. D. Ahern, J. Ahrens, A. C. Bauer, J. Bennett, E. W. Bethel, P.-T. Bremer, E. Brugger, J. Cottam, M. Dorier *et al.*, "A terminology for in situ visualization and analysis systems," *The International Journal of High Performance Computing Applications*, vol. 34, no. 6, pp. 676–691, 2020.

[110] A. C. Bauer, H. Abbasi, J. Ahrens, H. Childs, B. Geveci, S. Klasky, K. Moreland, P. O'Leary, V. Vishwanath, B. Whitlock *et al.*, "In situ methods, infrastructures, and applications on high performance computing platforms," in *Computer Graphics Forum*, vol. 35, no. 3. Wiley Online Library, 2016, pp. 577–597.

[111] P. Grosset, J. Pulido, and J. Ahrens, "Personalized in situ steering for analysis and visualization," in *ISAV'20 In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, 2020, pp. 1–6.

[112] U. Ayachit, A. Bauer, B. Geveci, P. O'Leary, K. Moreland, N. Fabian, and J. Mauldin, "Paraview catalyst: Enabling in situ data analysis and visualization," in *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, 2015, pp. 25–29.

[113] T. Kuhlen, R. Pajarola, and K. Zhou, "Parallel in situ coupling of simulation with a fully featured visualization system," in *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization (EGPGV)*, vol. 10. Eurographics Association Aire-la-Ville, Switzerland, 2011, pp. 101–109.

[114] U. Ayachit, B. Whitlock, M. Wolf, B. Loring, B. Geveci, D. Lonie, and E. W. Bethel, "The sensei generic in situ interface," in *2016 Second Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV)*. IEEE, 2016, pp. 40–44.

[115] E. Dirand, L. Colombet, and B. Raffin, "Tins: A task-based dynamic helper core strategy for in situ analytics," in *Asian Conference on Supercomputing Frontiers*. Springer, 2018, pp. 159–178.

[116] M. Larsen, J. Ahrens, U. Ayachit, E. Brugger, H. Childs, B. Geveci, and C. Harrison, "The alpine in situ infrastructure: Ascending from the ashes of strawman," in *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, 2017, pp. 42–46.

[117] D. Morozov and Z. Lukic, "Master of puppets: Cooperative multitasking for in situ processing," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2016, pp. 285–288.

[118] M. Dorier, R. Sisneros, T. Peterka, G. Antoniu, and D. Semeraro, "Damaris/viz: a nonintrusive, adaptable and user-friendly in situ visualization framework," in *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*. IEEE, 2013, pp. 67–75.

[119] M. Dorier, G. Antoniu, F. Cappello, M. Snir, R. Sisneros, O. Yildiz, S. Ibrahim, T. Peterka, and L. Orf, "Damaris: Addressing performance variability in data management for post-petascale simulations," *ACM Transactions on Parallel Computing (TOPC)*, vol. 3, no. 3, pp. 1–43, 2016.

[120] S. Brandon, D. Domyancic, J. Tannahill, D. Lucas, G. Christianson, J. McEnereny, and R. Klein, "Ensemble calculation via the llnl uq pipeline: A user's guide," *Tech. Rep. LLNL-SM-480999*, 2011.

[121] D. H. Ahn, N. Bass, A. Chu, J. Garlick, M. Grondona, S. Herbein, H. I. Ingólfsson, J. Koning, T. Patki, T. R. Scogland *et al.*, "Flux: Overcoming scheduling challenges for exascale workflows," *Future Generation Computer Systems*, vol. 110, pp. 202–213, 2020.

[122] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster, "Swift/t: Large-scale application composition via distributed-memory dataflow processing," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE, 2013, pp. 95–102.

[123] J. M. Wozniak, T. G. Armstrong, K. Maheshwari, E. L. Lusk, D. S. Katz, M. Wilde, and I. T. Foster, "Turbine: A distributed-memory dataflow engine for extreme-scale many-task applications," in *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, 2012, pp. 1–12.

[124] J. L. Peterson, R. Anirudh, K. Athey, B. Bay, P.-T. Bremer, V. Castillo, F. Di Natale, D. Fox, J. A. Gaffney, D. Hysom *et al.*, "Merlin: Enabling machine learning-ready hpc ensembles," *arXiv preprint arXiv:1912.02892*, 2019.

[125] A. Merzky, M. Santcroos, M. Turilli, and S. Jha, "Radical-pilot: Scalable execution of heterogeneous and dynamic workloads on supercomputers," *CoRR, abs/1512.08194*, 2015.

[126] C. Harrison, B. Ryujin, A. Kunen, J. Ciurej, K. Biagas, E. Brugger, A. Black, G. Zagaris, K. Weiss, M. Larsen *et al.*, "Conduit: Simplified data exchange for hpc simulations," 2019.

[127] P. Hintjens, *ZeroMQ: messaging for many applications*. " O'Reilly Media, Inc.", 2013.

[128]

[129] M. Dorier, O. Yildiz, T. Peterka, and R. Ross, "The challenges of elastic in situ analysis and visualization," in *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, 2019, pp. 23–28.

[130] A. Raveendran, T. Bicer, and G. Agrawal, "A framework for elastic execution of existing mpi programs," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE, 2011, pp. 940–947.

[131] C. Huang, O. Lawlor, and L. V. Kale, "Adaptive mpi," in *International workshop on languages and compilers for parallel computing*. Springer, 2003, pp. 306–322.

[132] D. Rajan, A. Canino, J. A. Izaguirre, and D. Thain, "Converting a high performance application to an elastic cloud application," in *2011 IEEE Third International Conference on Cloud Computing Technology and Science*. IEEE, 2011, pp. 383–390.

[133] L. Zhao and S. A. Jarvis, "Predictive performance modelling of parallel component compositions," *Cluster Computing*, vol. 10, no. 2, pp. 155–166, 2007.

[134] L. A. Drummond, J. Demmel, C. R. Mechoso, H. Robinson, K. Sklower, and J. A. Spahr, "A data broker for distributed computing environments," in *International Conference on Computational Science*. Springer, 2001, pp. 31–40.

[135] J. A. Kohl, T. Wilde, and D. E. Bernholdt, "Cumulvs: Interacting with high-performance scientific simulations, for visualization, steering and fault tolerance," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 255–285, 2006.

[136] L. Zhang, C. Docan, and M. Parashar, "The seine data coupling framework for parallel scientific applications," *Advanced Computational Infrastructures for Parallel and Distributed Adaptive Applications*, vol. 66, p. 283, 2010.

[137] J. Larson, R. Jacob, and E. Ong, "The model coupling toolkit: a new fortran90 toolkit for building multiphysics parallel coupled models," *The International Journal of High Performance Computing Applications*, vol. 19, no. 3, pp. 277–292, 2005.

[138] J.-Y. Lee and A. Sussman, "High performance communication between parallel programs," in *19th IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2005, pp. 8–pp.

[139] J. Dayal, D. Bratcher, G. Eisenhauer, K. Schwan, M. Wolf, X. Zhang, H. Abbasi, S. Klasky, and N. Podhorszki, "Flexpath: Type-based publish/subscribe system for large-scale science analytics," in *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2014, pp. 246–255.

[140] V. Vishwanath, M. Hereld, and M. E. Papka, "Toward simulation-time data analysis and i/o acceleration on leadership-class systems," in *2011 IEEE Symposium on Large Data Analysis and Visualization*. IEEE, 2011, pp. 9–14.

[141] G. Eisenhauer, M. Wolf, H. Abbasi, and K. Schwan, "Event-based systems: Opportunities and challenges at exascale," in *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, 2009, pp. 1–10.

[142] J. Allard, V. Gouranton, L. Lecointre, S. Limet, E. Melin, B. Raffin, and S. Robert, "Flowvr: a middleware for large scale virtual reality applications," in *European Conference on Parallel Processing*. Springer, 2004, pp. 497–505.

[143] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony *et al.*, "Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir," in *Tools for High Performance Computing 2011*. Springer, 2012, pp. 79–91.

[144] M. Wolf, J. Choi, G. Eisenhauer, S. Ethier, K. Huck, S. Klasky, J. Logan, A. Malony, C. Wood, J. Dominski *et al.*, "Scalable performance awareness for in situ scientific applications," in *2019 15th International Conference on eScience (eScience)*. IEEE, 2019, pp. 266–276.

[145] A. D. Malony, M. Larsen, K. A. Huck, C. Wood, S. Sane, and H. Childs, "When parallel performance measurement and analysis meets in situ analytics and visualization." in *PARCO*, 2019, pp. 521–530.

[146] A. D. Malony, S. Ramesh, K. Huck, N. Chaimov, and S. Shende, "A plugin architecture for the tau performance system," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–11.

[147] R. R. Sambasivan, R. Fonseca, I. Shafer, and G. R. Ganger, "So, you want to trace your distributed system," *Key design insights from years of practical experience. Parallel Data Lab*, 2014.

[148] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger, "Diagnosing

performance changes by comparing request flows." in *NSDI*, vol. 5, 2011, pp. 1–1.
[149] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," 2010.
[150] "Jaeger tracing," https://www.jaegertracing.io.
[151] X. Zhang, H. Abbasi, K. Huck, and A. D. Malony, "Wowmon: A machine learning-based profiler for self-adaptive instrumentation of scientific workflows," *Procedia Computer Science*, vol. 80, pp. 1507–1518, 2016.
[152] C. Wood, S. Sane, D. Ellsworth, A. Gimenez, K. Huck, T. Gamblin, and A. Malony, "A scalable observation system for introspection and in situ analytics," in *2016 5th workshop on extreme-scale programming tools (ESPT)*. IEEE, 2016, pp. 42–49.
[153] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden *et al.*, "The lightweight distributed metric service: a scalable infrastructure for continuous monitoring of large scale computing systems and applications," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 154–165.
[154] P. C. Roth, D. C. Arnold, and B. P. Miller, "Mrnet: A software-based multicast/reduction network for scalable tools," in *SC'03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. IEEE, 2003, pp. 21–21.
[155] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: design, implementation, and experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004.
[156] C. Kelly, S. Ha, K. Huck, H. Van Dam, L. Pouchard, G. Matyasfalvi, L. Tang, N. D'Imperio, W. Xu, S. Yoo *et al.*, "Chimbuko: A workflow-level scalable performance trace analysis tool," in *ISAV'20 In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, 2020, pp. 15–19.
[157] "Prometheus," https://prometheus.io/.
[158] "Graphite," https://graphiteapp.org/.
[159] K. A. Huck, A. Porterfield, N. Chaimov, H. Kaiser, A. D. Malony, T. Sterling, and R. Fowler, "An autonomic performance environment for exascale," *Supercomputing frontiers and innovations*, vol. 2, no. 3, pp. 49–66, 2015.
[160] W. Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko, J. Vetter, and N. Mallavarupu, "Falcon: On-line monitoring and steering of large-scale parallel programs," in *Proceedings Frontiers' 95. The Fifth Symposium on the Frontiers of Massively Parallel Computation*. IEEE, 1995, pp. 422–429.
[161] N. Cheriere, M. Dorier, G. Antoniu, S. M. Wild, S. Leyffer, and R. Ross, "Pufferscale: Rescaling hpc data services for high energy physics applications," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 2020, pp. 182–191.