Applying Cloud-Native Principles to Scientific Computations: An Experiment or a Trend?

Joanna Kosińska, *Member, IEEE*, Maciej Malawski, Krzysztof Maliszewski, Karol Zając, Sara Oliviero and Antonino Amedeo La Mattina.

Abstract—Cloud Computing has emerged as a transformative force in the software industry, offering scalability, flexibility, and cost-efficiency. Although its adoption has been widespread in various sectors, the scientific community has been slower to embrace this paradigm shift, particularly in the realm of High Performance Computing (HPC) applications. In this paper we investigate the feasibility and benefits of adopting cloud-native principles and technologies for MPI-based HPC applications. We propose a conceptual model for migrating scientific applications to cloud-native architectures. To validate our model, we perform practical migration of a real MPI-based HPC application to a cloud-native architecture using Kubernetes. Our evaluation demonstrates that cloud-native deployment can achieve comparable performance to traditional HPC environments while offering greater flexibility, scalability, and portability. We conclude by discussing the implications of our findings and outlining future research directions in this evolving field.

Index Terms—Cloud-native, containerization, orchestration, scientific applications, migration, High-Performance Computing (HPC).

1 Introduction

In the contemporary landscape of the software industry, cloud computing has transitioned from a novel concept to an ubiquitous standard [1], [2]. This paradigm shift has revolutionized the way applications are developed, deployed, and scaled. Although the adoption of cloud computing has been rapid in various sectors, the scientific community has exhibited a more gradual embrace of this transformative technology.

Using public cloud services through vendor APIs raises concerns about vendor lock-in. The integration of proprietary APIs can create dependencies that make transitioning from a specific cloud provider difficult. Platform independence and vendor neutrality lie at the heart of the concept of cloud-native computing.

Cloud-native computing [3] represents a paradigm shift in the way applications are designed, developed, and deployed. In its essence, cloud-native embraces a set of principles and practices that optimize applications for the dynamic and distributed nature of modern cloud environments. Cloud-native applications (CNAs) are characterized by their modular and containerized architecture, enabling efficient scaling, resilience, and ease of deployment. Despite the widespread adoption of cloud-native principles in the software industry [4], its applications in the realm of scientific computing remain relatively unexplored. The unique challenges and requirements of scientific applications require a nuanced exploration of how cloud-native methodologies can be adapted to improve the efficiency, scalability, and reproducibility of scientific workloads. This research aims to bridge this gap by investigating the integration of cloud-native principles within scientific computing, shedding light on the untapped potential and opportunities that lie at the intersection of cloud-native architectures and scientific research.

The main objective of our research is to investigate the feasibility of adopting the cloud-native paradigm in the deployment of MPI-based high-performance computing applications. This endeavor is motivated by the growing trend of cloud computing usage in the context of scientific applications. To achieve this overarching goal, the primary contributions of this work are as follows:

1) Comprehensive review and analysis of existing solutions and approaches for containerizing and orchestrating scientific applications in cloud environments.
2) Conceptual model for migrating scientific applications, particularly MPI-based HPC workloads, to cloud-native architectures.
3) Practical evaluation of the proposed model by migrating a representative MPI-based HPC application, BoneStrength [5], to a cloud-native architecture using Docker and Kubernetes.

This manuscript is structured into five sections. In Section 2, we conduct an extensive review of related works and examine the current state of the art in the intersection of cloud-native computing and scientific applications. Section 3 presents our concepts regarding the migration of scientific workloads to cloud-native. Section 4 details

- *Joanna Kosińska, Krzysztof Maliszewski and Maciej Malawski are with AGH University of Krakow, Faculty of Computer Science, al. A. Mickiewicza 30, 30-059 Krakow, Poland. E-mail: kosinska,malawski@agh.edu.pl,maliszewski@student.agh.edu.pl*
- *Maciej Malawski and Karol Zając are with Sano Centre for Computational Medicine (https://sano.science), Extreme-scale Data and Computing Team, Czarnowiejska 36 building C5, 30-054 Kraków, Poland. E-mail: m.malawski,k.zajac@sanoscience.org*
- *Sara Oliviero is with Department of Industrial Engineering, Alma Mater Studiorum, University of Bologna, Via Zamboni, 33-40126 Bologna, Italy. E-mail: sara.oliviero@unibo.it*
- *Antonino Amedeo La Mattina is with Medical Technology Lab, IRCCS Istituto Ortopedico Rizzoli, Via di Barbiano 1/10 - 40136 Bologna, Italy E-mail: antoninoamedeo.lamattina@ior.it*

Manuscript received April 19, 2020; revised August 26, 2021.

the practical migration of a representative MPI-based HPC application to a cloud-native architecture, following the principles established in the conceptual model. Finally, Section 5 provides a critical discussion of the migration results, drawing conclusions about the feasibility and benefits of the cloud-native paradigm in scientific computing, and identifying potential paths for future research.

2 Related work

The term "scientific applications" encompasses a wide spectrum of software designed to support computational processes in various scientific disciplines [6]. In this paper, we focus primarily on the domain of High-Performance Computing (HPC). An HPC application is designed to harness the computational power of large-scale, parallel computing systems like clusters and supercomputers. These applications tackle complex and computationally intensive problems that would be impractical or impossible to solve on conventional computers. They are often characterized by their need for high computational performance, large amounts of data processing, and parallelization across multiple processing units [7]. HPC applications are often underpinned by the Message Passing Interface (MPI) standard. MPI is a standardized library and protocol designed for parallel programming in distributed-memory systems [8]. MPI provides a set of functions that enable processes running on multiple nodes of a cluster or supercomputer to communicate and coordinate their execution to solve large, computationally intensive problems.

Key differences between scientific HPC applications based on MPI and typical commercial applications include:

- Computational intensity: Scientific HPC applications are far more computationally demanding, requiring specialized hardware architectures and highly optimized numerical algorithms.
- Data parallelism: MPI-based applications heavily leverage data parallelism, where the dataset is partitioned, and computations are performed simultaneously on multiple compute nodes.
- Tight coupling: Processes within an MPI application often need frequent and coordinated communication to exchange data and synchronize their progress during computation.
- Performance sensitivity: Any overhead or latency in communication can profoundly impact the overall performance and scalability of HPC applications.

2.1 Cloud technologies

Cloud Computing (CC) has been an interest of the scientific community since its inception [9]. However, adoption within scientific computing has unfolded at a measured pace. Although cloud computing is unlikely to entirely replace on-premise solutions in scientific computing, it is increasingly recognized as a pivotal complement that propels contemporary research endeavors. It offers an array of advantages in the scientific realm:

1) Elimination of upfront investment in hardware, making advanced computational resources more accessible and affordable to a wider range of organizations.
2) The cost-effective pay-per-use model allows organizations to experiment with new technologies and evaluate their impact on scientific processes. Then seamlessly scale operations as needed.
3) Rapid scalability responds to fluctuating computational demands.
4) Immediate and accessible resources ensure that researchers can focus more on their work and less on infrastructure logistics.
5) Cutting-edge hardware and software features place cloud-native scientific applications at the forefront of innovation.
6) Infinite scalability in compute and storage ensures that researchers are not constrained by the limitations of on-premise infrastructure, fostering ambitious and expansive scientific pursuits.
7) Built-in security models ensures that scientific applications hosted in the cloud adhere to the highest standards of data integrity and confidentiality.

Recognizing the potential of cloud computing in scientific applications, major public cloud vendors have actively embraced the integration of cloud services to support high performance computing workloads. These cloud providers offer a comprehensive suite of services and features tailored to the unique demands of scientific research including:

- specialized instances,
- advanced cluster management tools,
- high performance networking,
- high performance serverless services.

2.2 The shift to cloud-native

On top of CC, the cloud-native philosophy has emerged. Many applications that we use every day like Facebook, Twitter, Netflix, Microsoft Office, or Google collaboration tools are indeed cloud-native. The Cloud Native Computing Foundation (CNCF) [10] plays a significant role in advancing cloud-native technologies and practices. As a neutral home for open-source projects, CNCF fosters collaboration, standardization, and the development of tools and frameworks essential for building and managing CNAs. Key attributes of cloud-native applications include:

1) Containerization [11]: Applications and their dependencies are encapsulated within lightweight, portable containers, ensuring consistent runtime environments across diverse infrastructures.
2) Orchestration [12]: Automated tools orchestrate the deployment, scaling, health monitoring, and management of containerized applications. They ensure high availability, fault tolerance, and optimal resource utilization by dynamically allocating resources, restarting failed containers, and load-balancing traffic.
3) DevOps Practices [13]: Foster collaboration between the development and operations teams through shared tools, processes, and culture.
4) Continuous Delivery and Integration [14]: Continuous integration and continuous delivery (CI/CD) pipelines automate the testing, integration, and deployment of application updates, enhancing speed and reliability.

2.3 Requirements of containers

Containers, a foundational element of cloud-native computing strategies, leverage operating-system level virtualization to create isolated, self-contained application environments. This isolation promotes reproducibility, compatibility, and simplified deployment in various computing environments [15]. The benefits of containers in scientific computing are numerous:

1) Bring your own environment (BYOE) [16] including specialized software stacks and configurations. This flexibility eliminates compatibility issues and ensures consistent execution of research workflows.
2) Reproducible science [17] by packaging entire scientific workflows, including data, scripts, and software dependencies, into self-contained units.
3) Supporting commercially certified code [18] that is designed to function exclusively in designated operating system versions.
4) Maintaining static environments (Software Appliances) [18] ensures long-term stability and reproducibility for legacy code or applications with specific hardware or software requirements.
5) Running legacy code on old operating systems [19] extends the lifespan of legacy software and allows researchers to continue utilizing valuable tools.
6) Management of complicated software stacks [15] streamlines deployment, update, and testing processes.
7) Complex workflows [20] consolidated in containers facilitate the sharing of knowledge and advancements in scientific research.

Despite the above benefits, containerization introduces some overhead. Although studies [16] [15] [21] show that containers can achieve near bare-metal performance in many scenarios, there is always a trade-off to consider between raw performance and the convenience and portability offered by containers. This trade-off becomes particularly important for MPI applications where performance is paramount. These applications often leverage several advanced solutions to improve efficiency of the communication, including:

- Specialized kernel modules (e.g., xpmem [22], knem [23]): These modules provide optimized memory management functionalities within the kernel, enabling faster data movement between processes.
- Specialized network hardware (e.g., Infiniband [24]): These high-bandwidth, low-latency network fabrics are specifically designed for high-performance computing environments, facilitating faster communication between compute nodes.
- Remote Direct Memory Access (RDMA) [25]: This hardware technology allows nodes to directly exchange data in main memory without involving the CPU or operating system of either node, significantly reducing communication overhead.
- Kernel bypass [26]: This technique partially or completely bypasses the kernel network stack, allowing data transfers to be initiated directly from user space. While kernel setup is still required, this approach can significantly improve communication performance.

Although the concept of OS-level virtualization has been around for some time, numerous technologies now implement containerization. Docker [27] has undoubtedly emerged as the industry-leading container engine, offering robust cross-platform compatibility. Traditionally, running Docker applications requires root-level privileges. This poses potential security risks and is generally unacceptable in HPC environments where multi-user systems are the norm. Although Docker's latest updates now support rootless mode, some security concerns remain. Additionally, the configuration required for rootless Docker remains complex, demanding careful namespace setup by system administrators to ensure proper resource isolation and security. To address these concerns, several alternative container engines have emerged including Singularity [28], Shifter [29], and Charliecloud [30]. These engines share key characteristics that make them more suitable for HPC:

- non-root operation,
- docker image compatibility,
- simplified networking,
- single-file/directory images.

Security concerns surrounding Docker are less pronounced in cloud environments, where users often operate virtual machines with root-level control. This, combined with Docker's widespread adoption and rich feature set, has solidified its position as the leading containerization tool in cloud computing. Moreover, root access within cloud environments grants users additional flexibility such as mounting host filesystems into containers for specific use cases or employing alternative security modules for fine-grained container isolation. Most importantly, Docker allows for integration with cloud orchestrators, which is discussed in the next subsection.

2.4 Orchestrating the workloads

Container orchestrators abstract away the complexity of the underlying infrastructure and provide a unified platform to manage containerized workloads at scale. They offer a range of capabilities including:

- Resource limit control by allowing to reserve CPU and memory for each container and hence providing valuable information for scheduling decisions.
- Scheduling based on predefined policies, which ensures efficient resource utilization.
- Load balancing that prevents bottlenecks.
- Health checks resulting in automatic restarting or replacement of faulty containers.
- Fault tolerance by ensuring that the desired number of containers is always running.
- Auto-scaling by automatic adding or removing containers.

In addition to these core capabilities, container orchestrators also simplify networking, enable service discovery, and support Continuous Integration/Continuous Delivery (CI/CD) [31].

Among the diverse landscape of container orchestrators, Kubernetes has firmly established itself as the de facto standard for CNAs deployment and orchestration [32]. In addition to supporting obvious mechanisms as parameterized redeployment in the event of failures, ensuring high

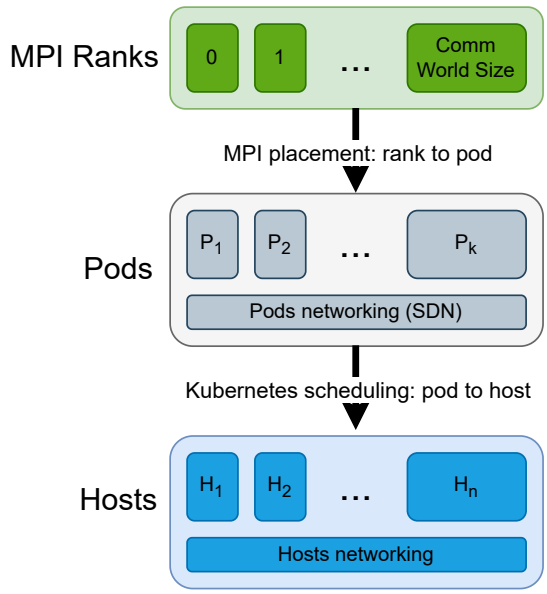

Figure 1. MPI Operator two layer rank mapping

availability, or sophisticated state management capabilities, Kubernetes offer other benefits in cloud environments [33] as:

- Cloud agnosticity: That means that Kubernetes can run on various cloud providers (e.g., AWS, Azure, Google Cloud) or on-premises infrastructure.
- Avoids VM overhead: Unlike traditional virtual machines (VMs), Kubernetes containers share the host operating system's kernel, resulting in significantly less overhead. This allows for faster scaling up and down of applications.
- Hybrid or multi-cloud deployments: This enables organizations to leverage the strengths of different cloud providers or maintain some workloads on-premises while migrating others to the cloud.
- Fine-grained control in multi-tenant environments: This ensures that applications from different users or teams can co-exist securely and efficiently on the same infrastructure.

Kubernetes provides a native resource type, called Job, specifically designed to manage batch workloads. Although management of this resource is efficient for loosely coupled or single-node workloads, they are not inherently designed for the specific requirements of HPC applications. The current standard for managing the lifecycle of MPI applications in Kubernetes is through the KubeFlow project's MPI Operator [34]. This operator streamlines the deployment and orchestration of MPI-over-Kubernetes instances, transforming the complex task of mapping MPI processes (ranks) to physical resources into a manageable two-layer process as depicted in Figure 1. The MPI Operator simplifies the deployment by handling various critical aspects such as:

1) Job creation: The operator creates Kubernetes jobs for each MPI worker, ensuring that the required number of worker processes is launched.
2) Communication setup: It establishes the necessary communication channels between MPI workers using the underlying SDN. This typically involves setting up a virtual network that allows workers to communicate efficiently.
3) Resource management: The operator manages the allocation of resources (CPU, memory, etc.) to the MPI workers.
4) Monitoring and logging: It provides mechanisms for monitoring the health and progress of the MPI job, as well as collecting logs for debugging and analysis.

Despite its capabilities, the MPI Operator may not be entirely reliable for large-scale HPC applications, particularly those involving more than a thousand ranks [35]. This limitation underscores the ongoing challenges in effectively running HPC applications in Kubernetes at extreme scales. The collaborative efforts of the Special Interest Groups (SIGs) dedicated to HPC interests and Kubernetes communities [36] demonstrate a strong commitment to making Kubernetes a viable platform for HPC applications, even on large scales.

2.5 Summary

The distinct characteristics of scientific HPC applications pose challenges and considerations when attempting to adapt them to cloud-native environments. With this in mind, we define the scientific cloud-native application as follows:

> DEFINITION 1.
> A **scientific cloud-native application** is an HPC application designed to leverage the scalability, elasticity, and resource management capabilities of cloud computing environments by packaging the application and its dependencies in containers and managing its deployment and scaling using container orchestration platforms.

3 MIGRATING WORKLOADS TO CLOUD-NATIVE

Transitioning scientific applications to a cloud-native paradigm requires a shift from a traditional job-centric perspective to a container-centric approach. In this section, we describe the design considerations for effectively migrating scientific applications to cloud-native architectures.

3.1 Containerizing computations

We propose three approaches for containerizing computations, which are as follows:

1) Fully containerized (basic): involves packaging of the entire MPI application within a container, including a basic MPI library. This method ensures that the application can run consistently in various environments. However, this comes at the cost of potentially suboptimal performance, as the basic MPI libraries may not fully utilize the underlying hardware capabilities.
2) Fully containerized (optimized): Attempts to maximize performance within the containerized environment by leveraging the hardware and network stack of the host system. This involves a more complex setup where the container is tailored to the specific HPC environment,

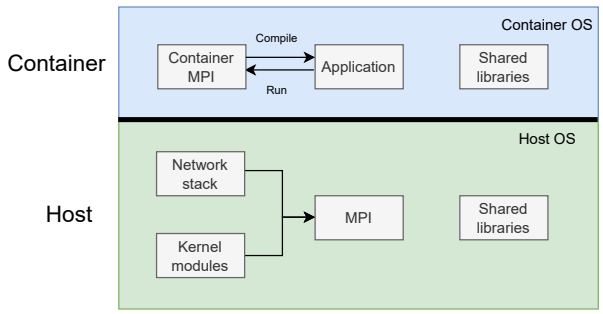

Figure 2. High-level design of fully containerized (basic) approach

including the installation of optimized MPI libraries and network configurations that align with the host system.

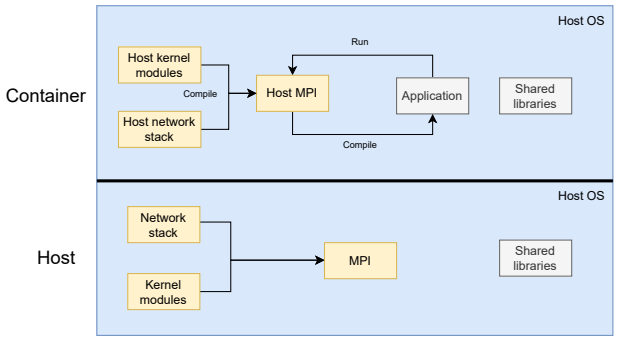

Figure 3. High-level design of fully containerized (optimized) approach

3) Hybrid container: Combines elements of containerization with traditional HPC resource management. In this model, the application is containerized with a basic MPI library, but during runtime it can utilize the optimized MPI installation of the host using techniques such as bind mounting. This approach provides a middle ground, offering better performance than the fully containerized (basic) approach while maintaining more flexibility and portability than the fully optimized method.

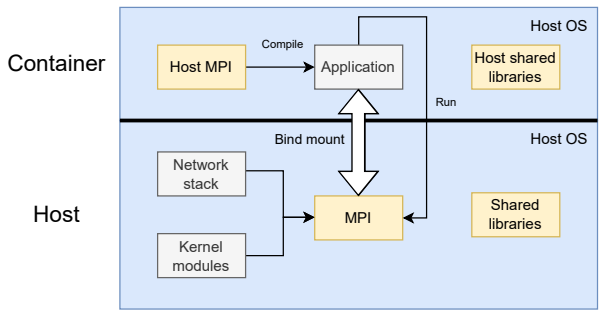

Figure 4. High-level design of a hybryd containerized approach

The decision of whether to utilize a nonoptimized MPI installation within a containerized environment can significantly impact application performance. However, the magnitude of this impact can vary greatly depending on several factors:

1) MPI usage frequency: If the application utilizes MPI communication infrequently the performance penalty associated with a basic MPI installation might be negligible.
2) Problem size and coupling: Larger and more tightly coupled problems that rely heavily on MPI communication are generally more susceptible to performance degradation when using a nonoptimized MPI implementation.

Table 1 summarizes the three approaches. Based on our experience, we highlight the pros and cons of our concepts and make some suggestions about the applicability of certain containerization approaches.

3.2 Orchestrating computations

Once containerized, scientific applications can be ported to Kubernetes. Adapting from traditional HPC workload managers to Kubernetes involves rethinking how applications are packaged, deployed, and managed, moving from a job-centric to a container-centric model. Table 2 provides our guidelines on how key HPC-specific features can be mapped to the Kubernetes container orchestration model.

4 Evaluation

This section details the practical migration and then evaluation to a cloud-native architecture. We conducted two experiments. Firstly, in 4.3, we evaluate the overall performance and the efficiency of communication. Secondly, in 4.4, we assess the scalability of cloud-native deployments.

4.1 Experimental setup

The evaluation was conducted in two computing environments:

1) Cloud-native environment orchestrated by Kubernetes
2) Traditional HPC environment managed by Slurm

Both environments shared an identical hardware configuration to isolate the performance impact of the software stack and deployment model. The tests were performed in a cloud environment, specifically on a private OpenStack [47] cloud cluster.

4.2 Example application

The investigation of the feasibility of migrating scientific applications to cloud-native environments we performed using a real application i.e. BoneStrength [5]. The application was developed by the University of Bologna. It is an In-Silico trial solution that estimates the number of fractures observed in a virtual cohort over a defined follow-up time. Figure 5 shows the simulation pipeline for a single patient. BoneStrength utilizes Ansys Mechanical APDL [48] to perform Finite Element simulations, using MPI to parallelize the process, and Python to analyze the results. The application is typically deployed using Slurm [49] sbatch scripts to run on HPC infrastructure.

Table 1
Comparison of containerization approaches for MPI applications

Approach	Advantages	Disadvantages	Recommended for
Fully containerized (basic)	Simplicity, Reproducibility	Potential performance limitations	Scenarios prioritizing portability and ease of use.
Fully containerized (optimized)	Potential for significant performance gains	Complexity, Limited portability across HPC systems	Performance-critical MPI applications where maximizing communication efficiency is essential.
Hybrid container	Performance improvements compared to basic approach, Better portability than fully optimized containers	Added complexity compared to basic approach, Dependency management challenges	Scenarios where performance is a concern, but a specific, vendor-supplied MPI library is required and cannot be easily containerized.

Table 2
Mapping HPC Concepts in Traditional Workload Managers into Kubernetes

HPC concept	Description	Kubernetes concept
Job submission	Specifies resources and scripts to execute.	YAML manifest
Resource allocation	Allocates resources (CPU, memory, GPUs).	Resource `requests` and `limits` in pod specifications. For GPU or other special hardware, use Node Feature Discovery [37] and device plugins.
Queueing and prioritization	Complex job scheduling.	`PriorityClass` and `ResourceQuota` with `LimitRange`.
Node selection	Control placement on specific nodes.	`nodeSelector`, (anti-)affinity rules, taints, and tolerations.
Workflow management	Built-in support for job dependencies. Workflow managers like Makeflow [38] or Pegasus [39] should be used for complex workflows.	`initContainers` allow running tasks before the main container starts. Workflow orchestration tools like Argo [40] or Kubeflow [41] should be used for complex pipelines.
Monitoring and reporting	Workload managers provide detailed reports on job execution, resource utilization, and accounting information.	The Kubernetes API provides rich information about pods, nodes, and resource usage. Third-party monitoring tools like Prometheus [42] and Grafana [43] can be used for visualizing metrics.
Security and user management	Managed at the cluster level, often through LDAP or Active Directory integration.	Implemention of *Role-Based Access Contro*l (RBAC) for fine-grained access control. Usage of *namespaces* to isolate resources and users.
File systems and data management	Shared file systems like NFS, Lustre, or GPFS for data storage and access.	Persistent Volumes (PVs) and Persistent Volume Claims (PVCs) for persistent storage. For shared filesystems, consider CSI drivers for NFS, Lustre, or other distributed file systems.
Software environment management	Modules and environment management tools.	Containers inherently encapsulate software environments. Container registries should be used for versioning and distribution.
Fault tolerance	Mechanisms for job checkpoint/restart.	Jobs with `restartPolicy` for automatic retries. Application-level checkpointing, storing state in Persistent Volumes.
Interactive jobs	Directly on compute nodes or through job submissions.	*Deployments* with services to expose interactive applications. Can also run Jupyter Notebooks [44] or other interactive tools in pods, providing web access through services or *Ingress*.
Array jobs	Manage a whole range of jobs with a single command.	Indexed Job with Static Work Assignment or Job Template Expansion patterns [45].

The migration process to cloud-native starts by containerizing the BoneStrength application. These steps involve:

1) Choosing a proper Docker image for the container. The next steps are accomplished in this container.
2) Installing Ansys dependencies.
3) Installing Python and data analysis libraries (e.g., NumPy, SciPy, Pandas).
4) Setting up SSH server (for Distributed Mode.
5) Creating a user and its context.

The subsequent steps involve mainly engineering tasks that lead to orchestration of the above-mentioned containers. Orchestration leverages Kubeflow's MPI Operator for management within a Kubernetes cluster, and optimization techniques to facilitate seamless execution in a cloud.

4.3 Performance of Cloud-native versus HPC approaches

In this experiment, we investigate whether a cloud-native deployment introduces significant overhead compared to a traditional HPC cluster setup. We evaluate the following performance metrics:

1) End-to-end execution time: Measures the total wall-clock time required to execute the benchmark application from job submission to completion.

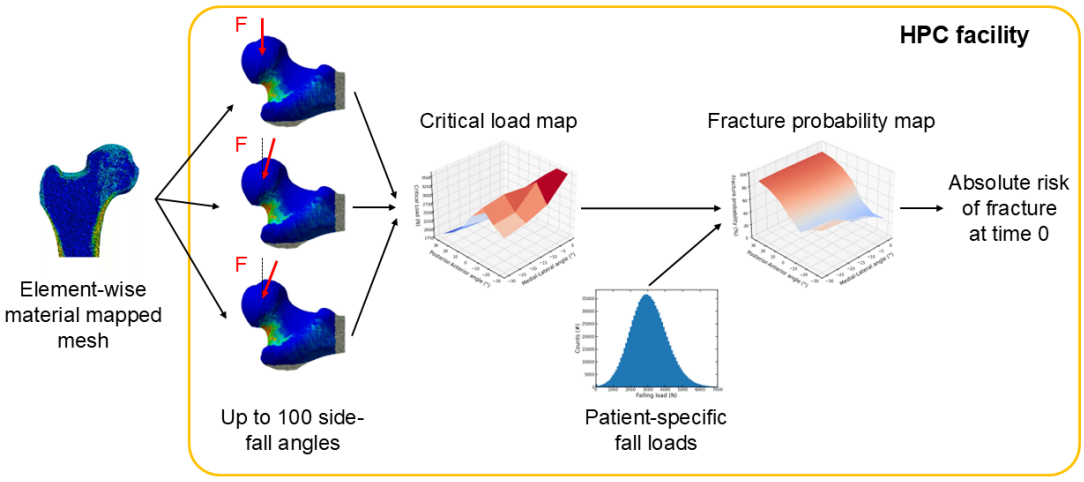

Figure 5. BoneStrength application visualization of the simulation pipeline for a single patient simulation [46]

Table 3
Comparison of BoneStrength metrics, i.e. end-to-end time and computational rate. The metrcs are measured in Kubernetes and Slurm under different distribution rates.

Distribution rate	Kubernetes		Slurm	
	Time (s)	Gflops	Time (s)	Gflops
1/1	1815.53 ± 10.2	3.0 ± 0.0	1782.4 ± 9.4	3.0 ± 0.0
1/2	1793.1 ± 5.9	3.63 ± 0.0	1782.9 ± 15.9	3.68 ± 0.0
1/4	1822.7 ± 25.5	4.03 ± 0.1	1790.6 ± 3.5	4.09 ± 0.0

Table 4
Comparison of BoneStrength inter-node and intra-node communication performance metrics between Kubernetes and Slurm.

		Kubernetes	Slurm
Latency (μs)	Inter-node	157.2 ± 12.6	130.25 ± 8.9
	Intra-node	3.67 ± 0.1	3.59 ± 0.1
Bandwidth (MB/sec)	Inter-node	382.61 ± 17.5	407.59 ± 9.7
	Intra-node	2265.28 ± 71.3	2322.03 ± 68.6

2) Computational rate (Gflops): Quantifies the raw floating-point operations per second achieved, reflecting the efficiency of CPU utilization.
3) Inter-node and intra-node communication: Latency and bandwidth measurements are collected to assess the communication performance within and between nodes.

We run simulations using a fixed number of four ranks. To evaluate the impact of varying communication patterns, we consider three distribution scenarios:

Distribution rate (1/1) – Single node execution. All MPI ranks are executed on a single node, minimizing inter-node communication.
Distribution rate (1/2) – 50% ranks per node. MPI ranks are evenly distributed across two nodes, introducing inter-node communication.
Distribution rate (1/4) – 25% ranks per node. MPI ranks are distributed across four nodes, maximizing inter-node communication.

Each scenario is executed ten times to ensure statistically significant results. As shown in the Tab. 3 and Fig. 6 the end-to-end time results for Kubernetes and Slurm under the different distribution scenarios show a consistent performance. The time differences between Kubernetes and Slurm are relatively small, with Kubernetes being slightly slower in each scenario. This slight increase in execution time on Kubernetes can be attributed to the overhead introduced by

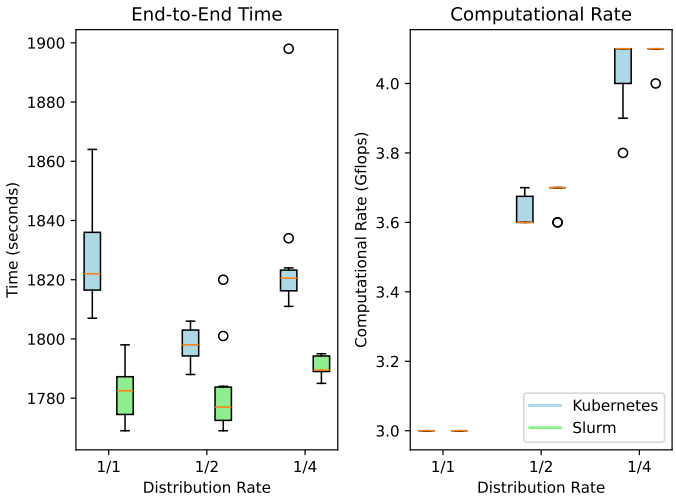

Figure 6. End-to-end time and computational rate by distribution rate comparison

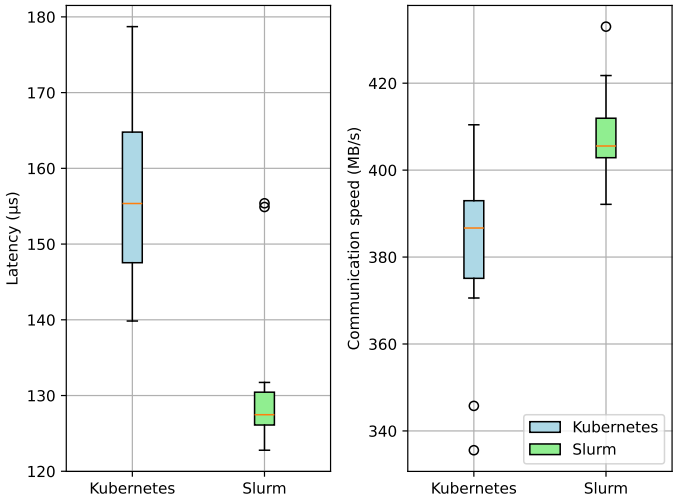

Figure 7. Inter-node communication efficiency comparison

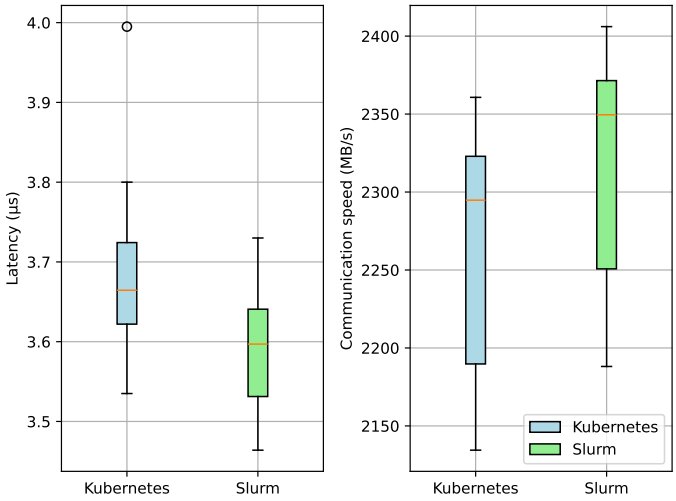

Figure 8. Intra-node communication efficiency comparison

containerization and orchestration. For the $1/1$ distribution rate, where all ranks are on a single node, the end-to-end time for Kubernetes is 1815.53 seconds compared to 1782.4 seconds on Slurm. The computational rates (Gflops) are identical in this scenario, indicating that the compute-intensive parts of the application perform similarly across both platforms. As we distribute the ranks more sparsely across nodes ($1/2$ and $1/4$ distribution rates), the end-to-end time for Kubernetes remains close to that of Slurm. However, the performance variability (standard deviation) increases, which is especially noticeable in the $1/4$ distribution scenario. This suggests that the network performance and communication overhead in Kubernetes might be less consistent compared to the more specialized HPC environment managed by Slurm. In terms of communication (Tab. 4, Fig. 7, Fig. 8), we can observe that Kubernetes exhibits slightly higher inter-node latency $(157.2\mu s \pm 12.6\mu s)$ compared to Slurm $(130.25\mu s \pm 8.9\mu s)$. This higher latency is expected due to the additional layers in the Kubernetes network stack, which introduce overhead. However, the inter-node bandwidth in Kubernetes (382.61 MB/sec ± 17.5 MB/sec) is only marginally lower than Slurm (407.59 MB/sec ± 9.7 MB/sec). This suggests that while Kubernetes might introduce some latency overhead, it does not significantly bottleneck the overall data transfer rate between nodes.

The intra-node latency and bandwidth remain virtually identical between Kubernetes and Slurm, indicating that the performance within a single node is not affected by the choice of orchestration platform.

The evaluation results demonstrate that the cloud-native deployment of BoneStrength on Kubernetes can achieve performance comparable to that of the traditional Slurm-based deployment on bare metal. Although minor differences in latency and end-to-end time are observed, they do not significantly impact the overall performance of the application, especially in scenarios with higher distribution rates. This suggests that migration to a cloud-native architecture using Kubernetes can be a viable option for scientific applications like BoneStrength, enabling greater flexibility and portability without sacrificing performance.

4.4 Scalability of Cloud-native versus HPC approaches

Although BoneStrength provides valuable insight into the performance of a real-world application, its fixed input size, limited by the finite element model of the patient's bone, restricts its use for evaluating scalability. Therefore, we utilize the HPCG (High Performance Conjugate Gradients) benchmark [50]. We analyze both weak (evaluates the ability to maintain performance while increasing the problem size proportionally to the number of processing units) and strong (assesses the ability to reduce execution time by distributing a fixed problem size across an increasing number of processing units, revealing the efficiency of parallelization) scalability. We primarily focus on the following HPCG metrics to evaluate scalability:

1) Time to solution: Measures the total wall-clock time required to solve the HPCG problem, from initialization to convergence. Lower values indicate better performance.
2) HPCG benchmark score: Represents the effective performance of the system, accounting for both computation and communication, measured in billions of floating-point operations per second (Gflops). Higher values indicate better performance.
3) Speedup (S): Calculated as the ratio of the execution time in a single processing unit to the execution time in multiple processing units. Speedup quantifies the performance gain achieved through parallelization. Ideally, it should be linear with the increase in processing units.
4) Performance speedup (PS): Calculated as the ratio of the HPCG benchmark score on a single processing unit to the execution time on multiple processing units.

The experiment is carried in two setups:

(A) – 1 worker instance with 16 vCPUs and 56 GB RAM,
(B) – 4 worker instances with 4 vCPUs and 14 GB RAM.

For the evaluation, the timed portion of the benchmark runs for 1 minute and each MPI process computes a solution for a cube of size $96 \times 128 \times 128$ points. For weak scaling scenarios, this size is adjusted accordingly to maintain a constant problem size per node. We run the benchmark

using 4, 8, 12, and 16 MPI ranks. To account for performance variations, we run the benchmark 10 times in each configuration, using the average values for analysis and comparison. In both cases, environments achieve near-linear speedup up to 8 ranks. However, beyond 8 ranks, the performance gains diminish, indicating that communication overhead starts to outweigh the benefits of parallelization. In strong scaling scenarios, where a fixed problem size is distributed across an increasing number of processing units, we expect to see a reduction in execution time. Ideally, we aim for linear speedup, where the performance gain is directly proportional to the increase in processing units.

The results (Tab. 5, Fig. 9) show that both Kubernetes and Slurm exhibit similar trends in reducing execution time as the number of ranks increases from 1 to 16. For instance, with 16 ranks on setup (A), Kubernetes achieves an execution time of 65.89 seconds, while Slurm achieves a slightly faster 62.648 seconds. The difference between the two platforms remains consistent across different configurations, suggesting that Kubernetes' cloud-native environment does not introduce significant overhead as the workload scales. Similarly, the benchmark score, which reflects overall performance by combining computation and communication efficiency, shows that Kubernetes maintains a performance comparable to Slurm. With 16 ranks in setup (A), Kubernetes scores 3.033 Gflops, close to Slurm's 3.188 Gflops. Both platforms show a similar trend in speedup as ranks increase, though Slurm consistently outperforms Kubernetes by a small margin. Despite this, the relative performance gain remains consistent, suggesting that Kubernetes can effectively parallelize workloads, although with a slight overhead. In case of performance speedup, Kubernetes values closely follow those of Slurm across all configurations, demonstrating that cloud-native deployments can scale effectively without significant performance degradation.

Looking at the weak scaling results in Tab. 6 and Fig. 10, we can see that both Kubernetes and Slurm maintain relatively stable execution times as the number of ranks increases. With 16 ranks in setup (A), Kubernetes completes the HPCG benchmark in 154.2 seconds, slightly slower than Slurm (148.687 seconds). This minor increase in execution time for Kubernetes is expected due to the overhead introduced by the cloud-native infrastructure but does not indicate a significant bottleneck. Similarly to strong scaling, the speedup and performance speedup metrics for weak scaling show that Kubernetes can handle increasing problem sizes with consistent gains, though not perfectly linear. At 16 ranks in setup (A), Kubernetes achieves a speedup of 0.67, slightly lower than Slurm (0.69), but still indicative of good scalability.

4.5 Summary

This section evaluated the migration of a representative MPI-based HPC application to a cloud-native architecture orchestrated by the MPIOperator on Kubernetes. Our findings demonstrate that cloud-native deployment can achieve performance comparable to traditional HPC clusters managed by Slurm. Although minor latency and end-to-end time differences were observed, they did not significantly impact overall application performance, particularly in scenarios with higher distribution rates. Furthermore, both strong and weak scaling experiments using the HPCG benchmark showcased Kubernetes capability to efficiently handle increasing computational workloads, achieving scaling efficiencies comparable to Slurm. Although minor performance differences were observed, the overall scaling behavior remained consistent, reinforcing Kubernetes viability for deploying scientific applications at scale.

5 Conclusions

In this paper, we explored the feasibility and benefits of adopting cloud-native principles and technologies in the context of scientific applications. We identified key design considerations and best practices for containerizing MPI applications, taking into account the unique requirements of HPC workloads, such as performance optimization, data management, and communication efficiency.

The evaluation results demonstrate that cloud-native Kubernetes deployments can achieve performance and scalability comparable to traditional HPC environments while offering greater flexibility and portability. While virtualization technologies and cloud platforms have traditionally been used to serve "long-tail science" applications which do not rely on high-performance or high-throughput computing, our results indicate that such environments may also offer competitive performance for massively parallel workflows, which have traditionally been the domain of HPC environments.

Although our research has yielded promising results, there are still open questions and areas for further exploration such as the following:

1) Advanced networking and hardware acceleration: Our experiments primarily utilized general-purpose machines with standard Ethernet interconnects. Further research should investigate the performance implications of leveraging advanced networking options (e.g., InfiniBand, RDMA) and specialized hardware accelerators (e.g., GPUs, TPUs) available in cloud environments.
2) Large-scale HPC workloads: It is crucial to explore the performance and scalability of cloud-native scientific applications on a true HPC scale, involving thousands of cores and massive datasets.
3) Security and compliance: Scientific data often requires stringent security and compliance measures. Further research should address data encryption, access controls, and compliance with regulatory requirements.

As cloud infrastructures continue to advance and mature, we can anticipate even greater synergy between HPC and cloud-native paradigms. This synergy will empower scientists and researchers with capable tools and resources to tackle complex scientific problems, accelerate discoveries, and drive innovation in various scientific disciplines. This comes on top of traditional advantages offered by virtualization, containerization and cloud technologies – including portability, reproducibility and overall reduction in technological debt related to maintaining distributed software platforms, including in the context of research. Accordingly, the answer to the titular question is that cloud-native principles are likely to become an established trend in scientific computing – alongside traditional HPC solutions.

Table 5
HPCG strong scalability comparison on Kubernetes and SLURM

Ranks	Setup	Kubernetes				SLURM			
		Time to solution (s)	HPCG score	S	PS	Time to solution (s)	HPCG score	S	PS
1	—	103.181	0.549	—	—	102.63	0.553	—	—
4	A	70.339	2.016	1.47	3.67	70.213	2.019	1.47	3.68
	B	60.460	1.876	1.71	3.42	59.628	1.904	1.73	3.47
8	A	64.101	3.112	1.61	5.67	60.271	3.307	1.71	6.02
	B	66.303	3.004	1.56	5.47	64.618	3.085	1.60	5.62
12	A	67.303	2.537	1.53	4.62	65.975	2.587	1.56	4.71
	B	64.226	2.217	1.61	4.04	64.932	2.409	1.59	4.39
16	A	65.890	3.033	1.57	5.53	62.648	3.188	1.65	5.81
	B	68.307	1.263	1.51	2.30	67.651	1.276	1.53	2.32

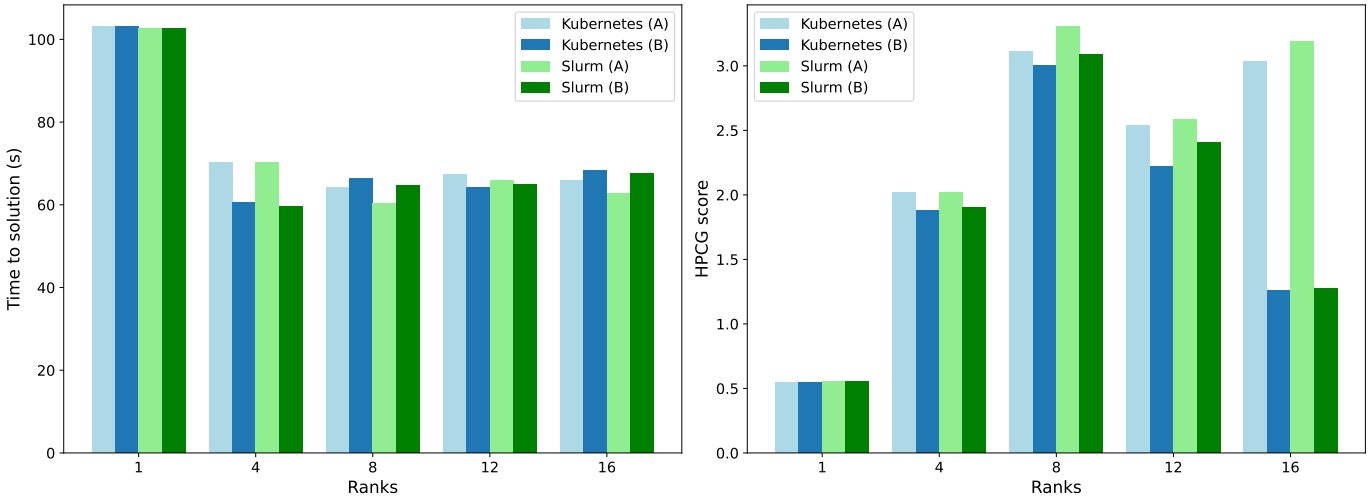

Figure 9. HPCG strong scalability comparison on Kubernetes and SLURM

Table 6
HPCG weak scalability comparison on Kubernetes and SLURM

Ranks	Setup	Kubernetes				SLURM			
		Time to solution (s)	HPCG score	S	PS	Time to solution (s)	HPCG score	S	PS
1	—	103.181	0.549	—	—	102.63	0.553	—	—
4	A	117.101	1.951	0.88	3.55	116.501	1.962	0.89	3.57
	B	117.243	1.949	0.88	3.55	116.433	1.963	0.89	3.58
8	A	77.597	2.966	1.33	5.40	73.915	3.112	1.40	5.67
	B	74.811	3.070	1.38	5.60	73.591	3.127	1.40	5.70
12	A	137.362	2.517	0.75	4.58	135.064	2.559	0.76	4.66
	B	138.212	2.501	0.75	4.56	136.187	2.538	0.76	4.62
16	A	154.200	2.996	0.67	5.46	148.687	3.106	0.69	5.66
	B	176.253	2.626	0.59	4.78	175.679	2.635	0.59	4.80

ACKNOWLEDGMENTS

The research presented in this paper was supported by the funds assigned to AGH University of Science and Technology by the Polish Ministry of Science and Higher Education.

This publication has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement Sano No 857533. The publication was created within the project of the Minister of Science and Higher Education "Support for the activity of Centers of Excellence established in Poland under Horizon 2020" on the basis of the contract number MEiN/2023/DIR/3796 and is supported by Sano project carried out within the International Research Agendas programme of the Foundation for Polish Science, co-financed by the European Union under the European Regional Development Fund.

REFERENCES

[1] T. Alam, "Cloud computing and its role in the information technology," *IAIC Transactions on Sustainable Digital Innovation (ITSDI)*, vol. 1, no. 2, pp. 108–115, 2020.

[2] W. Y. C. Wang, A. Rashid, and H.-M. Chuang, "Toward the trend of cloud computing," *Journal of Electronic Commerce Research*, vol. 12, no. 4, p. 238, 2011.

[3] C. Davis, *Cloud Native Patterns: Designing change-tolerant software*. Manning Publications, 2019.

[4] Y. Zhang, J. Han, J. Liu, M. Xian, H. Wang, Y. Chen, R. Zhang, and L. Zhang, "Cloud native technology development trend analysis research," in *6th International Workshop on Advanced Algorithms and Control Engineering (IWAACE 2022)*, vol. 12350. SPIE, 2022, pp. 345–350.

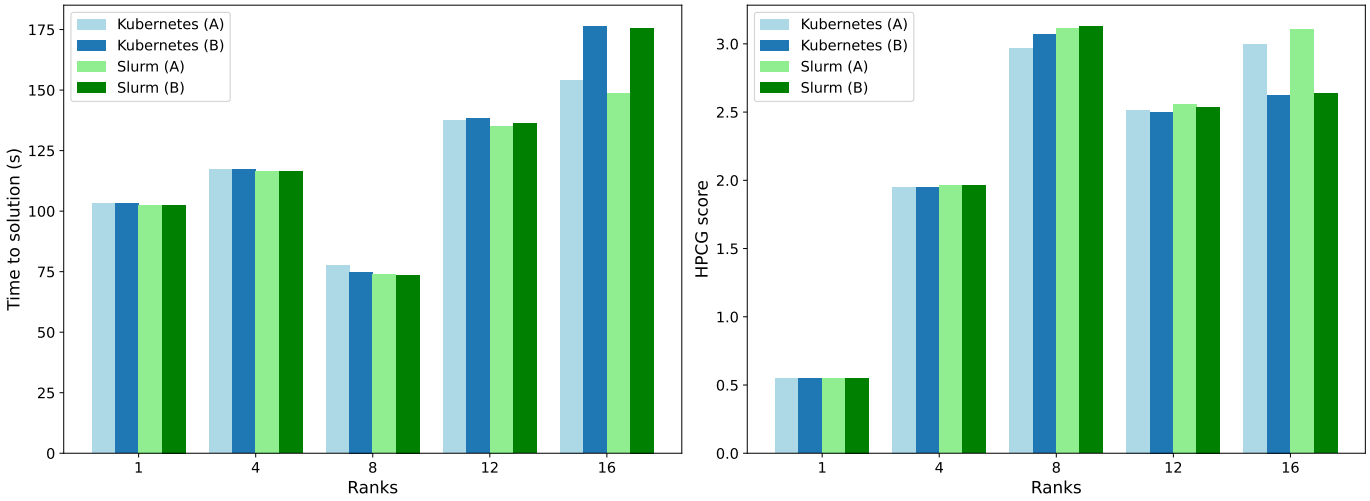

Figure 10. HPCG weak scalability comparison on Kubernetes and SLURM

[5] S. Oliviero, A. A. La Mattina, G. Savelli, and M. Viceconti, "In silico clinical trial to predict the efficacy of hip protectors for preventing hip fractures," *Journal of Biomechanics*, vol. 176, p. 112335, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0021929024004135

[6] R. F. Boisvert and P. Tang, "The architecture of scientific software, ifip tc2/wg2.5 working conference on the architecture of scientific software," 2001.

[7] T. Sterling, M. Brodowicz, and M. Anderson, *High performance computing: modern systems and practices*. Morgan Kaufmann, 2017.

[8] W. Gropp, S. Huss-Lederman, A. Lumsdaine, B. N. Ewing Lusk, W. Saphir, and M. Snir, *MPI: The Complete Reference: Volume 2, The MPI-2 Extensions*. MIT press, 1998.

[9] J. J. Rehr, J. P. Gardner, M. Prange, L. Svec, and F. Vila, "Scientific computing in the cloud," 2008.

[10] Cloud-native computing foundation. [Online]. Available: https://www.cncf.io/

[11] J. Watada, A. Roy, R. Kadikar, H. Pham, and B. Xu, "Emerging trends, techniques and open issues of containerization: A review," *IEEE Access*, vol. 7, pp. 152 443–152 472, 2019.

[12] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *ACM Queue*, vol. 14, pp. 70–93, 2016. [Online]. Available: http://queue.acm.org/detail.cfm?id=2898444

[13] G. Kim, J. Behr, and G. Spafford, *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution Press, 2016.

[14] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[15] L. Benedicic, F. A. Cruz, A. Madonna, and K. Mariotti, "Portable, high-performance containers for hpc," 2017. [Online]. Available: https://arxiv.org/abs/1704.03383

[16] A. J. Younge, K. T. Pedretti, R. E. Grant, and R. Brightwell, "A tale of two systems: Using containers to deploy hpc applications on supercomputers and clouds," *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 74–81, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:10848286

[17] C. Boettiger, "An introduction to docker for reproducible research," *SIGOPS Oper. Syst. Rev.*, vol. 49, no. 1, p. 71–79, 2015. [Online]. Available: https://doi.org/10.1145/2723872.2723882

[18] "Singularity documentation," https://docs.sylabs.io/guides/3.5/user-guide/introduction.html#why-use-containers.

[19] N. Poulton, *Docker Deep Dive*. Packt Publishing, 2020.

[20] S. Abraham, A. Paul, R. Khan, and A. Butt, "On the use of containers in high performance computing environments," 2020.

[21] P. Liu and J. Guitart, "Fine-grained scheduling for containerized hpc workloads in kubernetes clusters," 2022. [Online]. Available: https://arxiv.org/abs/2211.11487

[22] J. Vienne, "Benefits of cross memory attach for mpi libraries on hpc clusters," in *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*, 2014, pp. 1–6.

[23] B. Goglin and S. Moreaud, "KNEM: a generic and scalable kernel-assisted intra-node MPI communication framework," *Journal of Parallel and Distributed Computing*, vol. 73, no. 2, pp. 176–188, 2013.

[24] G. F. Pfister, "An introduction to the infiniband architecture," *High performance mass storage and parallel I/O*, vol. 42, no. 617-632, p. 10, 2001.

[25] R. J. Recio, P. R. Culley, D. Garcia, B. Metzler, and J. Hilland, "A Remote Direct Memory Access Protocol Specification," RFC 5040, Oct. 2007. [Online]. Available: https://www.rfc-editor.org/info/rfc5040

[26] R. Chen and G. Sun, "A survey of kernel-bypass techniques in network stack," in *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence*, 2018, pp. 474–477.

[27] Docker, Inc., *Docker Documentation*. [Online]. Available: https://docs.docker.com/

[28] Singularity. [Online]. Available: https://sylabs.io/singularity/

[29] Shifter. [Online]. Available: https://github.com/NERSC/shifter

[30] Charliecloud. [Online]. Available: https://hpc.github.io/charliecloud/

[31] N. Zhou, H. Zhou, and D. Hoppe, "Containerization for high performance computing systems: Survey and prospects," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, p. 2722–2740, Apr. 2023. [Online]. Available: http://dx.doi.org/10.1109/TSE.2022.3229221

[32] C. Carrión, "Kubernetes as a standard container orchestrator-a bibliometric analysis," *Journal of Grid Computing*, vol. 20, no. 4, p. 42, 2022.

[33] D. Y. Yuan and T. Wildish, "Bioinformatics application with kubeflow for batch processing in clouds," in *High Performance Computing*, H. Jagode, H. Anzt, G. Juckeland, and H. Ltaief, Eds. Cham: Springer International Publishing, 2020, pp. 355–367.

[34] K. Community, "Mpi operator," https://github.com/kubeflow/mpi-operator, 2024, kubernetes Operator for running MPI-based applications on Kubernetes.

[35] D. J. Milroy, C. Misale, G. Georgakoudis, T. Elengikal, A. Sarkar, M. Drocco, T. Patki, J. Yeom, C. E. A. Gutierrez, D. H. Ahn, and Y. Park, "One step closer to converged computing: Achieving scalability with cloud-native hpc," *2022 IEEE/ACM 4th International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC*, 2022.

[36] K. Community, "Kubernetes special interest groups (sigs)," 2024, accessed: 2024-08-12. [Online]. Available: https://github.com/kubernetes/community/tree/master/sig-list.md

[37] SIG Node, "Node feature discovery (nfd)," 2024, accessed: 2024-07-15. [Online]. Available: https://github.com/kubernetes-sigs/node-feature-discovery

[38] Cooperative Computing Lab, University of Notre Dame, "Makeflow," 2024, accessed: 2024-07-15. [Online]. Available: https://ccl.cse.nd.edu/software/makeflow/
[39] Pegasus WMS Team, "Pegasus workflow management system," 2024, accessed: 2024-07-15. [Online]. Available: https://pegasus.isi.edu
[40] Argo Project Contributors, "Argo workflows," 2024, accessed: 2024-07-15. [Online]. Available: https://argoproj.github.io/argo-workflows/
[41] Kubeflow Community, "Kubeflow," 2024, accessed: 2024-07-15. [Online]. Available: https://www.kubeflow.org
[42] "Prometheus," 2024, accessed: 2024-07-15. [Online]. Available: https://prometheus.io
[43] Grafana Labs, "Grafana," 2024, accessed: 2024-07-15. [Online]. Available: https://grafana.com
[44] Project Jupyter, "Jupyter notebook," 2024, accessed: 2024-07-15. [Online]. Available: https://jupyter.org
[45] Kubernetes job patterns. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/job/#job-patterns
[46] Simulation pipeline for a single patient simulation. Ansys. Accessed: 2024-07-28. [Online]. Available: https://images.ansys.com/is/image/ansys/simulation-pipeline-for-a-single-patient-simulation?wid=1347
[47] OpenStack Foundation, "Openstack." [Online]. Available: https://www.openstack.org/
[48] ANSYS, Inc., *ANSYS Mechanical APDL*, ANSYS, Inc., 2024, version 2024 R2, Accessed: 2024-08-29. [Online]. Available: https://www.ansys.com/products/structures/ansys-mechanical
[49] SchedMD LLC, *Slurm Workload Manager*. [Online]. Available: https://slurm.schedmd.com/documentation.html
[50] M. A. Heroux and J. Dongarra, "High performance conjugate gradients (hpcg) benchmark," Sandia National Laboratories and University of Tennessee, Tech. Rep., 2013, accessed: 2024-08-03. [Online]. Available: https://www.hpcg-benchmark.org/

Joanna Kosińska (M'20) is an Assistant Professor at the Faculty of Computer Science, AGH University of Krakow, Poland. She received her PhD degree in the Information and Communication Technology discipline. Her research interests focus on distributed computing, specifically Cloud-native computing and resource management.

Since 2020 she has been the CEO of Try IT Foundation. The Foundation aims to promote the discipline of IT among girls and women.

In 2022 and 2023 she was a visiting researcher at Sano Centre for Computational Medicine. The purpose of the internship was to find a new direction of research for Computational Medicine in the area of Cloud-native Computing and its observability.

Krzysztof Maliszewski is a master's student in computer science at the Faculty of Computer Science, AGH University of Krakow, Poland, and a software engineer at AVSystem where he works with cloud-native technologies on a daily basis, contributing to the development of innovative solutions. His primary interests revolve around distributed systems, particularly their performance optimization and automation.

Karol Zając holds a Master's degree in Computer Science. Currently works as a Scientific Programmer at Sano - Centre for Computational Medicine. His research is primarily focused on High-Performance Computing (HPC) and scientific workflows including VVUQ (Verification, Validation, and Uncertainty Quantification) processes. He has applied his expertise to adapt in silico computational models from the In Silico World (ISW) project onto HPC infrastructure, ensuring scalability and efficiency at a large-scale. Additionally, he has contributed to toolkit development supporting the automation of simulation campaigns for patient cohorts and data sharing. He supervised experiments, utilizing a total of 5 million CPU hours.

Maciej Malawski Maciej Malawski holds a PhD in computer science and an MSc in computer science and in physics. In 2011-2012 he was a postdoc at the University of Notre Dame, USA. Currently Director of Sano – Centre for Computational Medicine, Research Team Leader – Extreme-scale Data and Computing, associate professor at the Institute of Computer Science AGH University of Krakow and a senior researcher at Academic Computer Centre Cyfronet AGH. He has over 20 years of experience in research in parallel and distributed computing, high performance computing (HPC), grid and cloud technologies, serverless and container-based infrastructures, and federated machine learning. Interested in innovative applications of these technologies to scientific computing. His research interests include scientific workflows with focus on usage of novel and emerging large-scale computing infrastructures, performance evaluation, resource management, scheduling and cost optimization, with special focus on the use of advanced computing in medicine. He is a co-author of over 100 scientific papers, and has served community as a member of technical program committees of premier conferences on scientific, parallel and distributed computing (SC, ICCS, CCGrid, UCC, IPDPS), and was a general co-chair of EuroPar 2020. Since 2002 he participated in several European (CrossGrid, CoreGRID, ViroLab, VPH-Share, Eur-Valve, CECM, In Silico World, NearData) and nationally funded (NCN Opus, MEiN) research projects.

Sara Oliviero is a research fellow in the Department of Industrial Engineering at the University of Bologna (Italy). She received her PhD degree from the University of Sheffield (United Kingdom), where she worked on the non-invasive assessment of the densitometric and mechanical properties of the mouse tibia for preclinical applications. Her current research is focused on the development of an In Silico trial technology for the assessment of osteoporosis treatments.

Antonino Amedeo La Mattina is a researcher in the Medical Technology Lab at the Rizzoli Orthopaedic Institute in Bologna (Italy). He received his PhD degree in Information Engineering from the University of Pisa (Italy) in 2020. His research activity covers the development of Digital Twins for proximal femur fragility fracture risk predictions and In Silico Trials for osteoporosis treatments, with particular focus on efficient and secure health data processing.