applied sciences

MDPI

Article

HPC Cloud Architecture to Reduce HPC Workflow Complexity in Containerized Environments

Guohua Li [1], Joon Woo [1] and Sang Boem Lim [2,*]

[1] National Supercomputing Center, Korea Institute of Science and Technology Information, 245 Daehak-ro, Yuseong-gu, Daejeon 34141, Korea; ghlee@kisti.re.kr (G.L.); winadia@kisti.re.kr (J.W.)
[2] Department of Smart ICT Convergence, Konkuk University, 120 Neungdong-ro, Gwangjin-gu, Seoul 05029, Korea
* Correspondence: sblim@konkuk.ac.kr; Tel.: +82-2-450-3840

Abstract: The complexity of high-performance computing (HPC) workflows is an important issue in the provision of HPC cloud services in most national supercomputing centers. This complexity problem is especially critical because it affects HPC resource scalability, management efficiency, and convenience of use. To solve this problem, while exploiting the advantage of bare-metal-level high performance, container-based cloud solutions have been developed. However, various problems still exist, such as an isolated environment between HPC and the cloud, security issues, and workload management issues. We propose an architecture that reduces this complexity by using Docker and Singularity, which are the container platforms most often used in the HPC cloud field. This HPC cloud architecture integrates both image management and job management, which are the two main elements of HPC cloud workflows. To evaluate the serviceability and performance of the proposed architecture, we developed and implemented a platform in an HPC cluster experiment. Experimental results indicated that the proposed HPC cloud architecture can reduce complexity to provide supercomputing resource scalability, high performance, user convenience, various HPC applications, and management efficiency.

Keywords: HPC workflow; cloud platform; supercomputing; HPC cloud architecture; containerized environment

check for updates

Citation: Li, G.; Woo, J.; Lim, S.B. HPC Cloud Architecture to Reduce HPC Workflow Complexity in Containerized Environments. *Appl. Sci.* **2021**, *11*, 923. https://doi.org/10.3390/app11030923

Academic Editor: Bernabe Dorronsoro
Received: 19 November 2020
Accepted: 15 January 2021
Published: 20 January 2021

1. Introduction

Supercomputing services are currently among the most important services provided by national supercomputing centers worldwide [1]. This designation typically refers to the use of aggregated computing power to solve advanced computational problems related to scientific research [2]. Most of these services are composed of cluster-based supercomputers and are important for providing the computational resources and applications necessary in various scientific fields [3]. These supercomputing services are often divided into high-performance computing (HPC) services and high-throughput computing (HTC) services [4] according to the workflow of the job. An HPC service has a strong impact on jobs that are tightly coupled in parallel processing. These jobs (such as message passing interface (MPI) parallel jobs) must process large numbers of computations within a short time. In contrast, an HTC service involves independence between jobs that are loosely coupled in parallel processing. These jobs must process large and distributed numbers of computations over a specific period (such as a month or a year) [5].

Most national supercomputing centers are plagued with inconvenience and inefficiency in handling these types of jobs. Therefore, in this study, we focused on the current problems facing HPC users and supercomputing center administrators seeking to provide efficient systems. HPC users face four main problems when performing traditional HPC jobs; these are complexity, compatibility, application expansion, and a pay-as-used arrangement. Conversely, center administrators face five main problems with traditional HPC systems; these are cost, flexibility, scalability, integration, and portability.

The solution to these five problems with traditional HPC is to adopt cloud computing technology to provide HPC services in a cloud environment [6]. With the development of cloud computing technology and the stabilization of its service, the provision of HPC services in the cloud environment has become one of the main interests of many HPC administrators. Many researchers have begun to enter this field, using descriptors such as HPC over Cloud, HPC Cloud, HPC in Cloud, or HPC as a Service (HPCaaS) [7]. Although these approaches solve typical scalability problems in the cloud environment and reduce deployment and operational costs, they fail to meet high-performance requirements, because of the performance degradation of virtualization itself. In particular, the performance degradation in overlay networking solutions involving Virtual Machines (VMs) and networking virtualization has become a most serious issue. To address this, studies are currently underway to program the fast packet processing between VMs on the data plane using a data plane development kit [8]. Another fatal problem results from the inflexible mobility of VM images, which include various combinations or versions of, e.g., the operating system, compiler, library, and HPC applications. Thus, the biggest challenge to HPC administrators is to achieve scalability of the cloud, performance, and image service in virtual situations.

One solution to overcome this challenge is to implement an HPC cloud in a containerized environment. Integrating the existing HPC with a containerized environment leads to complexity in terms of designing the HPC workflow, which is defined as the flow of tasks that need to be executed to compute on HPC resources. The diversity of these tasks increases the complexity of the infrastructure to be implemented. We defined some requirements of an HPC workflow in a containerized environment and compared them with those of other projects such as Shifter [9], Sarus [10], EASEY [11], and JEDI [12], which are suggested by several national supercomputing centers. It is difficult to design an architecture that includes all functionalities that satisfy the requirements of users and administrators on the basis of these related research analyses.

In this study, we proposed an HPC cloud architecture that can reduce the complexity of HPC workflows in containerized environments to provide supercomputing resources scalability, high performance, user convenience, various HPC applications, and management efficiency. To evaluate the serviceability of our proposed architecture, we developed a platform that was part of the Partnership and Leadership of Supercomputing Infrastructure (PLSI) project [13] led by the Korean National Supercomputing Center, Korea Institute of Science and Technology Information (KISTI), and built a test bed based on the PLSI infrastructure. In addition, we developed a user-friendly platform that is easy to use and uses a minimal knowledge base interface to ensure user convenience.

The reminder of this paper is organized as follows: In Section 2, we describe related work on container-based HPC cloud solutions involving national supercomputing resources. In Section 3, the platform implemented is explained and information about system architecture and workflows is included, and the detailed method is described in Section 4. In Section 5, we present the results of evaluation applied to various aspects of the platform we developed. Finally, we provide concluding remarks in Section 6.

2. Related Work

To propose an HPC cloud architecture in a containerized environment, we first analyzed container solutions such as Docker [14], Singularity [15], and Charliecloud [16] that are suitable for an HPC cluster environment. Projects such as Shifter [9], Sarus [10], EASEY [11], and JEDI [12] provide HPC cloud services based on these container-based solutions. In these projects, we analyzed the HPC workflow from an architectural perspective. Next, we defined several requirements of the HPC workflow in a containerized environment and compared each of the projects with the proposed architecture.

2.1. Container Solutions

2.1.1. Docker

Docker is a lightweight container technology that easily bundles and runs libraries, packages, and applications in an isolated environment. Compared with VM-based virtualization technology, Docker is an isolation technology that shares the resources of the host to provide better performance than that of a VM. Nevertheless, the following problems exist in the HPC cluster environment with Docker technology [14]:

- Docker is not well integrated with other workload managers.
- It does not isolate users on shared file systems.
- Docker is designed without consideration of clients who do not have a local disk.
- The Docker network is a bridge type by default. Thus, to enable communication on multiple hosts, another overlay network solution is required.

2.1.2. Singularity

Singularity, which is a new container technology with a format different from that of the Docker image, was introduced in 2016 for the development of HPC services [15] for solving these problems. Singularity can be integrated with any resource manager on the host. For example, it is feasible to integrate it with resource managers such as HPC interconnects, scheduler, file system, and Graphics Processing Units (GPUs). Singularity also focuses on container mobility and can be run in any workload with no modification to any host. To provide a deeper understanding of the technology that implements this function, the architecture of the VM, Docker container, and Singularity are compared in Figure 1 [17].

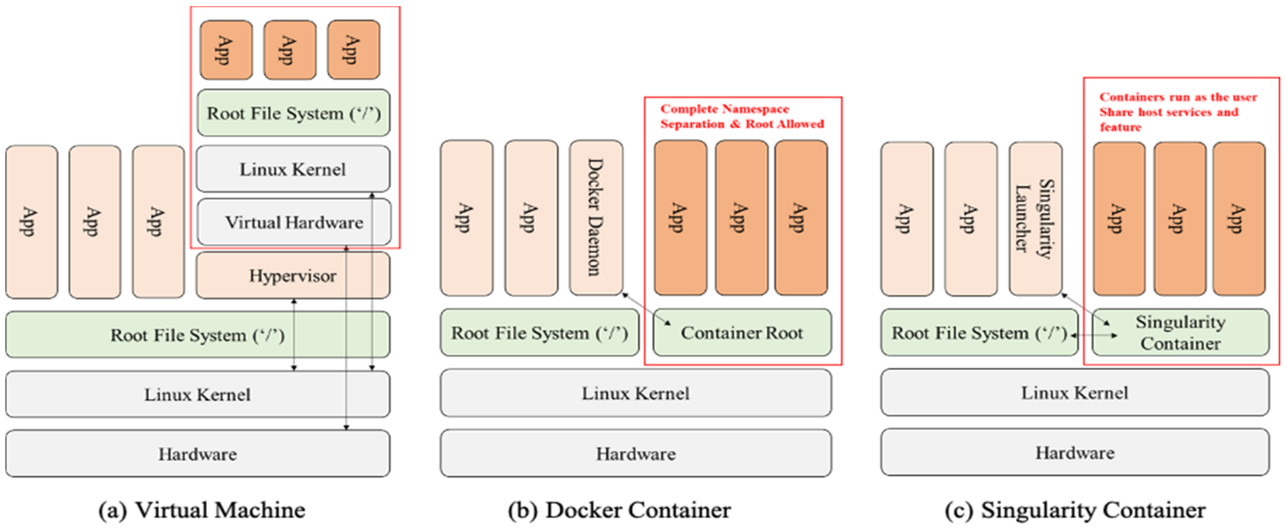


Figure 1. Architecture comparison of the Virtual Machine (**a**), Docker container (**b**), and Singularity (**c**).

2.1.3. Charliecloud

Charliecloud (developed at the Los Alamos National Laboratory) provides unprivileged containers for user-defined software stacks in HPC [16]. The motivation for developing Charliecloud was to meet the increasing demand at supercomputing centers for user-defined software stacks. These demands included complex dependencies or build requirements, externally required configurations, portability, consistency, and security. Charliecloud is designed as a lightweight, open source software stack based on the Linux user namespace. It uses Docker to build images, uses a shell script for automating configurations, and runs user code with the C program [16]. It also uses a host network like Singularity (described in the next section) to meet HPC performance standards.

2.2. *Container-Based HPC Cloud Solutions*

2.2.1. Shifter

Shifter is a user-defined container image solution developed by the National Energy Research Scientific Computer Center (NERSC) for HPC services. It has been implemented as a prototype on the Cray supercomputers at NERSC. It allows an HPC system to permit users to run a Docker image efficiently and safely. Shifter's workflow [18] is shown in Figure 2. It provides a user-defined image manager, workload management for resources, and a user interface to distribute containers in an HPC environment and provides HPC services to users. This solution covers various image types, as well as providing great detail for a container-based image management workflow. However, because the batch scheduler used in the traditional HPC field is used without integrating it with the container-based scheduler, the amount of automation for the job workflow is still limited.

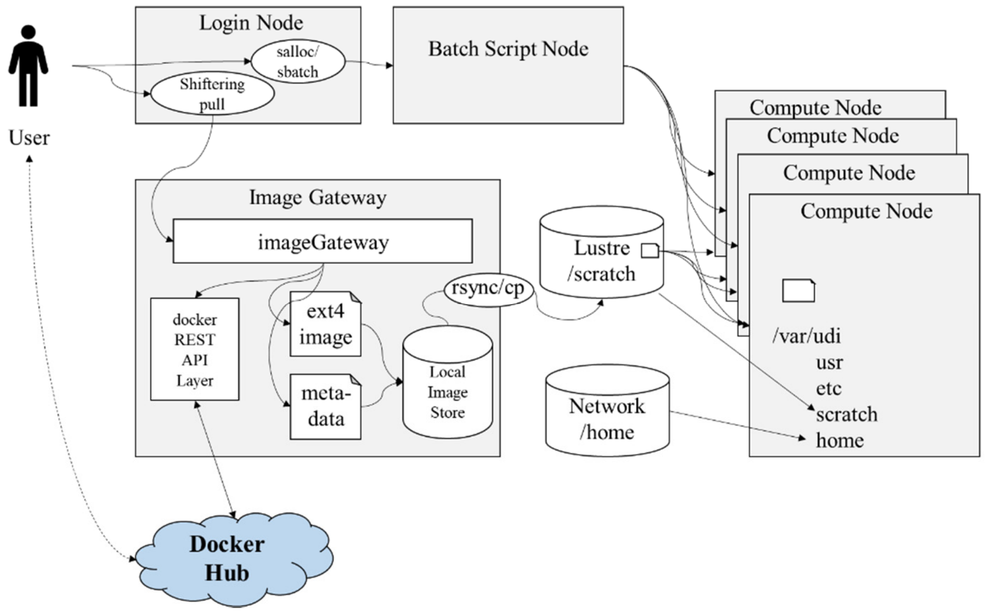


Figure 2. NERSC Shifter workflow.

The Swiss National Supercomputing Center (CSCS) has developed a CSCS Shifter [19] based on a NERSC Shifter to work with GPU resources at the Piz Daint supercomputer. A CSCS Shifter can be used by loading the module shifter-ng. To use Shifter commands, users must include the GPU-enabled software stack by loading daint-gpu [19]. As shown in Figure 3, the workload manager can distribute job requests from users, but this can also be a connected runtime service. This architecture not only manages images but also runs containers directly with these converted images. Although this solution implements automation of the job workflow by Shifter Runtime, it still has a dependency issue in that it has to run in its own supercomputer environment and a network performance issue between parallel processing running on multiple nodes.

2.2.2. Sarus

Sarus is a software package designed to run Linux containers in HPC environments and was developed by the CSCS in 2019. This software was developed specifically for the requirements of HPC systems and was released as an open source application [10]. The main reason for the development of Sarus is the increased demand for container-based HPC platforms that solve the following issues [10]:

- Suitable for an HPC environment: It includes diskless compute nodes and workload manager compatibility, is parallel file system friendly, and has no privilege escalation (multitenant).

- Vendor support: NVIDIA GPU, Cray MPI, and Remote Direct Memory Access (RDMA) (it should meet the standard open container initiative (OCI) hooks)
- User experience: The Docker-like Command Line Interface (CLI), Docker Hub integration, and writable container file system should be satisfied.
- Admin experience: It includes easy installation (e.g., single binary) and container customization (e.g., plugins).
- Maintenance effort: It can apply third-party technology and leverage community efforts.

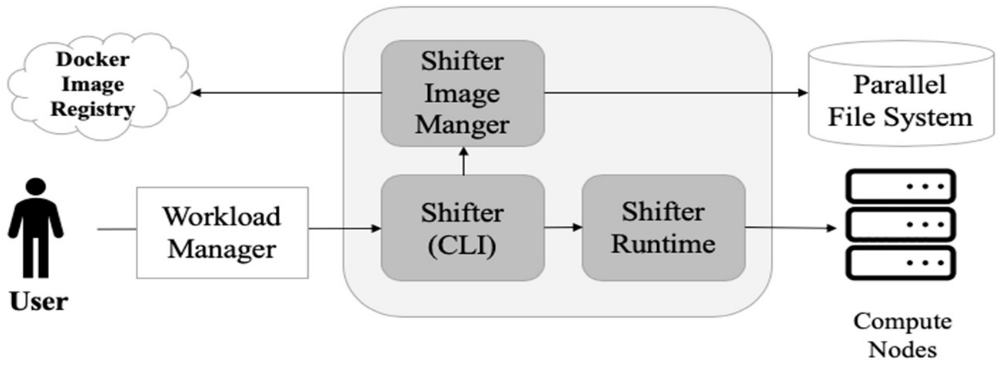


Figure 3. CSCS Shifter workflow.

The architecture of Sarus solves all the above issues; the Sarus workflow is shown in Figure 4. It includes an image manager and a container runtime manager. Users working through the CLI can send container image requests from Docker Image Registry, and a parallel file system can handle Docker image formats as well as TAR-compressed file formats. Users operating through the JavaScript Object Notation (JSON) format can send OCI bundle requests that can be handled with the root file system. It delivers requests from users to the OCI runtime, such as runc to execute the container process, which creates hooks such as the MPI job, NVIDIA GPU job, and other jobs. In this architecture, it is impressive to use the concept of hooks that can support various HPC workloads for job workflows. However, in the case of the Image Manager in Figure 3, it does not solve the workload of the image management server when numerous users send their requests at the same time. Furthermore, it is taxing for this Image Manager to download images from Docker Image Registry or move images compressed with TAR with only one manager.

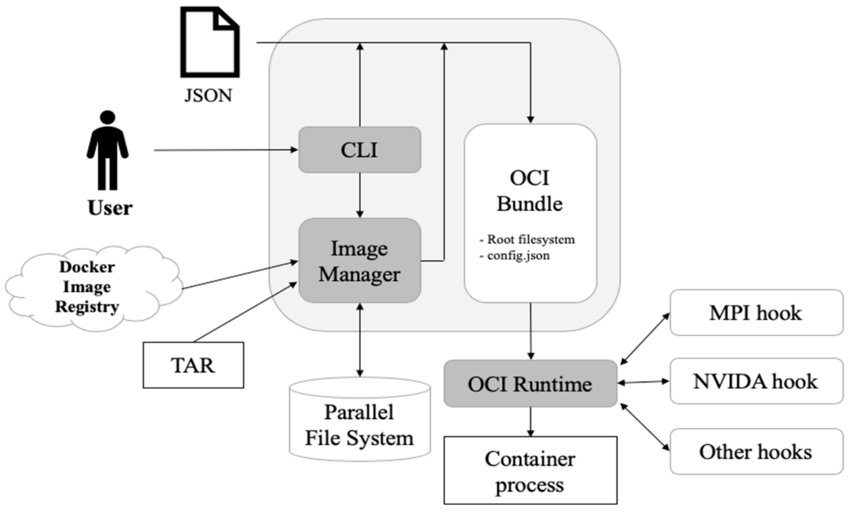


Figure 4. CSCS Sarus workflow.

2.2.3. EASEY

One application of Charliecloud is in developing a software program called EASEY, which is a containerized application for HPC systems [11]. As shown in Figure 5, the EASEY workflow for job submissions on HPC systems uses Charliecloud to connect user systems with HPC software stacks, such as batch schedulers and storage servers (file system), using EASEY middleware. We selected this study case because it is a good example of developing a user-defined software stack for the HPC environment, even though no automation is implemented for the job workflow.

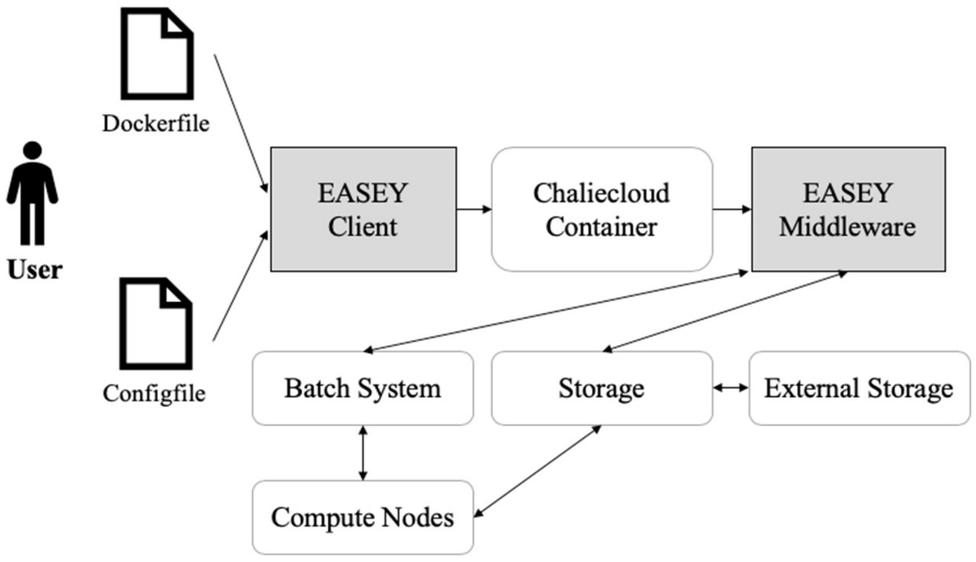


Figure 5. EASEY Charliecloud workflow.

2.2.4. JEDI

An example of using Singularity (also Charliecloud) is provided by jedi-stack, developed by the JEDI team in the Joint Center for Satellite Data Assimilation (JCSDA) [12]. As shown in Figure 6, with jedi-stack, jedi users can use Docker, Charliecloud, and Singularity to use cloud resources or HPC resources. This provides instant images for cloud computing and environment modules with selected HPC systems. For continuous integration testing, Travis-CI is used for leveraging Docker containers. This team uses packaged user environment concepts for defining the software container. The advantage of this architecture is that resource pools are divided into cloud and HPC environments, and each pool is provided by the different solutions Charliecloud (Docker based) and Singularity. As shown in this architecture, job workflow management is not mentioned or implemented.

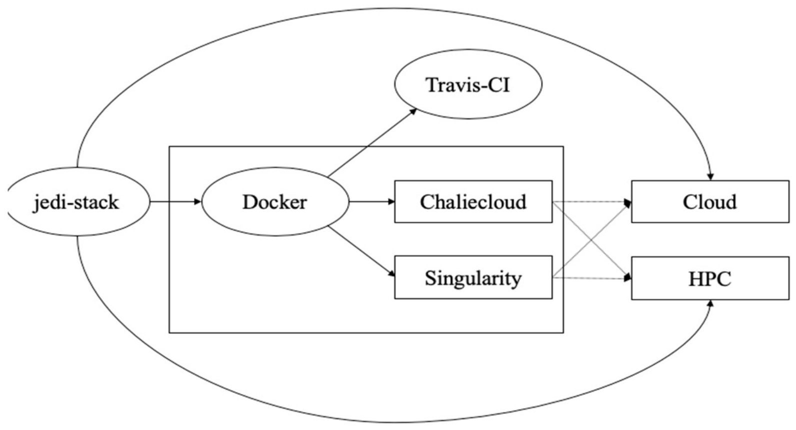


Figure 6. Jedi-stack workflow.

2.3. Requirements of the HPC Workflow in a Containerized Environment

From our review of related projects, we listed 10 requirements and compared them with those of our proposed platform in Table 1. These requirements were derived from our experiences of HPC users and system administrators. We define an HPC workflow as the flow of tasks that need to be executed to compute on HPC resources. Tasks in a containerized environment can be divided into image, template, and job (container and application) management.

Table 1. Comparison of requirement of projects vs. the proposed platform.

Requirements	Shifter	Sarus	EASEY	JEDI	Proposed Platform
A self-service interface	Image	Image, template, job	Image, template	Image	Image, template, job
Container solutions	Docker	Docker	Charliecloud	Docker, Charliecloud, Singularity	Docker, Singularity
A rapid expansion of new configuration environments: (1) Template-based image management (2) Image-based job management (3) Image format conversion	(1), (3)	(1), (2), (3)	(1)	None	(1), (2)
A pay-as-used model (on-demand billing)	None	None	None	None	Yes
Performance similar to that of bare-metal servers	None	Yes	Yes	None	Yes
Auto-provisioning that includes virtual instances and applications	Yes	Yes	None	None	Yes
Workload management	Image	Image, job (batch scheduler)	None	None	Image, job (batch scheduler)
Resource auto-scaling	None	None	None	None	Yes (image management)
Multitenancy capability	Yes	Yes	None	None	Yes
Portability of application	Yes	Yes	Yes	None	Yes

In Table 1, for a self-service interface requirement, the Sarus project designed image, template, and job management, which was implemented as command line interfaces. Our proposed platform was also designed to support image, template, and job management with integrated command line interfaces. Shifter and Sarus use Docker as their basic container solution. The EASEY project was designed based on a Charliecloud solution, which can be used in an HPC environment. The JEDI project supports various container solutions such as Docker, Charliecloud, and Singularity. We selected only Docker and Singularity for our base container solutions because we considered architectural ways to provide unprivileged containers with Docker instead of Charliecloud. For a rapid expansion of new configuration environments, we listed three functionalities: template-based image management, image-based job management, and image format conversion. Sarus supports all three features. We first attempted to implement two main features and included image format conversion between Docker and Singularity as our future work.

For a pay-as-used model, we implemented an on-demand billing system by collecting metering data with an existing authentication and authorization policy not supported in other projects. Sarus and EASEY use the host network with containers through multinode computing. We also evaluated container networking performance with tuning in our previous research. Shifter and Sarus support auto-provisioning that includes containers and inside running applications. For workload management, Sarus supports both image and job management. We referred to the architecture of this project. For resource auto-scaling, no projects support it. We implemented it for image management to distribute a request load. For multitenancy capability, Shifter and Sarus were designed to use a shared resource pool but provide various separated services to users. All projects support portability of application with container solutions.

3. Platform Design

To meet the requirements that were mentioned in the previous section, we designed a container-based HPC cloud platform based on system analysis. The system architecture and workflow designs were proposed with a consideration of the requirements of current users and administrators. The workflow of the image management system, job management system, and metering data management were explained in detail.

3.1. Architecture

This platform was designed in accordance with the three service roles of the cloud architecture (Figure 7); these are service creator, service provider, and service consumer roles that must be distinguished to enable self-service. In this figure, items in blue boxes were implemented by exiting from the open source software, those in green boxes were developed by integrating the necessary parts, and those in yellow boxes were newly developed. The service creator can manage a template consisting of Template Create and Template Evaluate processes. The previously verified templates can be searched using Template List and Template Search and can also be deleted using Template Delete. Template List, Template Search, and Template Delete were developed as CLI tools and provided as a service. All verified templates are automatic installation and configuration scripts for versions such as Operating System (OS), library, compiler, and application. All container images can be built based on verified templates. All jobs (including container and application) can be executed based on these built images.

HPCaaS is provided by our container-based HPC cloud service provider to service consumers through the container-based HPC cloud management platform, which consists primarily of job integration management and image management processes. Our platform provides services based on two container platforms, the hardware-based management of which is accomplished with Container Platform Management. Image management is based on a distributed system and is a base for the implementation of workload-distributing, task-parallelizing, auto-scaling, and image-provisioning functions of Image Manager in detail. We designed container and application provisioning by developing integration packages according to the different types of jobs because different container solutions need different container workload managers. The on-demand billing function was implemented using measured metering data. We designed real-time resource monitoring for CPU use and memory usage, and a function for providing various types of Job History Data records based on collected metering data. In addition, interfaces for connecting with other service roles were also implemented. Service Development Interface sends the template created by the service creator to the service provider. The image service and job service created on this template are delivered in the form of HPC Image Service and HPC Job Service through Service Delivery Interface.

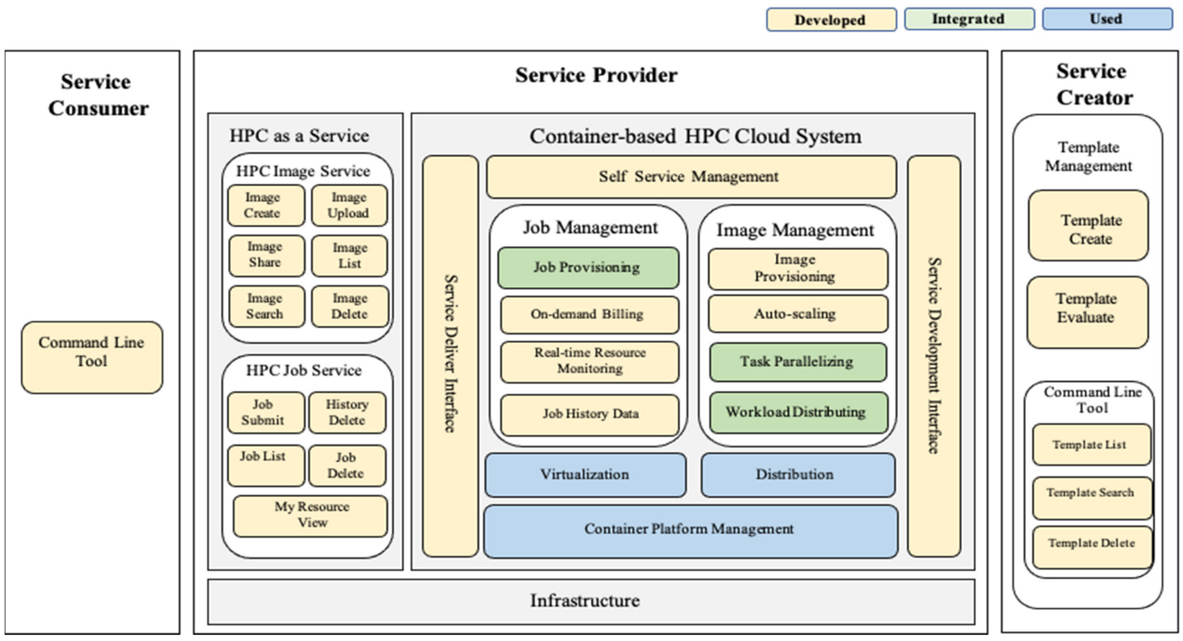


Figure 7. System architecture.

The workflow diagram presents a detailed design of the container-based HPC cloud platform, which includes image management (yellow boxes) and job integration management (blue boxes), as demonstrated in Figure 8. We proposed a distributed system for image management to reduce the workload resulting from requests. When HPC users request the desired container image, Image Manager automatically generates and provides the image based on the existing templates created by administrators. For example, when a user tries to execute an MPI job, a container image including MPI should be checked first. If the requested image exists, submit the MPI job with this image. If not, the user can request an image build with an existing template. If there is no template, the user can request a template from the administrator. Each user can request the image but can also share it with other users. In addition, we designed an auto-scaling scheduler for Image Manager nodes regarded as workers. To ensure job integration management, we integrated a batch scheduler and container orchestration mechanism to deploy the container and application simultaneously. After creating a container and executing applications, all processes and containers were automatically deleted to release the resource allocation. Additionally, a data collector for data metering was designed. Finally, we described My Resource View, which was designed to show the resource generated by each user and is used for implementing multitenancy. To share the resource pool with different or isolated services by each user, we designed My Resource View to check through usage statistics for each user's resources.

3.2. Image Management

The workflow of image management is shown in Figure 9. The user can submit image requests on the login node; when user requests are received, Image Manager delivers them to Docker Daemon or Singularity Daemon to deploy the images and then reports the history to Image Metadata Storage on the image manager node. Docker Daemon uses templates written in Docker file format to create Docker images automatically and in accordance with the user's request. All the created Docker images are stored in Docker Image Temporary Storage for the steps that follow. Singularity Daemon also uses templates written in definition file format to create Singularity images automatically, based on the request. Similarly, all created Singularity images are stored in Singularity Image Temporary Storage for upcoming steps. When the user requests Docker Image Share, Image Manager uploads the requested image to the Private Docker Registry that has already been built as a local hub. This local hub is used as Docker Image Storage to ensure high-speed transmission and security. When a user requests Singularity Image Share, Image Manager uploads the requested image to the parallel file system that has been mounted on all nodes. Once the image is uploaded, a job request can be submitted using this image.

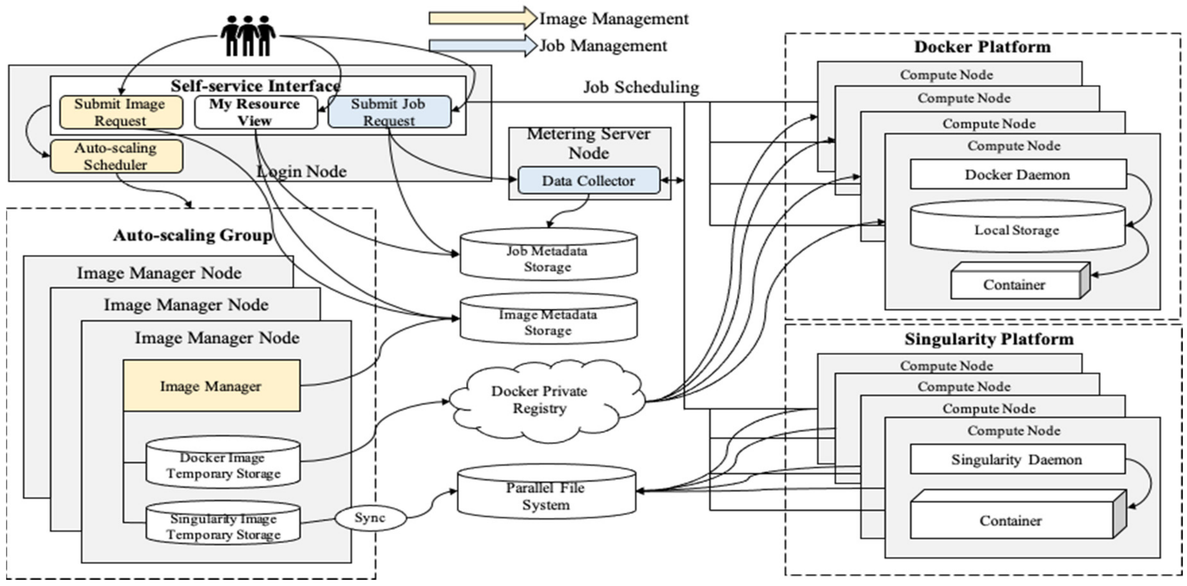


Figure 8. System workflow.

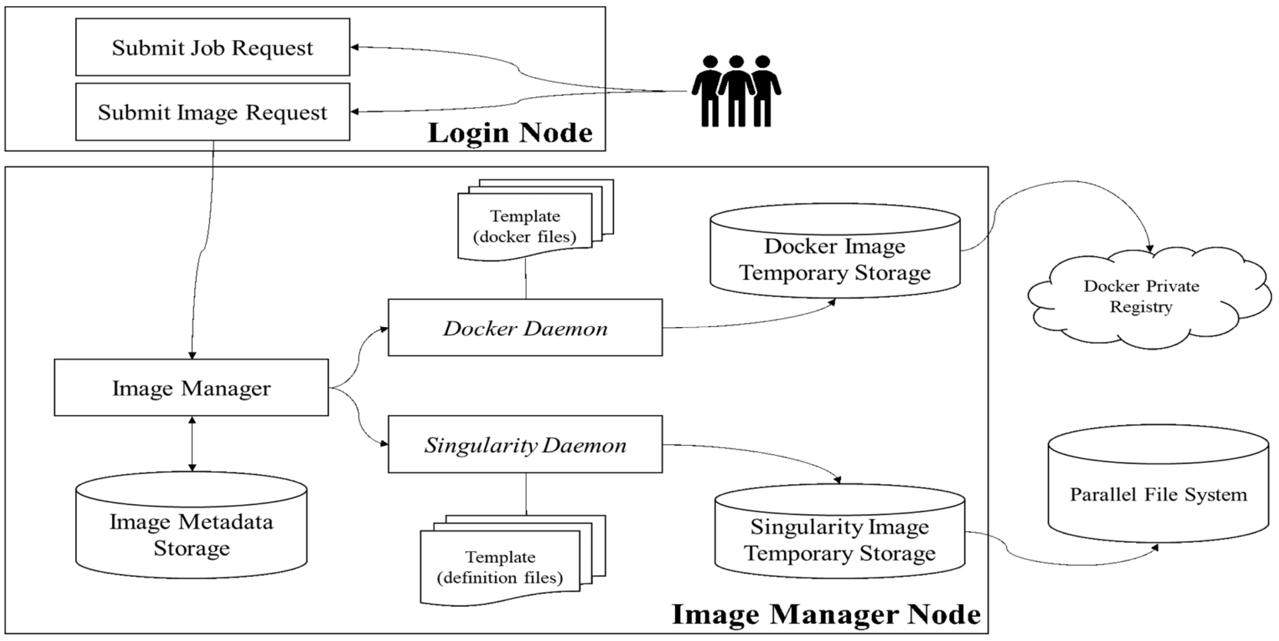


Figure 9. Image management workflow.

Image Manager constitutes the key point of our platform. We presented a distributed and parallel architecture for our platform to reduce the workload of Image Manager resulting from multiple users with access to Image Manager Server. As depicted in Figure 10, six features were designed as parallel workloads, thereby allowing users to send user requests simultaneously. On the login node, we deployed client parts from Client A to Client F; every client has its matching worker port designated Worker A to Worker F. We designed Docker Image Create, Docker Image Share, and Docker Image Delete Features for Docker workers; for Singularity workers, Singularity Image Create, Singularity Image Share, and Singularity Image Delete Features were designed. Between client and worker, we presented Task Queue for listing user requests by task unit. When User A creates task ① and User B creates task ②, according to the queue order, the Image Manager first receives task ① and then receives task ②. Likewise, tasks ③ and ④ requested by incoming users are queued in that order and are executed immediately after the preceding tasks are completed. Unlike other workload distributions for Image List and Image Search Features, these are designed separately to connect only to Image Metadata Storage.

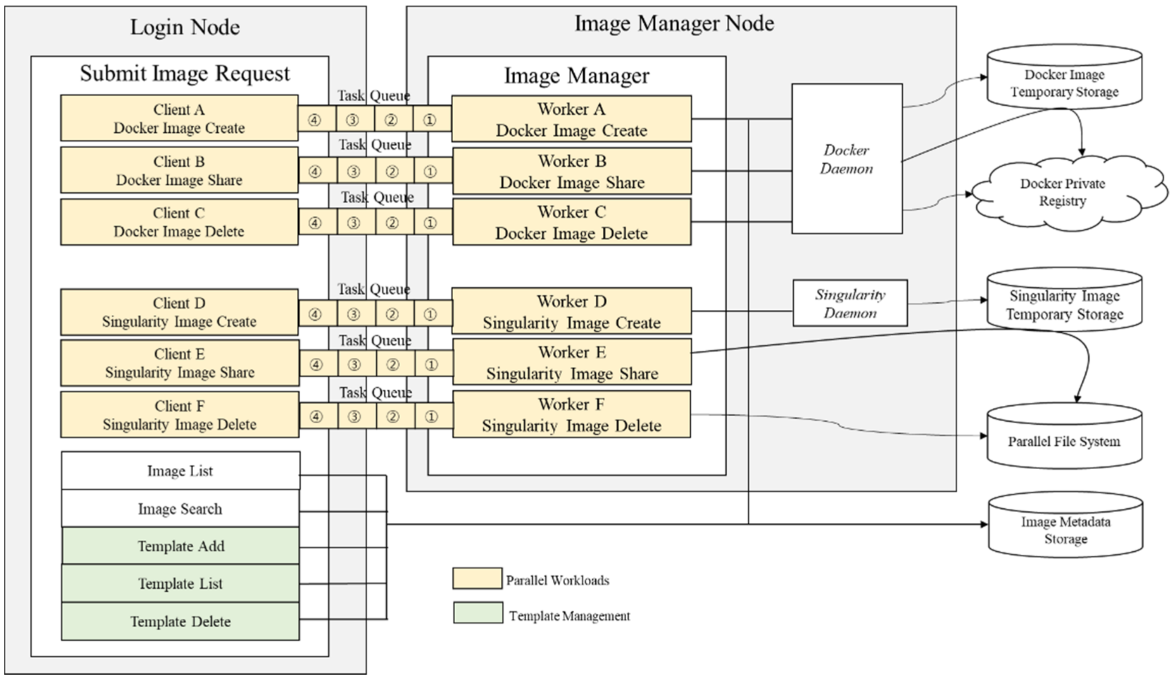

Login Node
Submit Image Request
Client A Docker Image Create
Client B Docker Image Share
Client C Docker Image Delete
Client D Singularity Image Create
Client E Singularity Image Share
Client F Singularity Image Delete
Image List
Image Search
Template Add
Template List
Template Delete
Task Queue
④ ③ ② ①
Image Manager Node
Image Manager
Worker A Docker Image Create
Worker B Docker Image Share
Worker C Docker Image Delete
Worker D Singularity Image Create
Worker E Singularity Image Share
Worker F Singularity Image Delete
Docker Daemon
Singularity Daemon
Docker Image Temporary Storage
Docker Private Registry
Singularity Image Temporary Storage
Parallel File System
Image Metadata Storage
Parallel Workloads
Template Management

Figure 10. Image Manager workflow.

We designed an auto-scaling scheduler composed of scale-up and scale-down rules for the scheduler. As shown in Figure 11, users can submit image requests to Auto-scaling Group using Login Node, which consists of a master worker node and several slave worker nodes. Redis-server runs on the Master Worker node and synchronizes the slave workers. Each worker will send the created image automatically to the public mounted file system or Docker Private Registry, depending on the type of platform.

To calculate the waiting time of the latency, the execution time of the image was added. After comparison of the execution time and the waiting time, it is determined whether to scale up or scale down the slave workers. Figure 12 presents the flowchart of the auto-scaling scheduler. We designed auto-scaling algorithms for each defined task queue that cannot be executed in parallel. When a user requests a task queue, our system performs the following determination steps starting with Create Image Start. Next, the system checks whether there are current active tasks in the auto-scaling worker group using Check Task Active. If tasks in the worker group are not currently active, any worker is selected to activate the task using Add Consumer, and the task is sent using Send Task to the selected worker using the routing key. When the task sent from the worker node is finished, Cancel Consumer is used to deactivate the task, and if it is not the master worker node, the worker node registered using Remove Worker is released. The task queue process for image generation is a scale-down and waits for the next user's request.

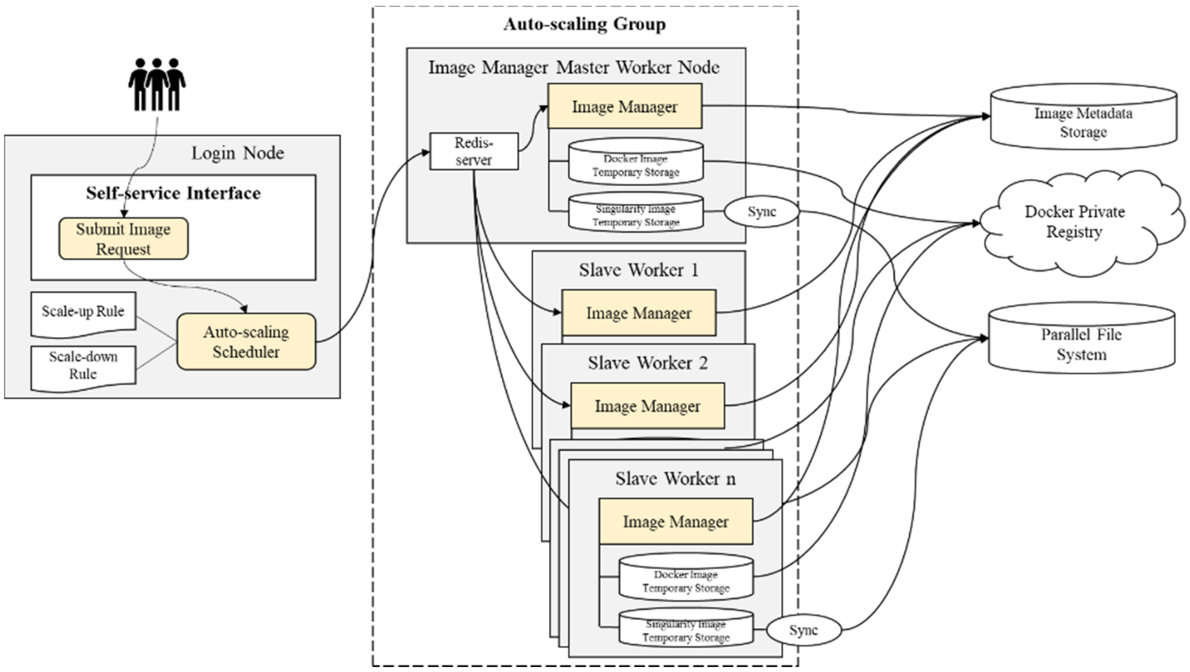


Figure 11. Auto-scaling scheduler workflow.

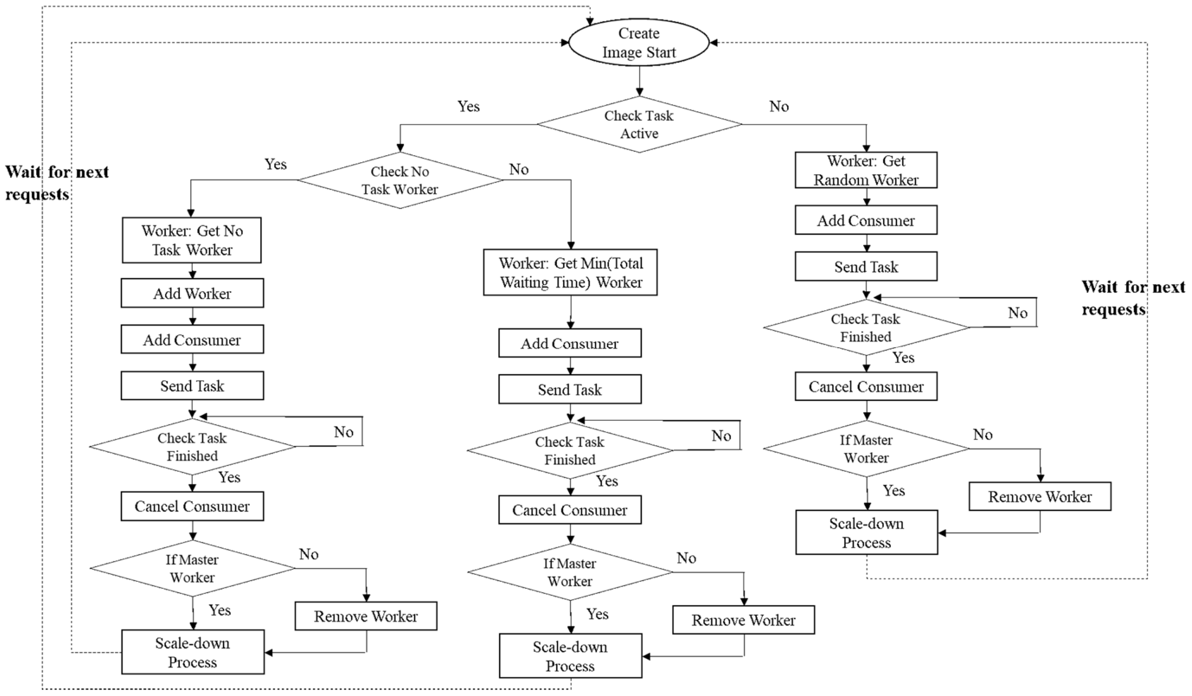


Figure 12. Flowchart of the auto-scaling scheduler.

However, if there are currently active tasks in the auto-scaling worker group, an arbitrary worker among the active workers is first selected using the following steps: add a worker node, activate the task queue using Add Consumer, and send the task by specifying the routing key. When the work is finished, and the worker node is released, it returns to the scale-down state and waits for another request. If there are currently running tasks in all workers, the worker node with the smallest sum of the waiting time is selected and sent. The variables and descriptions to get Equations (1) and (2) for the Get Min (total waiting time) worker are summarized in Table 2 below.

Table 2. Variables and descriptions of equations.

Variables	Description
N	The number of current workers with active tasks
n	The number of current active tasks
Φ CT	Current time
Φ ST	Start time of the current active task
Φ ET	Execution time for the image to run
Φ WT	Total waiting time for the active tasks
Φ W	Worker that has the minimum value of the total waiting time

Equation (1) explains Φ **WT** that calculates the total waiting time for active tasks. After getting a list of currently running tasks, how long each task has been executed up until the present time is calculated using the formula Φ **CT**−Φ **ST**. Here, Φ **ST** represents the start time of the current active task, Φ **CT** the current time of the system, and n the number of current active tasks. Φ **WT** denotes the sum of latencies of active tasks for each worker node that has active tasks. Finally, we can get the worker Φ **W** that has a minimum value of the total waiting time of the workers using Equation (2).

$$\Phi\ \mathrm{WT} = \sum_{n=1}^{\infty} (\Phi\ \mathrm{ET}_n - (\Phi\ \mathrm{CT}\ -\ \mathrm{ST}_n)) \tag{1}$$

$$\Phi \mathrm{W} = min(\sum_{N=1}^{\infty} (\Phi\ \mathrm{WT}_N)\ ) \tag{2}$$

However, when these equations are applied to an actual cluster environment, the following limitations exist. We implemented one worker per node; therefore N also stands for the number of nodes of Image Manager. The higher the value of N, the better, but it is limited by the number of network switch ports connecting nodes in a cluster configuration. Considering the availability of our HPC resources, we tested N up to 3. We also implemented one task per process; therefore, n also stands for the number of processes of the host. The maximum value of n is the number of cores per one node. However, considering each task is not executed completely independently but shares the resource of the host, we actually created 4 tasks and tested them. The optimization of the resource use about n is one more area of further research.

3.3. Job Management

We presented a design of the job integration management system. After creating and uploading the image, the user can use Submit Job Request to schedule jobs, allocate computer resources, create containers, and execute HPC applications. The platform then automatically deletes containers to release allocated resources. Depending on container platform types, such as Docker and Singularity, our system designs different processes for submitting jobs. The key aspect of job integration management is the integration work done with a traditional batch scheduler so that traditional HPC users also can use our system. In addition, our system is designed to automate container, application, and resource releases through Submit Job Request. Our system consists of three main features, i.e., Job Submit, Job List, and Job Delete, for which respective flowcharts are presented in the following sections.

Different containers require different schedulers; the Singularity platform can directly use traditional HPC batch schedulers for submitting jobs, while the Docker platform requires integration with the default container scheduler. Job integration management in our system was designed to support both platforms. As shown in Figure 13, the user can submit jobs through Job Submit, but since the job submission process varies based on the platform, we designed it as both Docker Job Summit and Singularity Job Submit. Job List, Job Delete, and History Delete were also designed.

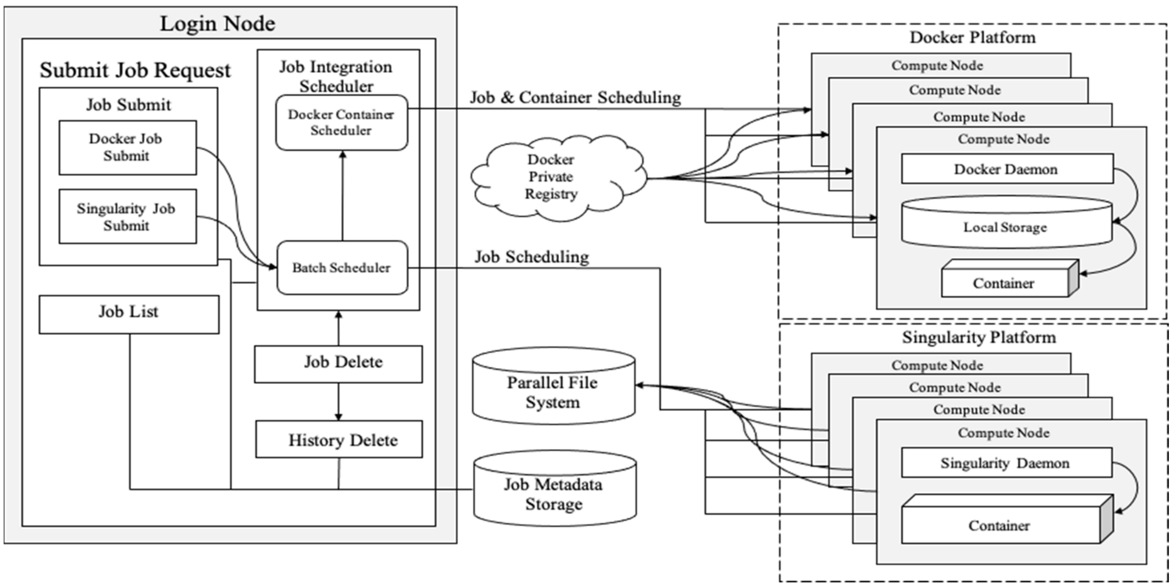


Figure 13. Job management workflow.

Metering data management was implemented by Data Collector, which consists of Resource Measured Data Collector, Real-time Data Collector, and Job History Data Collector, as specified in Figure 14. Resource Measured Data Collector collects resource measured data provided by the batch scheduler. Real-time Data Collector collects current CPU use and memory use for containers running in the container platform group by sending requests using the sshpass command every 10 s while the container is running. Job History Data Collector organizes measured data and real-time data designated by JobID as a history dataset that can be reconstructed into history data for a certain period. The sshpass command for sending requests every 10 s will create an overhead on both the metering node and the compute nodes. As a way to improve performance, there is a solution of collecting logs using another communication protocol or installing a log collection agent on the compute node side that can transmit to the metering node. It has not yet been applied as a future plan.

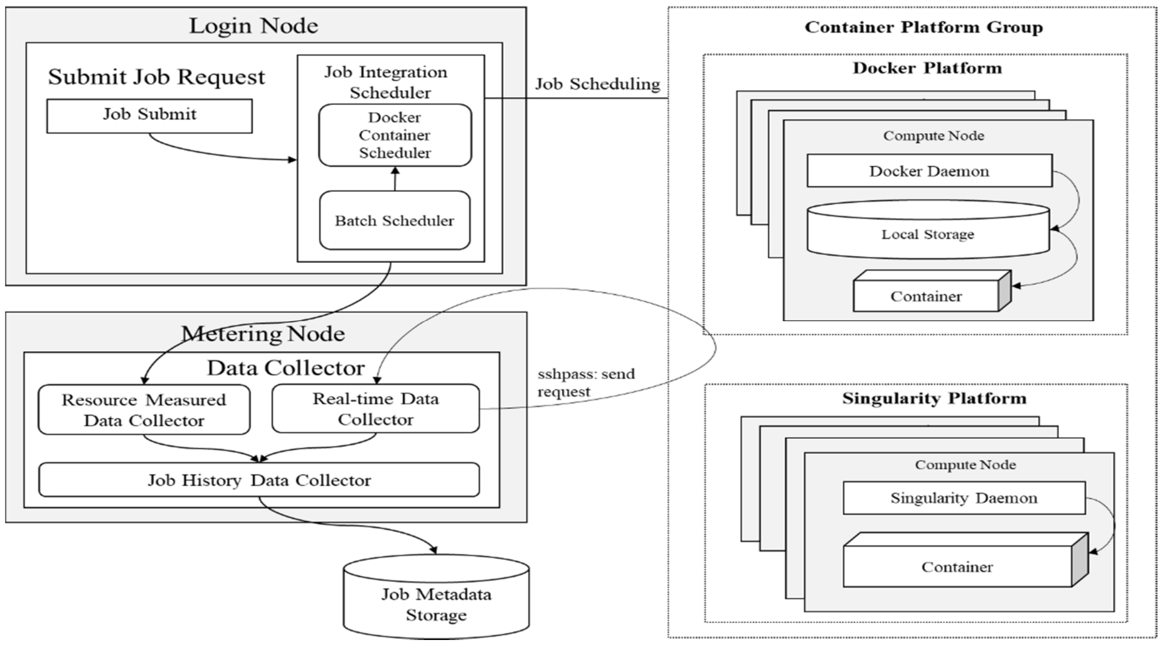


Figure 14. Metering management workflow.

4. Platform Implementation

Our research goal was to develop a system that provides container-based HPCaaS in the cloud. To evaluate our system, we created a cluster environment and verified the serviceability of our container-based HPC cloud platform using supercomputing resources. We configured the cluster to check the availability of our platform. As depicted in Figure 15, our container-based HPC cloud platform was constructed based on the network configuration of the KISTI cluster. There are three types of network configurations: a public network connected by a 1G Ethernet switch (black line), a management network connected by a 10G Ethernet switch (black line), and a data network connected by an InfiniBand switch (red line). We constructed three Image Manager Server nodes as an auto-scaling group, and these are connected to Docker Private Registry and the parallel file system GPFS storage with the management network. As the figure shows, we configured two container types, Docker Container Platform (green), which is configured with the Calico overlay network for container network communication, and Singularity Container Platform (blue), which is configured with the host network for container network communication. Both data networks for container communication are connected to InfiniBand Switch, and the Job Submit Server is deployed with PBSPro and Kubernetes. For Docker Container Platform, the integration scheduler of PBSPro and Kubernetes will work to create jobs. In contrast, for Singularity Container Platform, only PBSPro will work to create jobs. We also built Image Metadata Server and Job Metadata Server for storing and managing image metadata and job metadata, respectively.

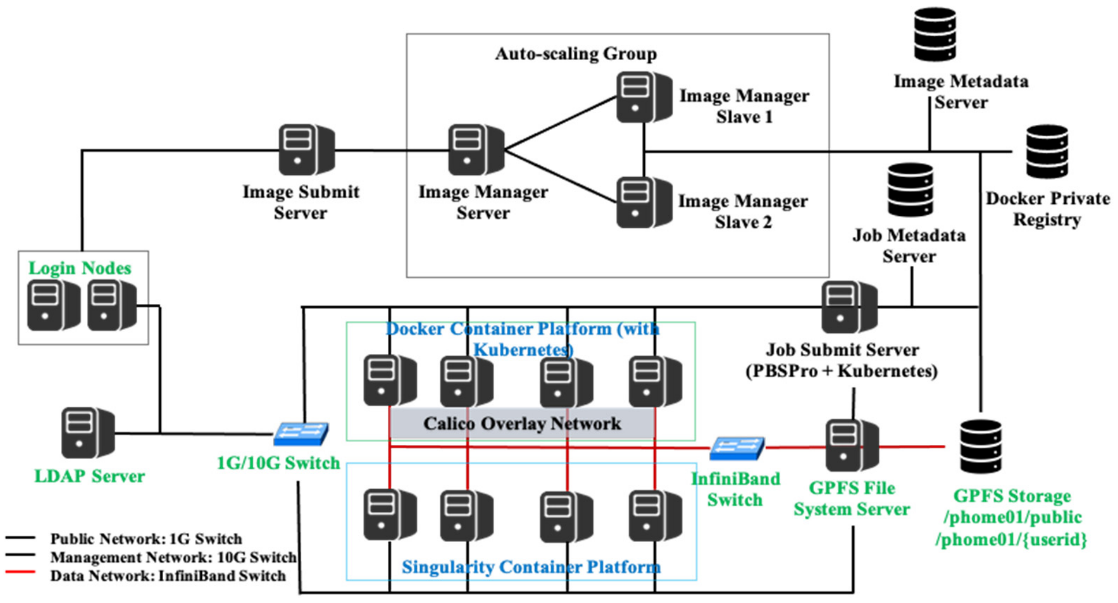


Figure 15. Cluster environment.

Table 3 shows the installed software information. We have software licenses for PBSPro and GPFS. Therefore, we used other open source software compatible with the software. The choice of open source software was based on easy and common use. We installed MariaDB v5.5.52 for Image Metadata Server as the relational database MySQL and MongoDB v2.6.12 for Job Metadata Server as the Not Only SQL (NoSQL) database. Job Metadata Server is more suitable for building NoSQL databases rather than relational databases for storing metering data on resource use. NoSQL is designed to improve latency and throughput performance by providing highly optimized key value storage for simple retrieval and additional operations. We selected MongoDB to implement the NoSQL database because it is designed based on the document data model and provides a manual for various programming languages, as well as a simple query syntax in Python. Job Submit Server and Image Submit Server that implemented the distributed system were installed using a combination of Python v2.7.5, Celery v4.1, and Redis-server v3.2.3. PBSPro v14.1.0 was installed as a batch scheduler, and Kubernetes v1.7.4 was installed as a container scheduler. The latest Docker v17.05-ce was installed with Calico v2.3 to configure the overlay network with Kubernetes for Docker containers. For Singularity Container Platform, we installed Singularity 2.2, which is the most stable version. PLSICLOUD is a user command line tool with which users submit image and job requests, and it was installed in Image Submit Server and Job Submit Server.

Table 3. Installed software information.

	Software	**Version**	**Installed Node (in Figure 14)**
Image Metadata Storage	MariaDB	5.5.52	Image Metadata Server
Job Metadata Storage	MongoDB	2.6.12	Job Metadata Server
Container Platform	Docker	17.06-ce	Docker Container Platform, Docker Private Registry, Image Manager Server
	Singularity	2.2	Singularity Container Platform, Image Manager Server
Batch Scheduler	PBSPro	14.1.0	Job Submit Server, Docker Container Platform, Singularity Container Platform
Docker Container Scheduler	Kubernetes	1.7.4	Job Submit Server, Docker Container Platform
Overlay Network Solution	Calico	2.3	Job Submit Server, Docker Container Platform
Distributed Task Queue	Celery	4.1	Image Submit Server, Image Manager Server, Job Submit Server
Programming Language	Python	2.7.5	Image Submit Server, Image Manager Server, Job Submit Server
Message Broker	Redis-server	3.2.3	Image Manger Server
User Command Line Tool (developed in Python)	PLSICLOUD	1.0	Image Submit Server, Job Submit Server

5. Evaluation

In this study, we evaluated a platform with essential attributes such as on-demand self-service, rapid elasticity and scalability, auto-provisioning, workload management, multitenancy, portability of applications, and performance, which can meet the listed requirements experienced by HPC users and administrators in Table 1.

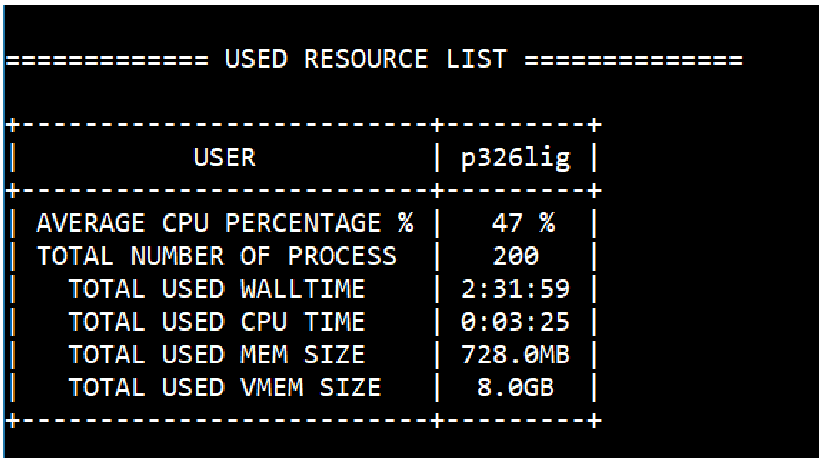
5.1. On-Demand Self-Service

On-demand self-service means that a consumer can unilaterally provision computing capabilities, such as computing time and storage, as needed automatically without requiring human interaction with each service provider [20]. HPC users can self-service on both current image resources and computational resources, which was automatically provided by the service provider, as shown in Figure 7. Users can also request their own images and submit jobs by allocating computational resources through the plsicloud_create_image and plsicloud_submit_job commands. We also presented on-demand billing management of this platform, which enables a pay-as-used model. By integrating the tracejob command of PBSPro, we implemented a resource usage calculation result for each user in Figure 16. As the figure shows, we provided average CPU use, total number of used processes, total used wall-time, CPU time, memory size, and virtual memory size for on-demand billing management evaluation. Based on this used resource information, our supercomputing center can apply a pricing policy for the pay-as-used model.

5.2. Rapid Elasticity and Scalability

Rapid elasticity and scalability mean that capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly up and inward commensurate with demand [20]. One important factor affecting rapid elasticity on our platform is the rapid movement of image files. When a user makes a job request for a resource with the required image, the system must quickly move the image created in Image

Temporary Storage to the compute nodes. For Singularity, there are no considerations when transferring an image, which can be also run as a container. The problem is to deploy the Docker image. Since Docker has a layered image structure and is managed by the Docker engine, it must be compressed into a file to move to the compute nodes. Considering the file compression time, file transfer time, and file decompression time, this is very inefficient. To solve this problem, we built Docker Private Registry that can store images connected with the management network. We calculated the degree of time deduction with and without Docker Private Registry for handling a 1.39 GB image. With Docker Private Registry, it took 122 s, 20 s faster than without Docker Private Registry.



```
============ USED RESOURCE LIST =============

+--------------------------+---------+
|           USER           | p326lig |
+--------------------------+---------+
| AVERAGE CPU PERCENTAGE % |   47 %  |
| TOTAL NUMBER OF PROCESS  |   200   |
|    TOTAL USED WALLTIME   | 2:31:59 |
|    TOTAL USED CPU TIME   | 0:03:25 |
|    TOTAL USED MEM SIZE   | 728.0MB |
|    TOTAL USED VMEM SIZE  |  8.0GB  |
+--------------------------+---------+
```

Figure 16. Resource usage.

For reducing the workload of creating images, we developed an auto-scaling scheduler with three workers implemented in a group for our platform. We applied this custom algorithm to compare the waiting time of each task with and without cases. With one worker, the second task must wait for the previous task until it is completed. The third task must wait for 2059 s, which is almost twice the first waiting time. For the fourth task in a queue, it will wait for 3083 s, requiring 6166 s in total as a waiting time. With this auto-scaling algorithm, three workers will work in parallel, so only 1080 s are needed for the fourth task; thus the total waiting time required is only 1200 s.

5.3. *Auto-Provisioning*

Automatic provisioning of resources is an integral part of our platform. In a containerized environment for the HPC cloud, the provisioning process of resources requires not only an image and a container but also their applications. More specifically, provisioning of a job in application units is needed rather than in container units. To solve this problem, we evaluated the auto-provisioning of image and job resources by making their life cycles.

Figure 17 shows the auto-provisioning life cycle of the image process. Once users request to build an image, configurations for image creation are verified and the state will be Creating. After image creation is complete, the state changes to Created and the image is automatically uploaded to Docker Private Registry or the shared file system. In this process, the image status is displayed as Uploading. Once it is uploaded, the state changes to Uploaded. If an error occurs during Creating, Uploading, and Deleting states, it is automatically displayed as the Down state. Images in the states of Created, Uploaded, and Down can be deleted, and these deleted images are automatically expunged in the repository.

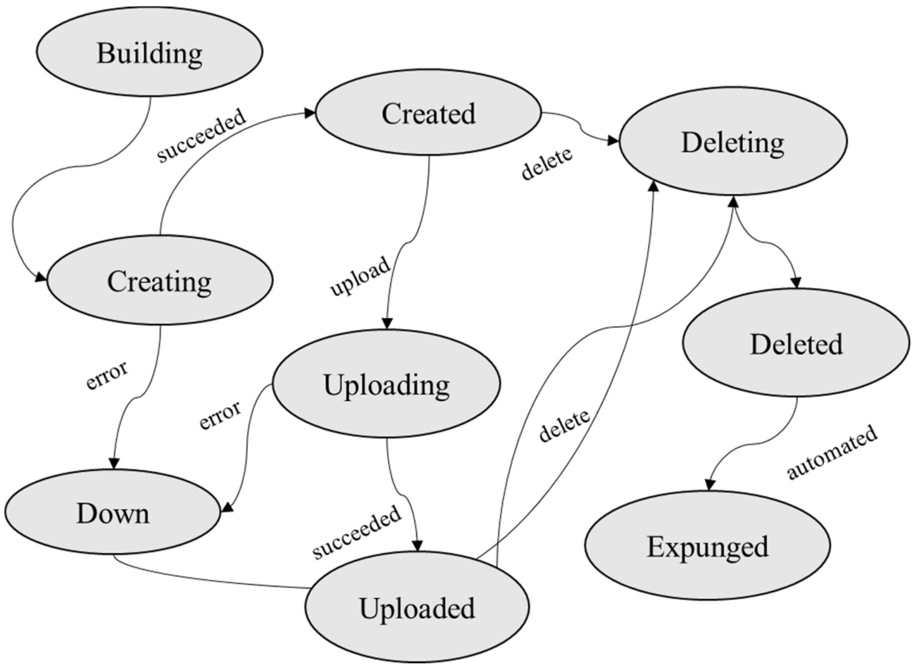


Figure 17. Auto-provisioning life cycle of the image process.

Figure 18 shows the auto-provisioning life cycle of the job process. Once users submit a job, containers are executed using a private or a shared image. In this process, the state changes from Building to Creating. After the creation is complete, the application is executed by changing to the Created state and then automatically going to the Running state. If the application execution is finished, the daemon of the job is automatically updated to the Finished state to complete the operation. Then, allocated resources are automatically released with expunging the container, including applications in the Expunged state. Jobs can be deleted in the states of Created, Running, Finished, and Down. If a forced request for resource release is received, the state shows Expunged, and then the allocated resource is released.

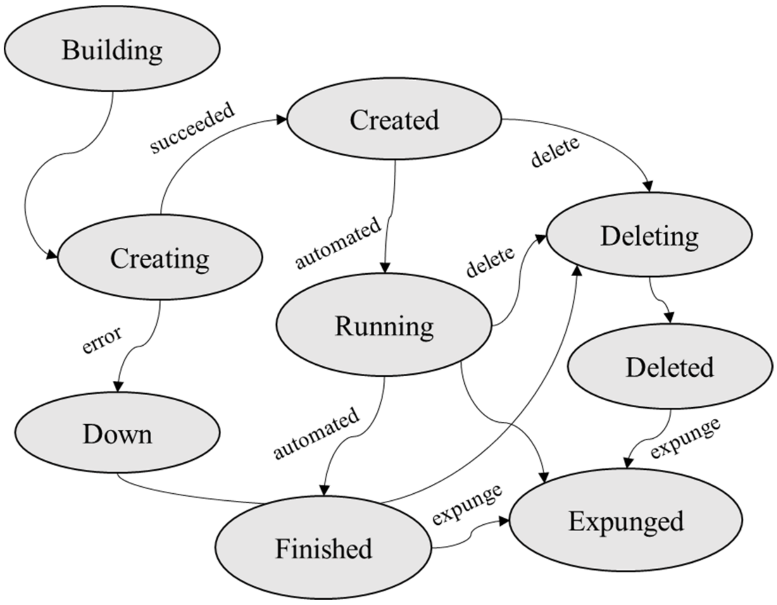
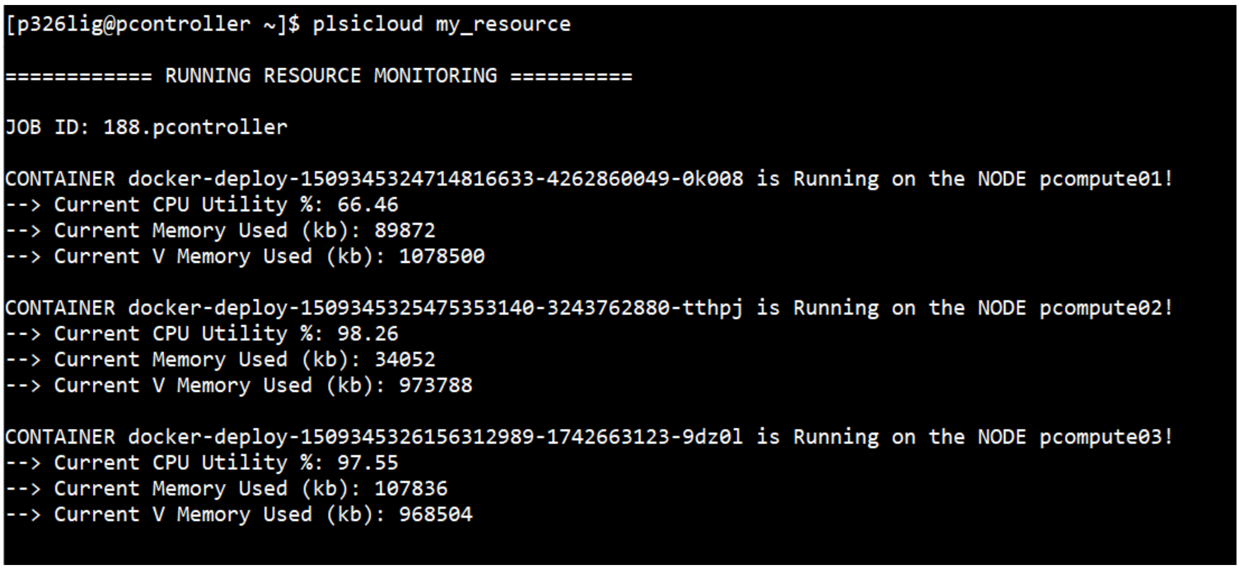


Figure 18. Auto-provisioning life cycle of the job process.

5.4. Workload Management

There are three workloads considered in our platform: image task, container execution, and application execution. Distributing and parallelizing workloads for image tasks are implemented by defining each client, task queue, and worker with Celery as a distributed framework and Redis as a message broker. Workloads for container and application execution are handled using an existing batch scheduler. We evaluated for managing these workloads by integrating some commands of the container scheduler (Kubernetes) and the batch scheduler (PBSPro). Figure 19 shows the resource monitoring, which includes information about containers and batch jobs using the plsicloud my_resouce command.



```
[p326lig@pcontroller ~]$ plsicloud my_resource

============ RUNNING RESOURCE MONITORING ==========

JOB ID: 188.pcontroller

CONTAINER docker-deploy-1509345324714816633-4262860049-0k008 is Running on the NODE pcompute01!
--> Current CPU Utility %: 66.46
--> Current Memory Used (kb): 89872
--> Current V Memory Used (kb): 1078500

CONTAINER docker-deploy-1509345325475353140-3243762880-tthpj is Running on the NODE pcompute02!
--> Current CPU Utility %: 98.26
--> Current Memory Used (kb): 34052
--> Current V Memory Used (kb): 973788

CONTAINER docker-deploy-1509345326156312989-1742663123-9dz0l is Running on the NODE pcompute03!
--> Current CPU Utility %: 97.55
--> Current Memory Used (kb): 107836
--> Current V Memory Used (kb): 968504
```

Figure 19. Resource monitoring information about containers and batch jobs.

5.5. Multitenancy

Multitenancy means that the provider's computing resources are pooled to serve multiple consumers assigned and reassigned to consumer demand [20]. In our platform, we provided the isolated environment to each user with shared resource pools. Figure 20 shows the concept of the multitenancy implemented in our platform. Image metadata and job metadata are stored in the two different types of databases—MySQL and NoSQL—according to the data characteristics of the resources. We can use PLSICLOUD to obtain the information sent to the service by each user. The development of the PLSICLOUD CLI tool allows evaluation of the cloud multitenancy model.

5.6. Portability of Applications

We evaluated the portability of applications by providing containerized applications. We verified containerization of frequently used container versions, OS versions, and compilers dependent on parallel libraries to meet the requirements of users, as shown in Table 4. We tested serviceability through the containerizing HPL task, a benchmark tool that tests HPC system performance and that is installed automatically from the OS to the application based on our templates [21]. Currently, our system supports Docker v17.06.1-ce and stable Singularity v2.2, along with CentOS 6.8 and 7.2; OpenMPI provides v1.10.6 and v2.0.2. Each of them contributes to a mathematical library GotoBLAS2 based on a parallel library.

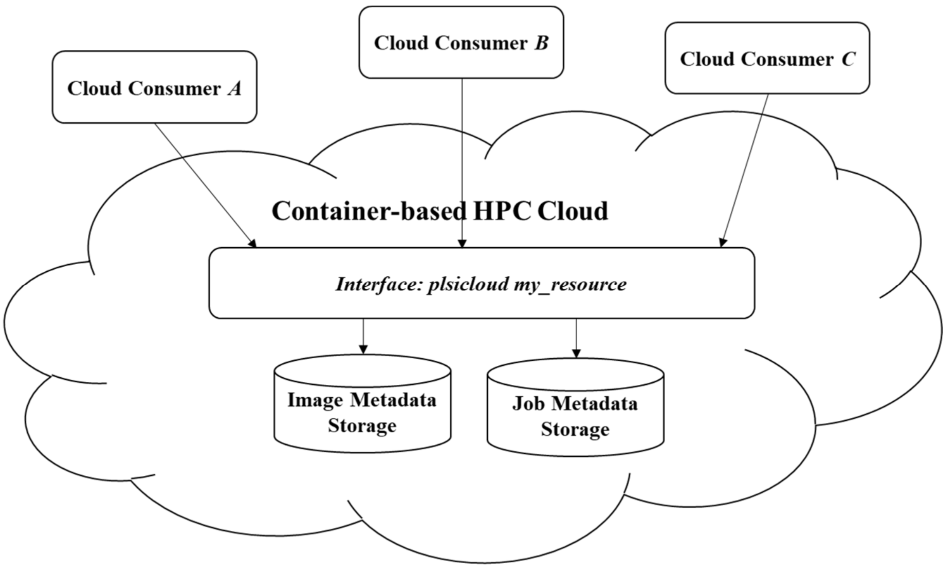


Figure 20. Multitenancy concept implemented in our platform.

Table 4. Template of containerized applications.

Container	OS	Library	Parallel Library		Application Library		Application	
Docker 17.06.1c-e / Singularity 2.2	CentOS 6.8	gcc gcc, Development Tools	OpenMPI	1.10.6	GotoBLAS	2-1.13	HPL	2.2
		gcc gcc, Development Tools		2.0.2	GotoBLAS	2-1.13	HPL	2.2
	CentOS 7.2	gcc gcc, Development Tools	OpenMPI	1.10.6	GotoBLAS	2-1.13	HPL	2.2
		gcc gcc, Development		2.0.2	GotoBLAS	2-1.13	HPL	2.2

5.7. Performance Evaluation with MPI Tests

We constructed two nodes for running point-to-point MPI parallel tasks to check different types of bandwidth and latency with the benchmark tool osu-micro-benchmarks v5.4 [22]. The latency test mainly measures the latency caused by sending and receiving messages by data size between two nodes using the ping-pong test. The test results of latency between two nodes with bare-metal, singularity, and docker-calico are almost the same and will be not mentioned in this paper [21]. The test results of bandwidth (Figure 21) include bandwidth, bi-directional bandwidth, and multiple-bandwidth tests. As shown in the figure, the peak bandwidth exists in a certain message size interval. This interval is almost the same in three cases of bare-metal, singularity, and docker-calico, except for the instability of docker-calico.

Results between the two nodes are insufficient to evaluate the performance of the container-based HPC platform in an HPC environment. Therefore, we constructed 8 nodes with 16, 32, 64, and 72 cores for measuring the latency test with various MPI blocking collective operations (barrier in Figure 22, gather in Figure A1, all-gather in Figure A2, reduce in Figure A3, all-reduce in Figure A4, reduce-scatter in Figure A5, scatter in Figure A6, all-to-all in Figure A7, and broadcast in Figure 23 using the same benchmark tool osu-micro-benchmarks.

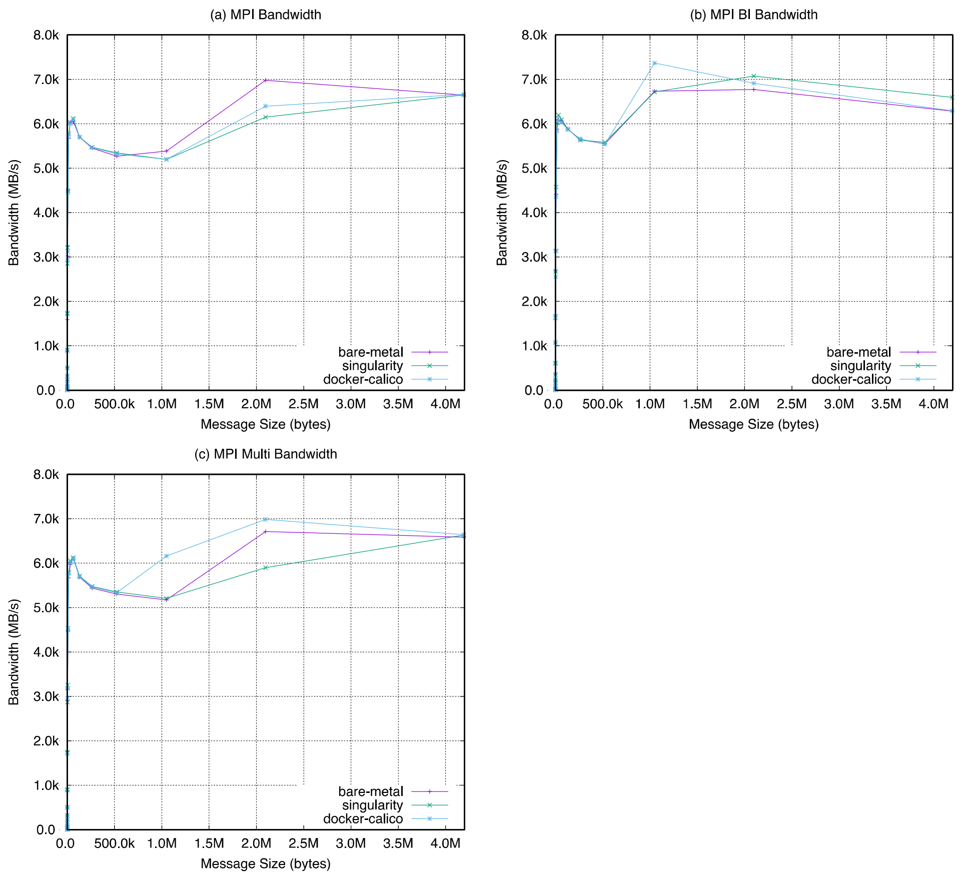


Figure 21. MPI bandwidth includes (**a**) MPI Bandwidth, (**b**) MPI BI Bandwidth, and (**c**) MPI Multi Bandwidth.

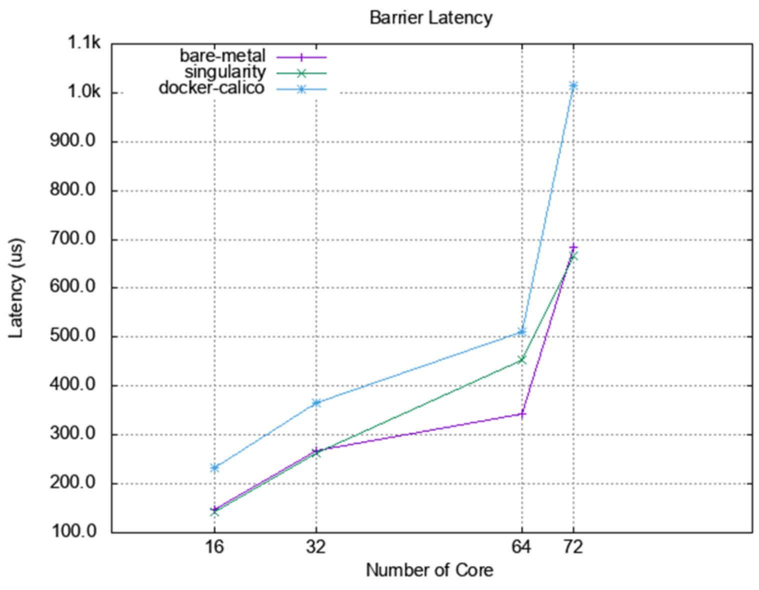


Figure 22. MPI barrier latency.

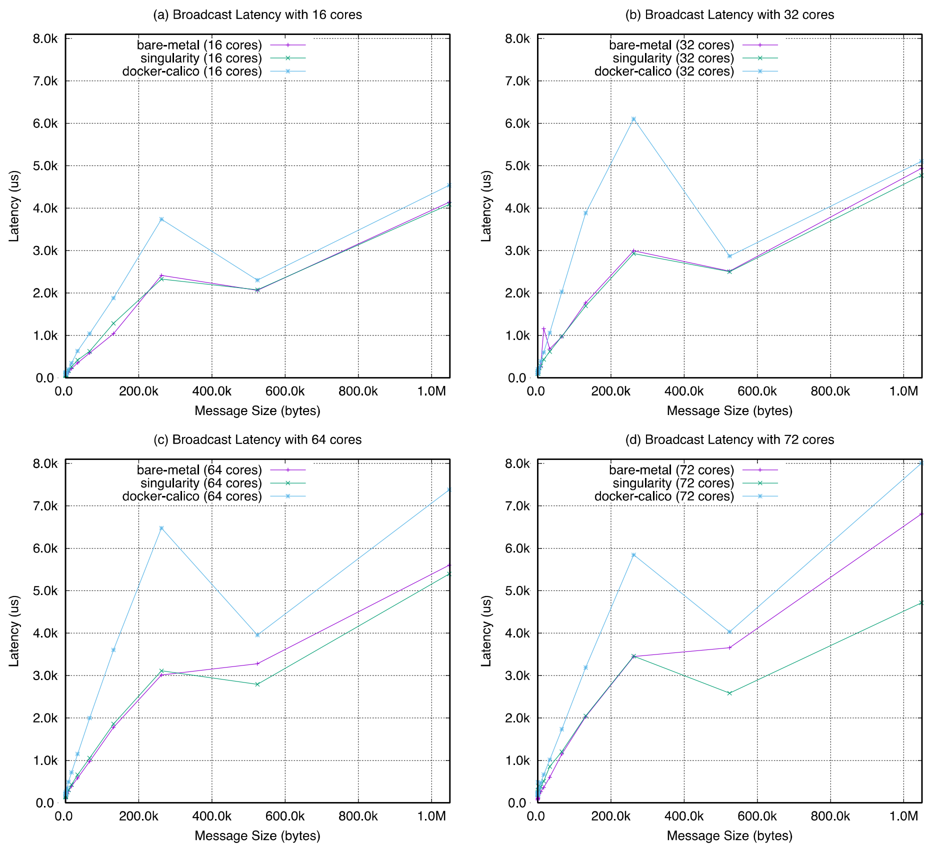


Figure 23. MPI broadcast latency with (**a**) 16 cores, (**b**) 32 cores, (**c**) 64 cores, and (**d**) 72 cores.

In Figure 22, the barrier latency of singularity shows almost the same performance with the bare-metal case except with 64 cores. The value of docker-calico shows a performance gap between singularity and bare-metal. For the remaining operations, as the number of cores increases in the same number of nodes, the performance results are almost the same with bare-metal, singularity, and docker-calico when looking at the overall graph change. However, in Figure 23, the results of docker-calico in a particular interval are worse than the other two cases in a specific message size range.

6. Conclusions and Future Work

The container-based approach to the HPC cloud is expected to ensure efficient management and use of supercomputing resources, which are areas that present challenges in the HPC field. This study demonstrates the value of technology convergence by attempting to provide users with a single cloud environment through integration with container-based technology and traditional HPC resource management technology. It provides container-based solutions to the problems of HPC users and administrators, and these can be of practical assistance in resolving issues such as complexity, compatibility, application expansion, pay-as-used billing management, cost, flexibility, scalability, workload management, and portability.

The deployment of a container-based HPC cloud platform is still a challenging task. Thus far, our proposed architecture has set and used measurement values for various resources by mainly considering KISTI's computer-intensive HPC jobs. In future work, we must consider HTC jobs, network-intensive jobs, and GPU-intensive jobs, especially for machine learning or deep learning applications, and add measurement values for new resources that satisfy the characteristics of these jobs. Another potential research task is to automate the process of creating and evaluating templates for HPC service providers as service creators. In our platform, there is not enough generalization in the degree of automation to conform with the application characteristics realized whenever a new template is added.

In the current job integration management part, additional development is required for the batch scheduler for general jobs and other interworking for Kubernetes with container jobs. In our platform, the user cannot access the container directly. Container creation, execution, application execution, and container deletion are all automated. However, it is possible to solve this issue by developing a linked package for the Application Programming Interface (API) of the existing Kubernetes and batch job schedulers; these will be connected to the machine on which the job is running, while submitting the job according to the user's requirements. The evaluation of our platform was conducted in a small cluster environment. If we apply our platform to a large cluster environment, future evaluations of availability, deployment efficiency, and execution efficiency would be needed. We hope that our proposed architecture will contribute to the widespread deployment and use of future container-based HPC cloud services.

Author Contributions: Conceptualization, G.L. and J.W.; methodology, G.L. and S.B.L.; software, G.L.; validation, J.W.; formal analysis, G.L.; resources, J.W.; writing-original draft preparation, G.L.; writing—review and editing, G.L. and S.B.L.; supervision, S.B.L.; project administration, J.W. All authors have read and agreed to the published version of the manuscript.

Funding: This research has been performed as a subproject of Project No. K-21-L02-C01-S01 (The National Supercomputing Infrastructure Construction and Service) supported by the KOREA INSTITUTE of SCIENCE and TECHNOLOGY INFORMATION (KISTI).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: No new data were created or analyzed in this study. Data sharing is not applicable to this article.

Acknowledgments: This study was performed as a subproject of the KISTI project entitled the National Supercomputing Infrastructure Construction and Service.

Conflicts of Interest: This manuscript has not been published or presented elsewhere in part or in entirety and is not under consideration by another journal. We have read and understood your journal's policies, and we believe that neither the manuscript nor the study violates any of these. There are no conflict of interest to declare.

Appendix A

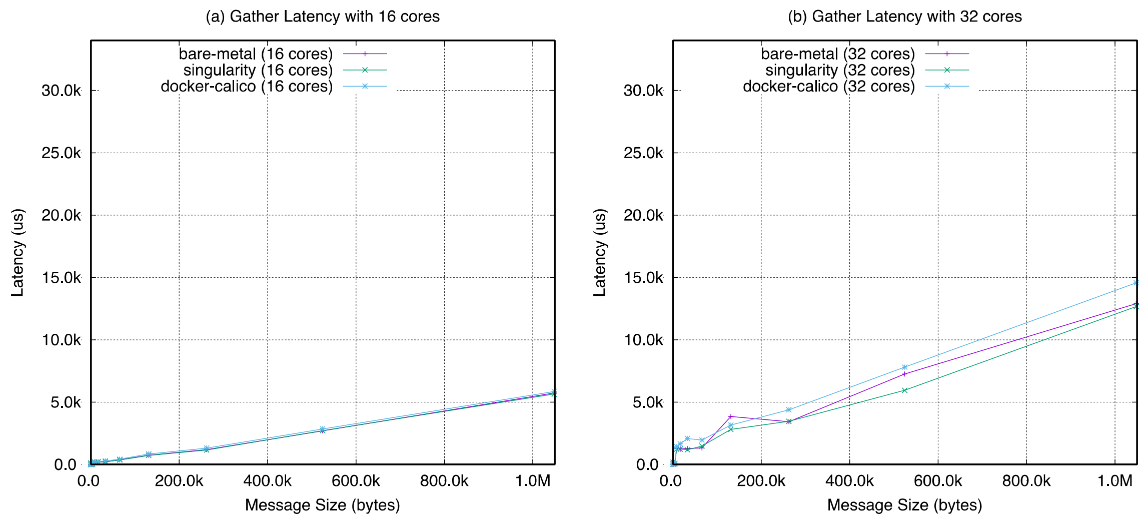


Figure A1. *Cont.*

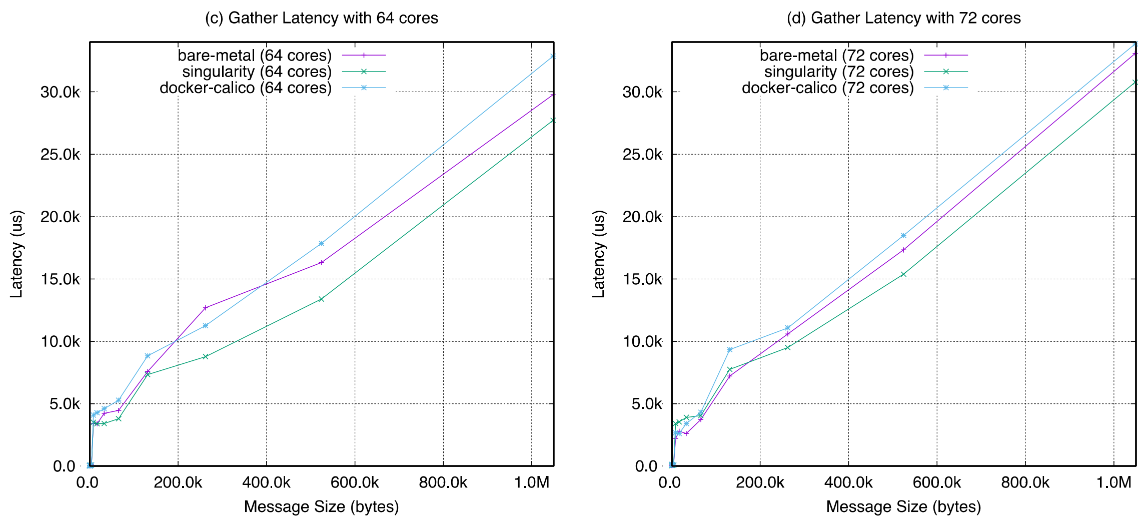


Figure A1. MPI gather latency with (**a**) 16 cores, (**b**) 32 cores, (**c**) 64 cores, and (**d**) 72 cores.

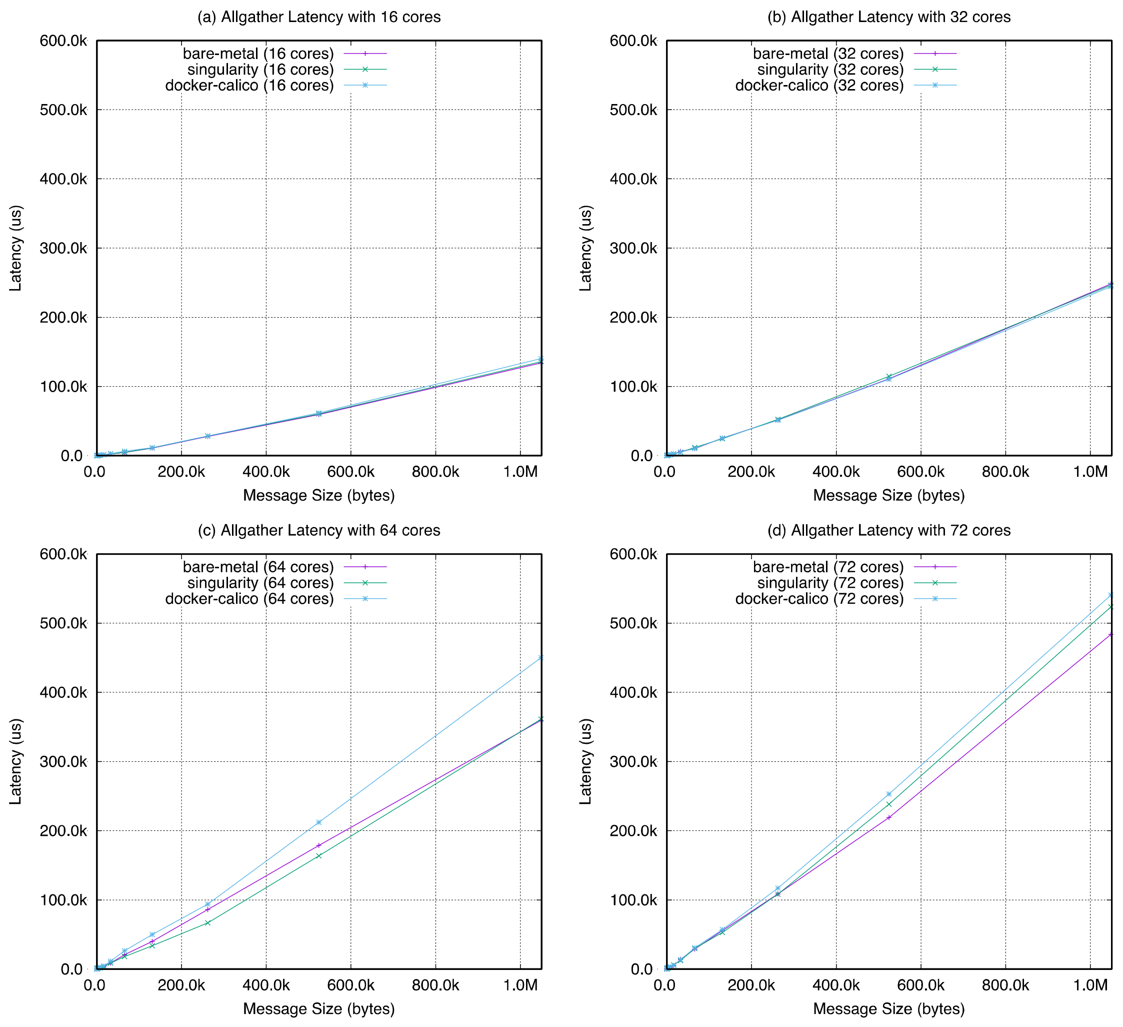


Figure A2. MPI all-gather latency with (**a**) 16 cores, (**b**) 32 cores, (**c**) 64 cores, and (**d**) 72 cores.

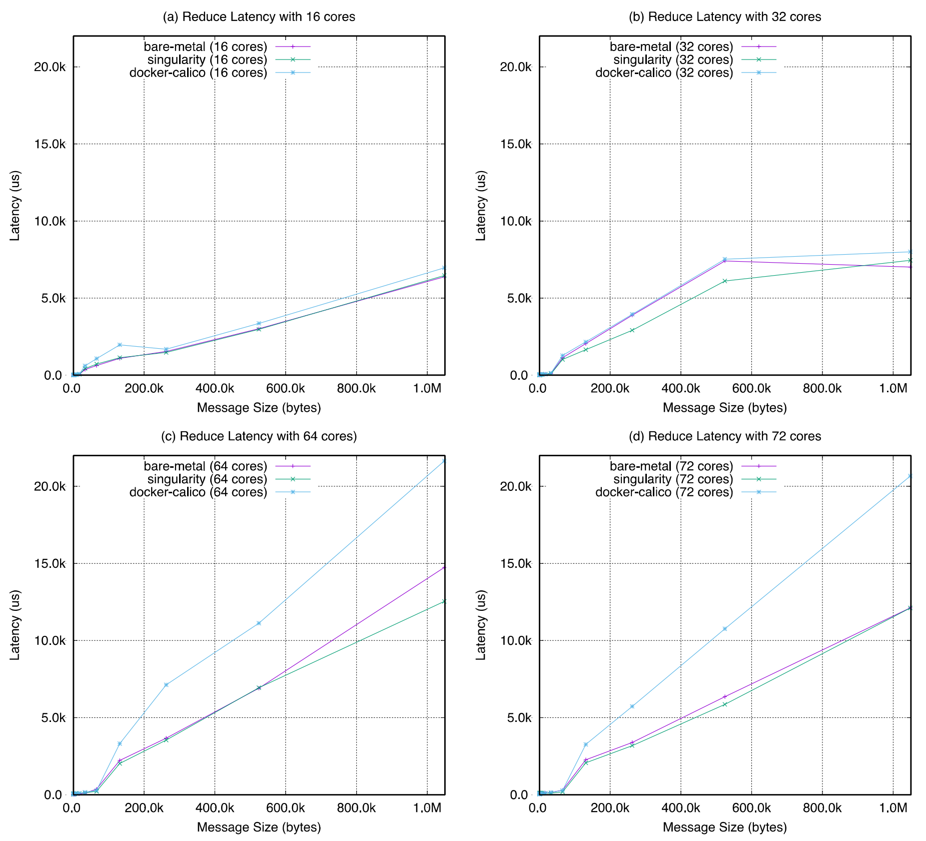


Figure A3. MPI reduce latency with (**a**) 16 cores, (**b**) 32 cores, (**c**) 64 cores, and (**d**) 72 cores.

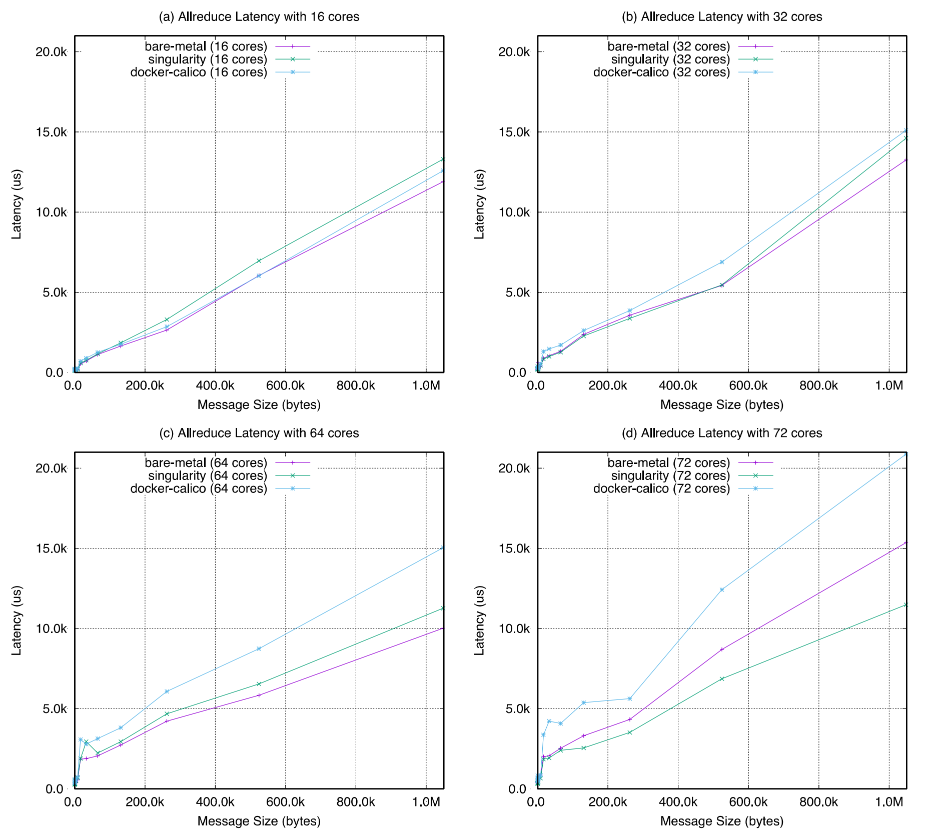


Figure A4. MPI all-reduce latency with (**a**) 16 cores, (**b**) 32 cores, (**c**) 64 cores, and (**d**) 72 cores.

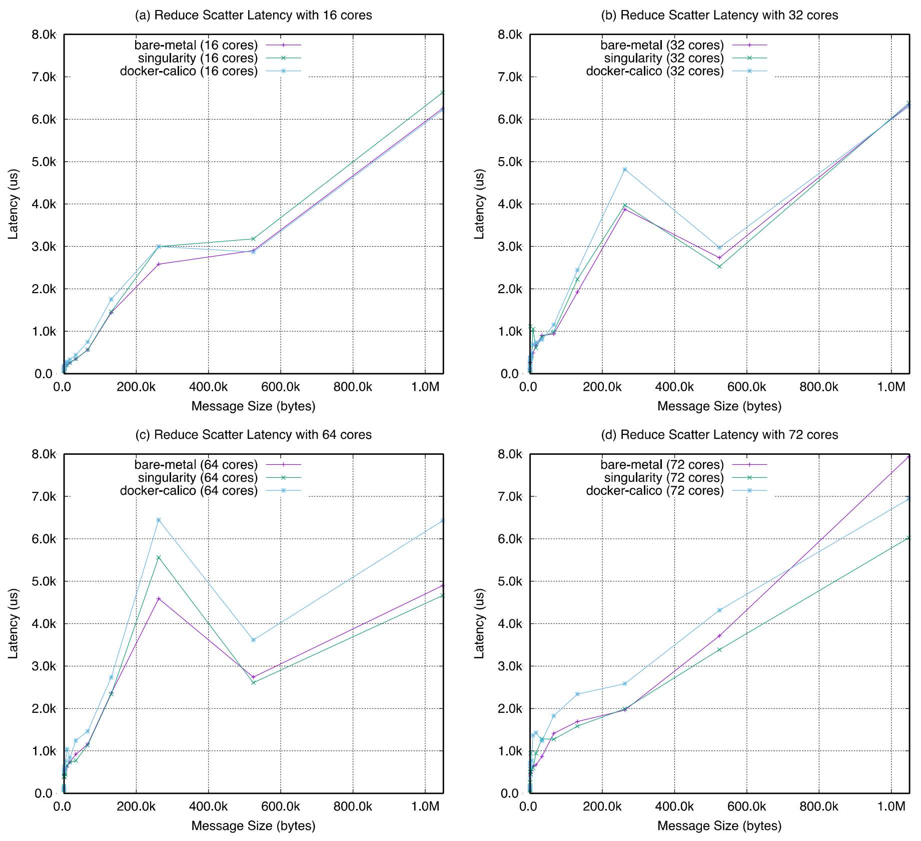


Figure A5. MPI reduce-scatter latency with (**a**) 16 cores, (**b**) 32 cores, (**c**) 64 cores, and (**d**) 72 cores.

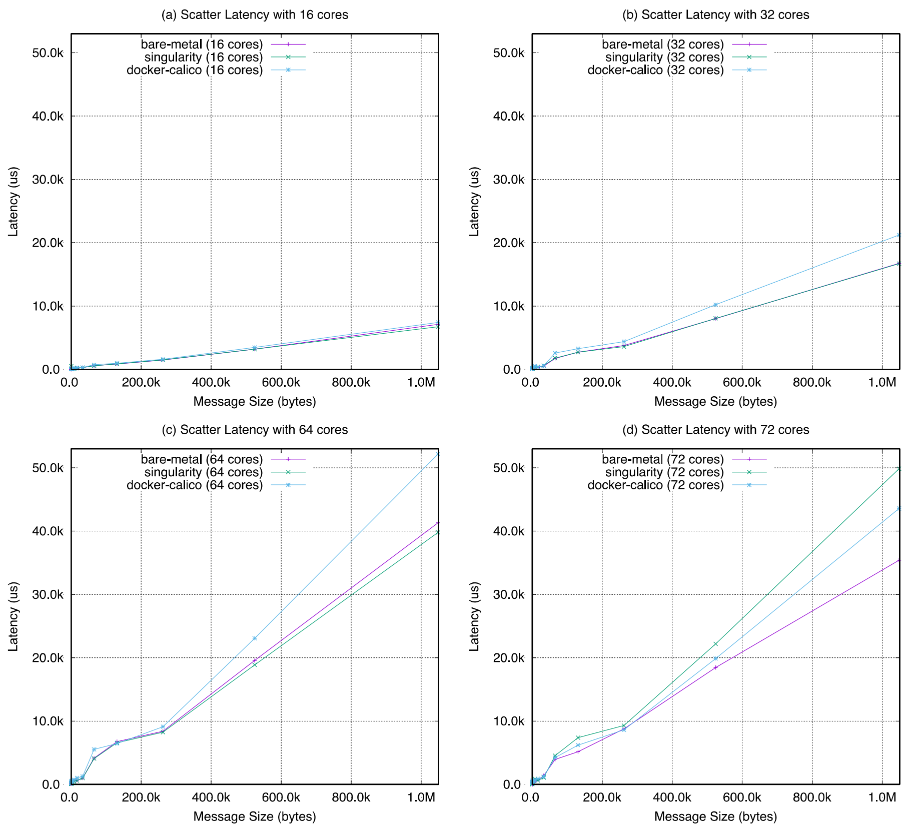


Figure A6. MPI scatter latency with (**a**) 16 cores, (**b**) 32 cores, (**c**) 64 cores, and (**d**) 72 cores.

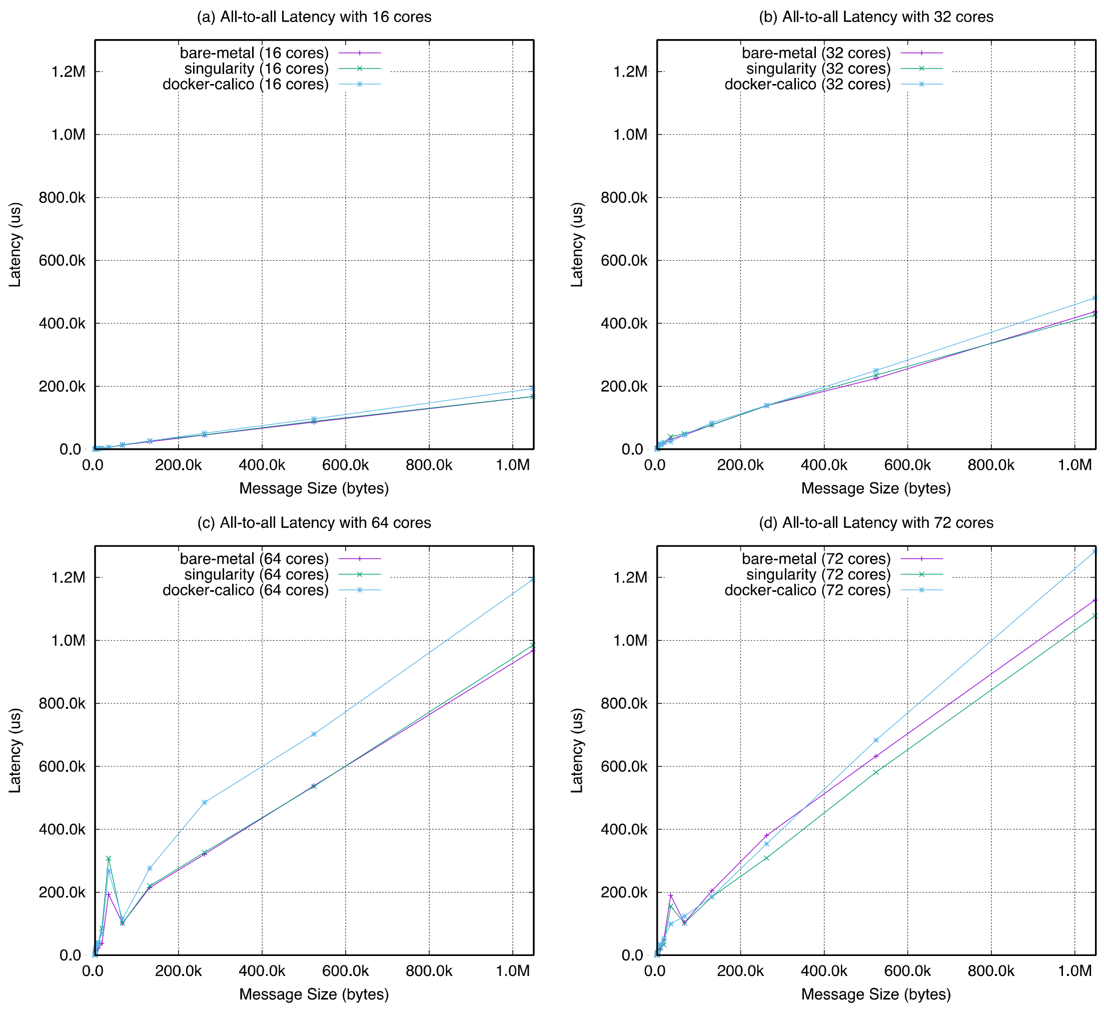


Figure A7. MPI all-to-all latency with (**a**) 16 cores, (**b**) 32 cores, (**c**) 64 cores, and (**d**) 72 cores.

References

1. Joseph, E.; Conway, S.; Sorensen, B. *High Performance Computing in the EU: Progress on the Implementation of the European HPC Strategy*; European Commission: Brussels, Belgium, 2014.
2. Fusi, M.; Mazzocchetti, F.; Farrés, A.; Kosmidis, L.; Canal, R.; Cazorla, F.J.; Abella, J. On the Use of Probabilistic Worst-Case Execution Time Estimation for Parallel Applications in High Performance Systems. *Mathematics* **2020**, *8*, 314. [CrossRef]
3. Smirnov, S.; Sukhoroslov, O.; Voloshinov, V. Using Resources of Supercomputing Centers with Everest Platform. In Proceedings of the 4th Russian Supercomputing Days, Moscow, Russia, 24–25 September 2018; pp. 687–698.
4. Fienen, M.N.; Hunt, R.J. High-Throughput Computing Versus High-Performance Computing for Groundwater Applications. *Ground Water* **2015**, *53*, 180–184. [CrossRef] [PubMed]
5. Raicu, I.; Foster, I.T.; Zhao, Y. Many-task computing for grids and supercomputers. In Proceedings of the 2008 Workshop on Many-Task Computing on Grids and Supercomputers, Austin, TX, USA, 17 November 2008; pp. 1–11.
6. Balakrishnan, S.R.; Veeramani, S.; Leong, J.A.; Murray, I.; Sidhu, A.S. High Performance Computing on the Cloud via HPC+Cloud software framework. In Proceedings of the 2016 Fifth International Conference on Eco-Friendly Computing and Communication Systems (ICECCS), Bhopal, India, 8–9 December 2016; pp. 48–52.
7. Jamalian, S.; Rajaei, H. ASETS: A SDN Empowered Task Scheduling System for HPCaaS on the Cloud. In Proceedings of the 2015 IEEE International Conference on Cloud Engineering, Tempe, AZ, USA, 9–13 March 2015; pp. 329–334.
8. Shanmugalingam, S.; Ksentini, A.; Bertin, P. DPDK Open vSwitch performance validation with mirroring feature. In Proceedings of the 2016 23rd International Conference on Telecommunications (ICT), Thessaloniki, Greece, 16–18 May 2016; pp. 1–6.
9. Gerhardt, L.; Bhimji, W.; Canon, S.; Fasel, M.; Jacobsen, D.; Mustafa, M.; Porter, J.; Tsulaia, V. Shifter: Containers for HPC. *J. Phys. Conf. Ser.* **2017**, *898*, 082021. [CrossRef]
10. Benedicic, L.; Cruz, F.A.; Madonna, A.; Mariotti, K. Sarus: Highly Scalable Docker Containers for HPC Systems. In *Mining Data for Financial Applications*; Springer Nature: Cham, Switzerland, 2019; pp. 46–60.

11. Höb, M.; Kranzlmüller, D. Enabling EASEY Deployment of Containerized Applications for Future HPC Systems. In *Mining Data for Financial Applications*; Springer Nature: Cham, Switzerland, 2020; Volume 12137, pp. 206–219.
12. Miesch, M. JEDI to Go: High-Performance Containers for Earth System Prediction. In Proceedings of the JEDI Academy, Boulder, CO, USA, 10–13 June 2019.
13. Woo, J.; Jang, J.H.; Hong, T. Performance Analysis of Data Network at the PLSI Global File System. In Proceedings of the Korea Processing Society Conference; Korea Institute of Science and Technology Information: Daejeon, Korea, 2017; pp. 71–72.
14. Piras, M.E.; Pireddu, L.; Moro, M.; Zanetti, G. Container Orchestration on HPC Clusters. *Min. Data Financ. Appl.* **2019**, *11887*, 25–35.
15. Hu, G.; Zhang, Y.; Chen, W. Exploring the Performance of Singularity for High Performance Computing Scenarios. In Proceedings of the 2019 IEEE 21st International Conference on High Performance Computing and Communications, IEEE 17th International Conference on Smart City, IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), Zhangjiajie, China, 10–12 August 2019; pp. 2587–2593.
16. Priedhorsky, R.; Randles, T. Charliecloud. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Denver, CO, USA, 12–17 November 2017; p. 36.
17. Aragon, C.; Dernat, R.; Sanabria, J. Performance Evaluation of Container-based Virtualization for High Performance Computing Environments. *arXiv* **2017**, arXiv:1709.10140.
18. Benedicic, L.; Cruz, F.A.; Schulthess, T.C.; Jacobsen, D. *Shifter: Fast and Consistent HPC Workflows Using Containers*; Cray User Group (CUG): Redmond, WA, USA, 11 May 2017.
19. Madonna, A.; Benedicic, L.; Cruz, F.A.; Mariotti, K. Shifter at CSCS-Docker Containers for HPC. In Proceedings of the HPC Advisory Council Swiss Conference, Lugano, Switzerland, 9–12 April 2018.
20. Mell, P.; Grance, T. The NIST Definition of Cloud Computing, in Recommendations of the National Institute of Standards and Technology. Technical Report, Special Publication 800-145. September 2011. Available online: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf (accessed on 20 January 2021).
21. Lim, S.B.; Woo, J.; Li, G. Performance analysis of container-based networking solutions for high-performance computing cloud. *Int. J. Electr. Comput. Eng. (IJECE)* **2020**, *10*, 1507–1514. [CrossRef]
22. Bureddy, D.; Wang, H.; Venkatesh, A.; Potluri, S.; Panda, D.K. OMB-GPU: A Micro-Benchmark Suite for Evaluating MPI Li-braries on GPU Clusters. In Proceedings of the EuroMPI 2012 Confierence, Vienna, Austria, 23–26 September 2012; pp. 110–120.