Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

TEES: topology-aware execution environment service for fast and agile application deployment in HPC*

Mingtian SHAO, Kai LU[†‡], Wanqing CHI, Ruibo WANG, Yiqin DAI, Wenzhe ZHANG[†‡]

College of Computer, National University of Defense Technology, Changsha 410073, China

[†]E-mail: lukainudt@163.com; zhangwenzhe@nudt.edu.cn

Received June 16, 2021; Revision accepted Oct. 24, 2021; Crosschecked Mar. 2, 2022; Published online June 1, 2022

Abstract: High-performance computing (HPC) systems are about to reach a new height: exascale. Application deployment is becoming an increasingly prominent problem. Container technology solves the problems of encapsulation and migration of applications and their execution environment. However, the container image is too large, and deploying the image to a large number of compute nodes is time-consuming. Although the peer-to-peer (P2P) approach brings higher transmission efficiency, it introduces larger network load. All of these issues lead to high startup latency of the application. To solve these problems, we propose the topology-aware execution environment service (TEES) for fast and agile application deployment on HPC systems. TEES creates a more lightweight execution environment for users, and uses a more efficient topology-aware P2P approach to reduce deployment time. Combined with a split-step transport and launch-in-advance mechanism, TEES reduces application startup latency. In the Tianhe HPC system, TEES realizes the deployment and startup of a typical application on 17 560 compute nodes within 3 s. Compared to container-based application deployment, the speed is increased by 12-fold, and the network load is reduced by 85%.

Key words: Execution environment; Application deployment; High-performance computing (HPC); Container; Peer-to-peer (P2P); Network topology

https://doi.org/10.1631/FITEE.2100284 **CLC number:** TP315

1 Introduction

High-performance computing (HPC) systems have more and more computing resources. Their service scope is gradually expanding, user groups are increasingly complex, and the trend toward diversification of user needs is becoming more and more prominent. In the context of commercialization, timeliness and convenience are always user concerns.

The standard execution environment of an HPC system makes it difficult to directly support the running of user applications. Therefore, users need to deploy the application and to configure the execution environment on the compute nodes. Manually configuring the environment on the compute nodes is too burdensome for the user, and the normal user does not have a significantly enough permission level to configure it freely. Especially with the development of HPC, exascale computing (Djemame and Carr, 2020; Meizner et al., 2020) is coming, and as the number of compute nodes grows, this problem will become more pronounced. In summary, there are mainly the following problems:

1. Environment configuration

User burden is a result of two issues: (1) the user does not have sufficient permission to do any configuration; (2) the user workload is too large when there

[‡] Corresponding authors

[*] Project supported by the National Natural Science Foundation of China (No. 61902405), the Tianhe Supercomputer Project of China (No. 2018YFB0204301), the PDL Research Fund of China (No. 6142110190404), and the National High-Level Personnel for Defense Technology Program, China (No. 2017-JCJQ-ZQ-013)

ORCID: Mingtian SHAO, https://orcid.org/0000-0003-2368-4946; Kai LU, https://orcid.org/0000-0002-6378-7002; Wenzhe ZHANG, https://orcid.org/0000-0002-8798-2195

© Zhejiang University Press 2022

are many compute nodes. In addition, configuring the environment is a tedious process. When the user debugs the application on the login node, the environment configuration of the login node would be completed. Then the user has to individually migrate these configurations to the compute nodes.

2. Deployment overhead and startup latency of the application

Inevitably, the user must deploy the application to compute nodes. This deployment overhead is not obvious when the number of compute nodes is small. Typically, startup latency refers to the time interval between issuing a command and starting the application. Here, we define startup latency as the time interval between the time when the user finishes developing and debugging the application on the login node and the time when the application starts running on the compute node. This latency includes environment configuration time, deployment time, and so on. In practical situations, this definition of startup latency has a more realistic meaning. Of course, users want this latency to be as small as possible. Obviously, as the number of compute nodes increases, the deployment overhead and startup latency increase.

Container technology has been applied to HPC systems to solve these problems (Chen et al., 2018; di Nitto et al., 2020; Höb and Kranzlmüller, 2020; Srirama et al., 2020). Containers, a lightweight virtualization technology, such as Docker (Merkel, 2014; Boettiger, 2015), Shifter (Gerhardt et al., 2017; Belkin et al., 2018), and Singularity (Kurtzer et al., 2017; Godlove, 2019), are favored by many users (de Velp et al., 2020). Containerization enables encapsulation and migration of the application and its dependencies using a container image. This also solves the problem of the user's permission for environment configuration. The user can do any configuration in the container image, but this container-based method still has significant drawbacks.

First, the container image is entirely maintained by the user (Huang et al., 2019). The HPC system is highly customized. The software stacks are highly customized to fit the special hardware in the HPC system. It is difficult for users to maintain the container image by themselves. Second, the container image contains many files in addition to the application and its dependencies. Deploying such an unwieldy container image to compute nodes can be a time-consuming process (Verma et al., 2015).

There are usually two common ways to speed up the deployment of a container image or other files to compute nodes: through shared storage (Harter et al., 2016; Du et al., 2017; Hardi et al., 2018; Zheng et al., 2018) or with a peer-to-peer (P2P) approach (Wang et al., 2017). Both approaches have their own disadvantages in HPC systems. When the number of nodes or the size of the transferred files is too large, the use of shared storage will easily trigger the network transmission bottleneck. In this case, the P2P approach will perform better, but will introduce explosive traffic into the network topology, which will put a lot of stress on the network. The details will be discussed in Section 2.2.

The self-deployed execution environment (SDEE) for HPC (Shao et al., 2022), our previous work, implements a lightweight execution environment that uses only two layers of the overlay file system. SDEE also solves the problem of the user's ability to entirely maintain the execution environment, but the deployment overhead and startup latency of the SDEE are still not satisfactory when the number of compute nodes is large.

To solve these problems, we design the topology-aware execution environment service (TEES) for fast and agile application deployment in HPC. TEES is designed based on SDEE.

TEES combines the advantages of a topology-aware P2P approach with shared storage to greatly reduce deployment overhead. When the number of compute nodes used by the application is small, the execution environment deployment is carried out directly through the shared file system. We design the topology-aware P2P approach to deploy the execution environment when the number is large. The topology-aware P2P approach makes good use of the local topology structure in the HPC system. It improves the P2P transmission speed and reduces the network traffic of the intermediate topology. Specifically, in a test involving a super large number of compute nodes, TEES shows a superior performance.

TEES also reduces the application startup latency through a split-step transport and launch-in-advance mechanism. On the login node, this mechanism divides the user's upper file system into two parts based on dependency analysis of the user's application. The file system is divided into the urgent part and the hysteretic part. The urgent

part is transferred first using the transport scheme described earlier. After the transmission of this part is complete, the execution environment and the user's application are directly launched, and then the hysteretic part is transmitted.

In summary, TEES solves application deployment problems in HPC. It realizes the lightweight execution environment and reduces the burden of application deployment. It supports fast application deployment of different numbers of compute nodes, especially at large and very large numbers. It also reduces the startup latency of the application. At the same time, the work has good portability and is suitable for HPC systems of different numbers of compute nodes. In the Tianhe HPC system, TEES realizes the deployment and startup of a typical application on 17 560 compute nodes within 3 s. TEES is actually 12 times faster compared to a container-based application deployment, and the network load is reduced by 85%.

2 Background

In this section, we introduce several phenomena in the HPC system to highlight the theoretical value and practical significance of our work.

2.1 Network topology

In the hardware design of HPC systems, multiple compute nodes are usually integrated into a node group. As shown in Fig. 1, in the Tianhe HPC system, there are eight compute nodes in a node group, among which there is a proxy node with high-speed network performance. Each cube in Fig. 1 represents a node group, and the black vertex represents the proxy node in the group. The proxy node is directly connected to the upper topological structure, and the other nodes in the group are called slave nodes of the proxy node. The slave nodes are the white vertices of the cube in each node group in Fig. 1. Their network interactions with the upper topological structure need to be forwarded by the network cards of these proxy nodes. The nodes in a node group are physically close to each other. All the slave nodes in a group can quickly obtain data from the proxy node in the same group.

In the network topology, these compute nodes do not have a completely equal status. There are obvious dependencies among them. This topology is common in HPC systems.

In the Sunway TaihuLight HPC system (Dongarra, 2016; Fu et al., 2016), each super node includes 256 Sunway processors (compute nodes) that are fully connected by the super node network.

In the BlueGene/Q HPC system (Haring, 2011; Boyle, 2012), each compute card includes one chip, which includes four processors. Thirty-two compute cards and, optionally, up to two input/output (I/O) cards are packaged on the next-level board which is called the node card. These 32 compute

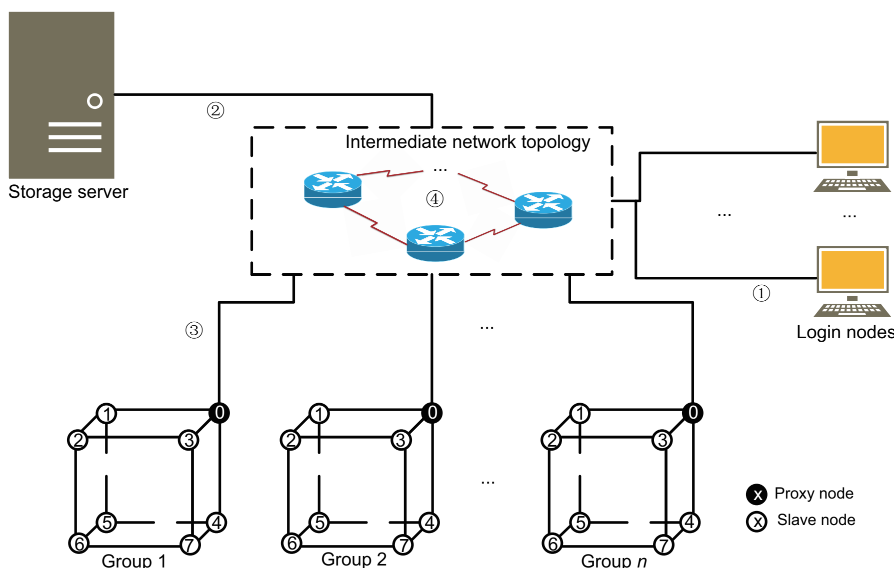

Fig. 1 Network topology of the Tianhe high-performance computing system

cards communicate with the upper network topology through these two I/O cards.

As the scale of the HPC system continues to grow, such topologies will become more common. Making full use of such topology features can reduce network load and improve network transmission performance.

2.2 Data transmission mode

Users commonly need to transfer files from the login node to the compute nodes. Because the user's application itself is a file, the application runs on the compute nodes and the first step is to transfer the application file to the compute nodes.

In the simplest way, the user directly transfers files from the login node to the compute nodes through the network connection between the login node and the compute nodes. We call this "one-to-all" mode. This is fine when the user has only a few compute nodes. However, when the number of compute nodes is large, this method is extremely inefficient, because the login node experiences too much network transmission pressure.

The second way is through shared storage. In the HPC system, compute nodes usually have no local disk but only dynamic random access memory. To achieve data persistence, all nodes can access the shared storage. Shared storage is usually a high-speed storage server or storage node. For example, the storage server in Fig. 1 is the global shared storage. Based on shared storage, it is easy to transfer files from the login node to the compute nodes. The user first copies the file from the login node to the shared storage, and then the compute node copies the file from shared storage to the local.

In such a scenario, the larger the number of compute nodes, the greater the pressure on the shared storage. Even a small number of compute nodes accessing a large file at the same time can put huge pressure on shared storage. This pressure comes mainly from the limited bandwidth of the shared storage. This determines how much data can flow in or out at any given moment. In addition, a large number of compute nodes accessing the shared storage at the same time puts huge pressure on the intermediate topology that connects the compute nodes and the shared storage. Network load capacity is limited, and when a large number of compute nodes make simultaneous requests to storage nodes, it is likely that messages with data will be blocked somewhere in the intermediate topology. In this case, the file transfer efficiency is not high, and the user experience is very poor.

The third way is to use the P2P approach. This is a good way to balance the load on the shared storage network. When there are many compute nodes, this approach can achieve higher efficiency than the second method. The HPC system's resource management tool generally includes the P2P transmission function. For example, sbcast is an efficient P2P transmission function of the resource management tool, SLURM (Yoo et al., 2003). It can quickly copy the user's files from the login node to a large number of compute nodes, but this approach will introduce a lot of unreasonable traffic in the network. We will discuss this in detail in Section 2.3.

2.3 Network load state

When users transfer files from the login node to compute nodes, different transmission modes bring different network load states. We have analyzed the network load state in Fig. 1 under three different transmission modes in Section 2.2. In such a hypothetical scenario, the user copies an application file (file A) from the login node to 5000 compute nodes.

1. Case 1: the user copies file A directly from the login node to compute nodes. In this case, file A flows out of link ① and the flow is repeated 5000 times. The flow also appears 5000 times in the intermediate network topology. File A flows from link ③ to the eight compute nodes in one node group, repeating the flow eight times.

2. Case 2: the user copies file A from the login node to the compute nodes through shared storage. In this case, at the first step, the user copies file A from the login node to the shared storage. The network load is minimum. File A flows out of link ① once and flows in from link ② once. At the second step, the compute nodes copy file A from the shared storage to the local. File A flows out of link ② and the flow is repeated 5000 times. The flow also appears 5000 times in the intermediate network topology. File A flows from link ③ to the eight compute nodes in one node group, repeating the flow eight times. The transfer process at the second step is similar to that in case 1, but it is more efficient, because the network load capacity of shared storage is far greater than that of the login node.

3. Case 3: use a P2P approach. The user copies file A from the login node to 5000 compute nodes. In the intermediate network topology, up to 5000 network transmissions may occur. There are at most eight inflows in link ③. If file A happens to be transferred from one compute node to the next, and these compute nodes are in the same node group, then there will be one less inflow in link ③ and one less network transmission in the intermediate topology. However, most of the time, the parent-child nodes are not in the same node group. At this point, new traffic is generated, which is the outflow traffic of link ③. Most of the time, it is the worst. The child of a slave node in a node group in the P2P transmission structure is a slave node in another node group. Therefore, the transfer of file A between these two compute nodes goes through the proxy nodes in these two node groups. Obviously, in this process, the transfer of file A has traveled a long way. Such traffic is obviously not smart, because the P2P approach does not take into account the special topology of the HPC system.

In all of these cases, we observe redundant and even unintelligent traffic in the network. This provides inspiration for our topology-aware P2P approach design.

3 Design and implementation

This section describes our design choices and the implementation of TEES.

3.1 Main idea

TEES provides a topology-aware execution environment service for the HPC system. TEES starts an execution environment for the user when he/she logs in, and the execution environment is deployed automatically when the user runs applications. In particular, TEES reduces the deployment time of the execution environment and the startup latency of the user's application. At all scales, TEES provides a comprehensive solution, and in particular, optimizes application deployment for large-number compute node deployments.

The framework of TEES is shown in Fig. 2. TEES uses a hierarchical file system and process isolation to create a private execution environment for users. It implements a more lightweight execution environment than the container. Only two overlay file system layers are used to avoid space overhead of a container image. This lightweight design also reduces network transmission stress related to the deployment of the environment.

TEES reasonably combines the advantages of different file transfer modes at different scales. TEES is optimized on a large scale to improve the efficiency of network transmission. When the number of compute nodes is small, TEES directly deploys the application and its execution environment to the compute nodes through shared storage. When the number of compute nodes is large, the P2P transmission is optimized for the specific HPC network topology.

TEES achieves a quick start of the application.

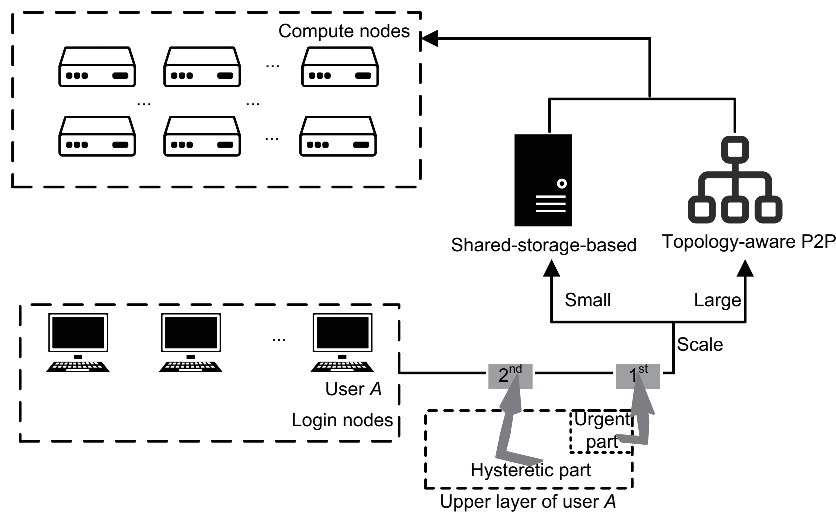

Fig. 2 Framework of the topology-aware execution environment service (TEES)

The container image is usually a large file, but the files related to the current application make up only a very small portion of it. TEES uses a two-layer file system that is much lighter than the container image, but not all the files in the upper layer are required by the current application. TEES splits the user's upper-layer file system according to the current application as shown in Fig. 2. First, the files required for the current application are transferred, then the application is launched, and finally, the remaining files are transferred.

3.2 Execution environment on a node

In terms of the execution environment, we follow the SDEE design. Here is only a brief introduction. TEES creates an isolated process tree for each user, so that users cannot snoop on the processes of others. When users exit, TEES needs only to kill the root process of the process tree. TEES uses an overlay file system with only two layers. The "/" directory of the node is used as the lower layer of the overlay file system, and an empty directory is superimposed as the upper layer for each user. This design enables users and system administrators to jointly maintain the system environment configuration.

When implementing the automatic deployment of the execution environment, TEES needs only to synchronize the user's upper layer to the corresponding compute nodes for the compute nodes to have the same application execution environment as the login node. This reduces the burden on the user to manually configure the execution environment on the compute nodes and realizes the user's privacy protection at the same time.

3.3 Rapid deployment

Observing the unique topology in the HPC system, the shared storage, and network bottlenecks, TEES designs a complete set of rapid deployment solutions.

3.3.1 Advantage combination and scheme selection

Obviously, when the number of compute nodes used by an application is small, using shared storage has obvious advantages. However, when the number of compute nodes is large, even if the transferred file size is small, it can easily cause traffic congestion. Therefore, shared storage has obvious advantages when the number of compute nodes is small.

When the number of compute nodes used by the application is large, the shared storage no longer has a performance advantage and brings significant network traffic to the intermediate topology of the HPC system. Such traffic is likely to affect the state of the network and even affect other users' applications. At this point, P2P transmission is a good choice. We need only to establish a threshold to distinguish whether the number of nodes is large or small. The P2P approach requires each node that receives the file to transfer the file to the next node, which is a good way to solve the shared storage bottleneck problem. However, traditional P2P introduces a lot of traffic in the network topology, so we optimize the P2P approach in this case.

3.3.2 Topology-aware P2P

When the number of compute nodes is large, TEES chooses the P2P approach. According to the unique network topology in HPC systems, we design a topology-aware P2P transmission approach.

First, several compute nodes are integrated into a node group. In our Tianhe platform, eight compute nodes are integrated into a node group. However, only one of the eight nodes (proxy node) has a high-speed network card and is directly connected to the intermediate topology. Therefore, this node has better network performance, and the interactions among the other seven nodes and the intermediate topology need to pass through this proxy node.

In consideration of this special in-group topology, we design topology-aware P2P. We maintain a list of the proxy nodes and a list of slave nodes of each proxy node. When a user submits a job (runs an application), the node list used by the user's application is analyzed to generate a tree structure of P2P transmission. The user's login node is regarded as the root node of the tree. Several important elements in the tree are as follows:

1. Proxy nodes

Because the proxy nodes have better network transmission performance, we arrange them at the top of the tree, as close to the root node as possible.

2. Slave nodes

For a slave node in the application node list, if its proxy node is included in this node list, the slave node is used directly as its child node. If the proxy node is not in the application node list and the proxy

node is at the idle state, we calculate its degree of utilization (that is, the number of its slave nodes in the current application node list). If the degree of utilization is larger than half of the number of the nodes in this node group, we add this idle proxy node to the P2P tree and temporarily set the proxy node to the allocated state. Its slave nodes in the node list are also added as child nodes of this proxy node in the tree. If the proxy node is not idle, we call such slave nodes orphan nodes and finally add these nodes at the last level of the tree. Because the user application node list is usually continuous, in most cases, only the beginning and end of the node list may encounter a situation in which the proxy node is not in the node list.

3. Treewidth

We set the treewidth according to the performance of each compute node (such as cores of CPU) to achieve the highest transmission performance. For example, we set the treewidth to 15 in Fig. 3, which means that the proxy node can pass the file to its seven slave nodes and eight other proxy nodes in the next layer at most.

According to the above conditions, we present the structure of the topology-aware P2P tree (Fig. 3) and the algorithm for constructing the tree (Algorithm 1).

After this tree structure has been created on the login node, we also pass the tree structure to the next layer of nodes while transferring the file. Then, each node finds its child nodes to continue transmission according to the tree structure and waits for the signal that the transmission is complete. After the node receives the transmission completion signals of all child nodes, the node generates the transmission completion signal and returns it to its parent node. Finally, after the login node receives confirmation signals from the first-layer nodes, which means that the entire transmission process has been completed, the temporarily occupied proxy nodes are set to idle state.

Because the P2P transmission process is speedy, the few proxy nodes that we temporarily set to the allocated state will have little effect on the system.

Algorithm 1 Topology-aware P2P tree generation

Require: compute node list required by application, $N_{\rm c}$; proxy node list of all nodes, $N_{\rm p}$; treewidth, W; half of the number of the nodes in a node group, t
Ensure: tree structure of topology-aware P2P, tree

```
 1: Divide Nc by node group to Nb and No /* The elements
    of Nb are the nodes in the same node group and the
    proxy node of this group is in Nc; The elements of No
    are the slave nodes in the same node group and the proxy
    node of this group is in Nc */
 2: Set the current login node as the root node of the tree
 3: for nodeset in Nb do
 4:    pn ← proxy node of nodeset
 5:    sn ← slave nodes of nodeset
 6:    Add pn to tree /* If no father node is specified, it
       would find a proxy node with fewer than W children
       as the father of pn */
 7:    Add each node in sn to tree with father = pn
 8: end for
 9: for nodeset in No do
10:    if len(nodeset) > t then
11:       pn ← proxy node of this nodeset
12:       if pn is idle state then
13:          Set the state of pn to allocated
14:          Add pn to tree
15:          Add each node in nodeset to tree with father=pn
16:       else
17:          Add each node in nodeset to tree
18:       end if
19:    else
20:       Add each node in nodeset to tree
21:    end if
22: end for
```

3.4 Quick startup

The TEES startup latency includes the time required for environment deployment and execution environment startup. This is the time interval between the time when the user issues an application run command and the time when the application actually starts to run. Because TEES implements automatic deployment that is transparent to users, this startup latency is consistent with what we defined in Section 1. The TEES execution environment starts faster than containers such as Docker and Singularity. The upper layer of the overlay file system that we needed to deploy is also more lightweight than the container image. Because we find that not all files in the upper layer are necessary for the current user's application, we create the following design:

3.4.1 Split-step transport and launch-in-advance mechanism

We divide the upper layer of the user's overlay file system into two parts according to the user's current application: an urgent part and a hysteretic part.

1. Urgent part

The user's application is, of course, included in

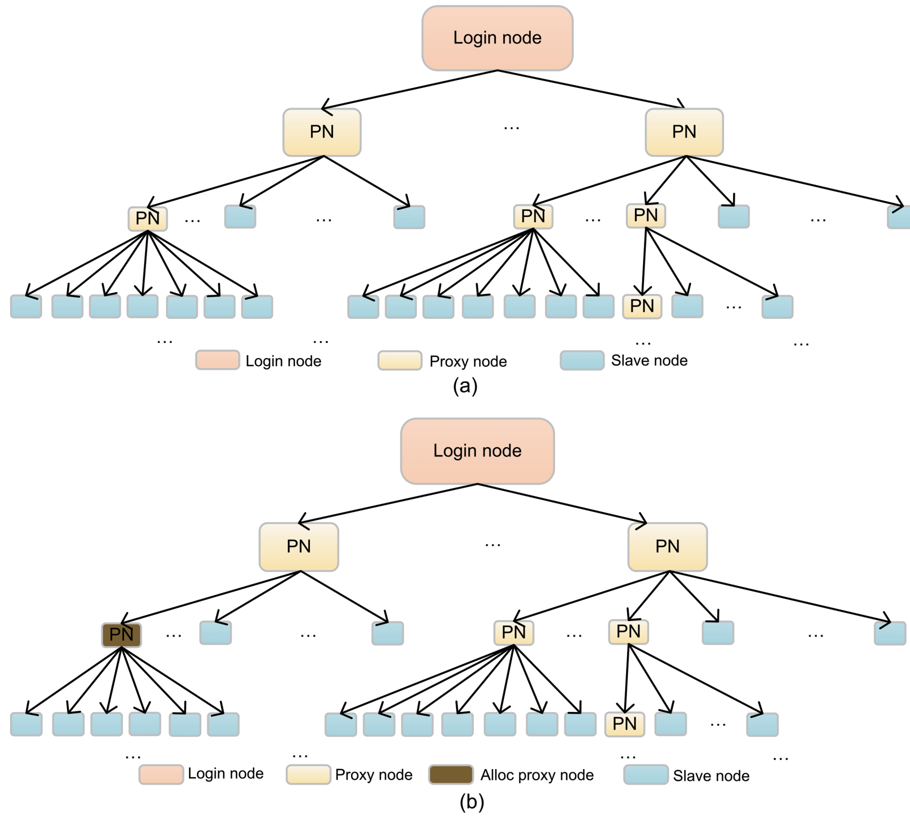

Fig. 3 Structure of the topology-aware peer-to-peer (P2P) tree: (a) structure at the ideal state; (b) structure when P2P tree temporarily occupies the proxy node

this part. We analyze the dynamic dependencies of the current application, and if the application's dependencies appear in the upper layer, they are added to the urgent part.

2. Hysteretic part

This part of the user's upper-layer file system excludes the urgent part.

We use the scheme in Section 3.3 to give priority to the transmission of the urgent part. When this transmission is completed, we directly start the execution environment on the corresponding compute nodes to start the application. Then we transmit the hysteretic part.

3.4.2 File dependencies

Now that the application has started, is it necessary to transfer the hysteretic part? We analyze only the dynamic dependencies of the application and add them to the urgent part. However, file dependencies in the user's application are difficult to diagnose. In most cases, the user's application is just a binary executable file, and we cannot know which files it needs to read and write. If this application does read or write files, the files are either in the hysteretic part or are local files on the compute nodes.

Because the user's application starts after the transfer of the urgent part is completed, if the application does not need to read or write files in the hysteretic part, the application can run smoothly until the end. If the application needs to read and write files in the hysteretic part, we implement the hook of the file read and write system call. When a file needs to be read and written, we first determine the existence of the file, and if it does exist, we go directly to the natural system call. If it does not exist, it means that the file is still being transferred in the hysteretic part. At this time, the read and write operations for this file are blocked until the file in the hysteretic part is transferred.

Although it is quite possible for the application to read and write files, our launch-in-advance mechanism still makes sense, because the application may have a lot of other work to do before it reads or writes files.

3.5 Portability and scalability

In HPC systems, TEES has good portability. The execution environment on the node can run on the Linux operating system and is suitable for the current mainstream kernel version. The design of rapid application deployment and startup is universally applicable in HPC systems.

The TEES transmission scheme can support different scales of the HPC system. The threshold can be set according to the actual condition of the HPC system, and if you want to bypass shared storage completely, you just need to set the threshold to 0. If you want to use shared storage entirely, set the threshold to a maximum value. Or, if you want to take full advantage of the two methods to achieve the maximum transmission efficiency, you need only to measure the transmission performance equilibrium point of the two methods on the actual system, and then set the threshold to this equilibrium point. The treewidth of the P2P transmission structure can also be determined according to the computing resources of actual compute nodes, to maximize performance.

To take full advantage of topology-aware P2P, it is necessary only to maintain the list of proxy nodes and their slave nodes according to the topological relationship in the node group in the actual HPC system.

TEES directly uses shared storage for transmission when the number of compute nodes is small. If there is a way to optimize this approach in the future, it will be easily integrated into TEES. In other words, TEES has good scalability.

3.6 Safety and reliability

The TEES execution environment has been designed to limit the user's permissions and has been well verified. Users have the right to customize their own environment without affecting the underlying standard system environment. At the same time, each node runs a daemon, which has limited functions and is responsible only for starting, stopping, and deleting the execution environment and carrying out P2P transmission. Users cannot upgrade their privileges through this daemon.

TEES is reliable when the network is unobstructed. If there is a problem in the network, such as a node group that cannot communicate with the outside world for some physical reason, TEES does not consider this situation. This situation should be quickly noticed by the monitoring system, and these nodes should be removed from the idle node list. In this case, the user's node list will not include these failure nodes.

The TEES P2P transmission can also be used as an auxiliary tool for network status monitoring. For example, transmitting an empty file to all nodes is a convenient way. If some nodes fail, the transmission will be blocked in the P2P transport tree. After the timeout, the transmission path of network failure will be reported to the root node.

4 Use cases and comparison

In this section, we introduce some typical use cases for TEES to highlight its contribution.

4.1 TEES deployment

The system administrator (root user) first needs to deploy TEES on the HPC system. The administrator needs to start a daemon on each login node and compute nodes and configure things such as the threshold of the transport mode, the treewidth of the P2P tree structure, and the list of proxy nodes. This daemon is responsible for starting and killing the execution environment. Topology-aware P2P transport is also its job. The daemon of the login nodes is also responsible for generating the topology-aware P2P tree structure for the list of application nodes and dividing the user's upper layer into the urgent part and hysteretic part.

4.2 Use cases

When a user logs into the login node, the TEES daemon on the login node starts an execution environment for the user. The user logs into this private execution environment. In this execution environment, the user can directly use the standard system environment. At the same time, the user has the right to make any customization. The user can develop and debug the application and configure the environment freely. All these happen in the upper-layer file system. The changes made by the user do not affect the real system environment.

When the user runs the application by submitting it to the resource management system, such as SLURM and PBS (Feng et al., 2007), the

daemon on the login node will perform the following functions: analyze the list of compute nodes, select the transport method (shared-storage-based or topology-aware P2P), and generate the tree structure if using the topology-aware P2P approach. At the same time, the daemon will perform a dependency analysis on the user application and divide the upper layer into the urgent part and hysteretic part. The urgent part will then be immediately deployed on compute nodes. The execution environment is started at each node and the application runs. Finally, the hysteretic part is transmitted.

The whole process is transparent to the user. From the user's point of view, only the application development and environment configuration are required at the login node. Then the application can run directly on the compute node through the resource management system. The entire deployment and startup process is fast, resulting in a good user experience.

4.3 Comparison

In this subsection, we discuss our comparison of TEES with previous works, primarily involving the user's manual work and the container-based approach, from many dimensions (Table 1). All of these approaches address the deployment of applications and their execution environments, but obviously, manual work is the least efficient and the permissions of non-root users are limited. The system administrator also needs to complete a lot of work. The container-based method solves the problem of user permissions, but heavy container images need to be maintained by users and the deployment process exerts enormous pressure on the network. TEES addresses the weaknesses of both methods, and more importantly, compatibility with resource management systems such as SLURM makes the deployment process transparent to the user, which dramatically improves productivity. The following section verifies the advantages of TEES (Table 1).

5 Evaluation

In this section, we evaluate the TEES deployment time, network load status, and application startup latency.

5.1 Methodology

We evaluated TEES on the latest Tianhe HPC system. A total of 17 560 compute nodes were used. Each node was equipped with an FT processor. The operation system on each node was the customized HPC Kylin Linux. Eight compute nodes were assembled in one node group, including one proxy node and seven slave nodes. The proxy node was equipped with a self-developed high-speed network card to connect with the intermediate topology directly. All

Table 1 Comparison of TEES with previous works

Performance	TEES	Container-based method (Docker, Singularity, etc.)	Manual work on bare-metal
Main problem addressed	Deployment of application and its execution	Deployment of application and its execution environment	
Size of execution environment	Lightweight upper-layer file system	Cumbersome container image	Small, but scattered
Use mode	Transparent to users	Manually maintain and run the container	Completely manual
Environment configuration permissions	High (configure freely and all these happen in the upper-layer file system)	High (configure freely in the Docker image)	Low (only non-root permission)
Deployment time	Short	Long	Very long
Application startup latency	Low	High	Very high (most of the time is wasted on deployment)
Network load	Low (with topology-aware P2P approach)	High (with typical approach)	High (with typical approach)
SLURM-friendly	Yes	No	No

compute nodes and login nodes had access to the same shared storage. The SLURM version of the system was 19.05.7.

The login node had the same processor and memory configuration as each compute node. The login node was also equipped with a self-developed high-speed network card. Moreover, both the login node and the compute nodes had the same preset standard system environment. We tested and verified the performance of TEES on this experimental platform.

We set the threshold of scheme selection to 512; this means that when the number of compute nodes is larger than 512, TEES uses the topology-aware P2P approach. This threshold is the performance equilibrium point that we tested on the experimental platform. To test the effect of topology-aware P2P, we set the treewidth to 15. In addition to sbcast, we added a topology-unaware P2P transmission method: random-P2P. The tree shapes of random-P2P and topology-aware P2P are expected to be exactly the same. The only difference between random-P2P and topology-aware P2P is that the positions of the proxy node and slave nodes in the tree structure of random-P2P are random.

In summary, we tested six methods in total: topology-aware P2P, random-P2P, sbcast, one-to-all, shared-storage-based, and TEES. Topology-aware P2P, random-P2P, and sbcast are P2P methods. One-to-all is a centralized parallel copy method. Shared-storage-based method requires the support of high-speed storage servers or storage nodes. TEES is a scale-adaptive solution that combines the benefits of topology-aware P2P and shared-storage-based methods. We tested each of these six methods for deployment time, network load status, and application startup latency, and highlighted the advantages of TEES.

5.2 Deployment time

For TEES, the deployment time is the transfer time of the upper file system. The TEES execution environment is very lightweight, and TEES adopts a split-step transmission mechanism. In most cases, the size of urgent part deployed by TEES does not exceed 16 MB. When the hysteretic part is transmitted, the application is already started. It can be said that the deployment time of the hysteretic part transmission is hidden. Therefore, we tested only the deployment time of the urgent part.

For container-based application deployment, the deployment time is the transfer time of the container image. Python is one of the most popular programming languages today. We tested the time to deploy a Python image. Python, an official Docker image in DockerHub, which is downloaded more than one billion times, is about 333 MB in size. It would be larger after the user encapsulates his/her own application and the dependencies.

We chose 15, 120, 1080, and 8760 as the cluster sizes for the test. When the number of nodes is exactly one of these values, the topology-aware P2P tree is full. If the cluster size is between two of these numbers, the experimental results would be similar to those of the full tree case.

As shown in Fig. 4, we used several methods to test the TEES deployment time and the container at different numbers of the compute nodes. The methods we used were, one-to-all, shared-storage-based, sbcast, random-P2P, and topology-aware P2P. We have also marked the selection of TEES with a dotted line in Fig. 4.

It can be seen from the results that the deployment time of the one-to-all method increases linearly with the increase of the number of compute nodes. This approach is unacceptably inefficient when the

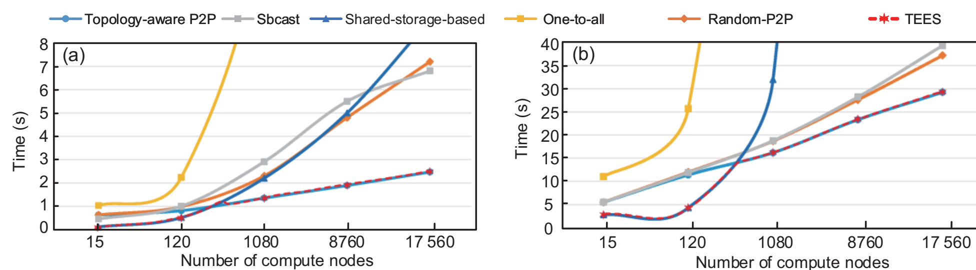

Fig. 4 Deployment time of the six transmission methods when the size of the file deployed is 16 MB (a) and 333 MB (b)

number of compute nodes counted exceeds 1080. The shared-storage-based approach shows strong vitality when the number of compute nodes is small. This also depends on the size of the file. However, it is not difficult to see that when the number of compute nodes is small, the shared-storage-based method is more efficient than any other method.

For the three P2P methods, the topology-aware P2P always has the shortest deployment time. Specifically with the increase of the number of compute nodes, the advantage of the topology-aware P2P method becomes more and more obvious. In terms of the TEES application deployment, topology-aware P2P is 65% faster than that of random-P2P and 63% faster than sbcast when the number of compute nodes reaches 17 560. As for container-based deployment, topology-aware P2P is 21% faster than random-P2P and 25% faster than sbcast.

The difference between a container deployment method and a TEES deployment method is the size of the files being transferred. The container image size is much larger than that of the urgent part of TEES. We have also seen that topology-aware P2P has better acceleration when the size of the file transferred is small. This is because a connection needs to be established before a file can be transferred between two nodes. The process is time-consuming. For small-size files, the process of establishing a connection takes up a large portion of the total time. Topology-aware P2P reduces the time to establish a connection, because most of the connections are inside a node group. When the size of the file is large, the advantage of establishing a connection rapidly is no longer obvious because the file itself takes a long time to transfer. Consequently, in the case of small-size files, topology-aware P2P has a better acceleration effect.

TEES is the best choice in most cases. Only in the vicinity of the threshold, there may be some slight fluctuations.

5.3 Network load status

We conducted traffic monitoring on the connection link between the high-speed network card of the proxy node and the intermediate topology. This value can well reflect the pressure state of the intermediate network topology. The traffic includes both outgoing and incoming flows. We monitored traffic for the transmission methods in Section 5.2. For each transmission method, we collected the traffic monitoring results for the proxy node. The results are shown in Fig. 5.

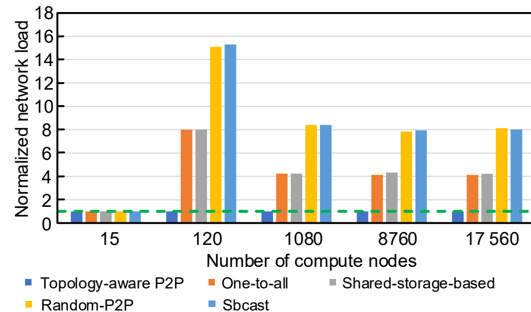

Fig. 5 Network load status of the five transmission methods

For different numbers of compute nodes, topology-aware P2P has the lowest network load, and we set it as the baseline to determine the network load of the other methods. As we can see from Fig. 5, one-to-all and shared-storage-based methods have similar network loads. The difference between these two is that in the one-to-all approach, the login node experiences a lot of network stress. In the shared-storage-based method, this network pressure is transferred to the shared storage. Compared to these two approaches, the topology-aware P2P approach reduces network load by 75% in large-scale node cases.

The network loads are similar for random-P2P and sbcast, because in these two methods, the probability that two nodes in the same node group are in a parent-child relationship in the P2P tree is minuscule. Specifically when the number of compute nodes is large, this probability is even more unlikely. Compared to these two approaches, topology-aware P2P reduces network load by more than 85%.

5.4 Application startup latency

We tested the application startup latency of TEES and container-based methods. For TEES, the startup latency of an application is the time interval between the user submitting the run command to the resource management system and the start of the application, including the time required to divide the user's upper-layer file system, deploying the urgent part, and starting the execution environment. For the container-based approaches, the application startup latency includes the time of container

image encapsulation, container image deployment, and container startup.

On 17 560 compute nodes, we tested the TEES and container-based approaches for application startup latency. For the deployment method of the container-based approach, we used traditional P2P sbcast and topology-aware P2P. The results are shown in Fig. 6. The abscissa represents time and the ordinate represents the percentage of the startup progress.

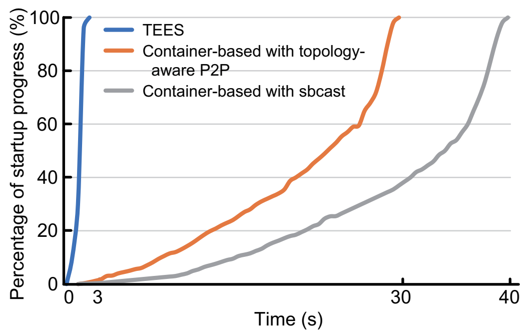

Fig. 6 Application startup latency of the TEES and container-based methods (the number of compute nodes is 17 560)

From the results, we can see that TEES realizes the deployment and startup of a typical application on 17 560 compute nodes within 3 s. The container-based approach, which uses the sbcast deployment method, has an application startup latency of about 40 s, which is 12 times slower than TEES. The topology-aware P2P method used in the container-based approach can reduce the startup latency by about 25%.

6 Discussion

With the development of HPC systems, the number of nodes is gradually increasing, and the cost of application deployment becomes more and more important. To solve these problems, in this study we propose TEES. TEES is a topology-aware execution environment service that enables fast and agile application deployment on HPC systems.

The topology-aware P2P approach is designed in TEES. It realizes fast file transfer with a large number of compute nodes, and effectively reduces the network load. Good results are obtained in our experiment. Notice that in the experiment, the treewidth was set to 15, and the 17 560 compute nodes did not actually fill the last layer of the tree. According to this rule, when the number of compute nodes reaches 68 160, the last layer of the tree structure is occupied. Therefore, deployment time at this scale should not be far from the results with 17 560 compute nodes. We can safely guess that TEES can deploy and start a typical application in about 3 s with 68 160 compute nodes. This will have even greater significance on larger-scale HPC systems in the future.

New requirements may arise in the future, such as the use of other resource management systems, like PBS, in HPC systems. TEES has been matched to SLURM, which is used as the resource management system in the Tianhe series of HPC systems. If PBS is used in the future, TEES will be able to adapt seamlessly and quickly by simply implementing the corresponding standard interface, because TEES does not conflict in any way with the resource management system workflow.

We also note the potential direction of optimization for TEES. As HPC hardware resources become increasingly independent, we are fully capable of directly invoking the underlying network drivers for file distribution and data transfer, thus bypassing the bloated network protocol stack and resulting in performance improvements. In addition, through functional extensions, TEES could provide a universal tool for file distribution and data transfer for large-scale HPC systems, which would improve the productivity of the system administrator in the maintenance and updating of the system environment. These may be our future research directions.

During the deployment of an application using TEES, a failure of one compute node, such as a network failure or unexpected power failure, may cause problems for TEES. Although the probability of such an event is very low and the problem should be detected and alerted by the resource management system in advance, if TEES had the ability to handle these emergencies, it would be better. Adding mechanisms such as proactive reporting of failed nodes and interactive solutions to compute node replacement could also be a possible direction in future TEES improvements.

We mentioned the scalability of TEES in Section 3.5, which means that TEES could facilitate future optimization in function expansion and performance improvement.

7 Related works

Application deployment has been a common problem in HPC. Specifically in recent years, with the increasing scale of HPC systems, this problem is becoming more and more prominent. The traditional solution relies mainly on manual work by users, but it is clear that manual work is no longer appropriate in large-scale HPC systems. Container technologies, such as Docker (Merkel, 2014; Boettiger, 2015) and Singularity (Kurtzer et al., 2017; Godlove, 2019), package the application and its dependencies as a container image. Therefore, the application can be deployed using the container image. Container technology is also an excellent solution to the problem of user permissions. Users can customize the environment freely in the container image.

Deploying a bulky container image to a large number of compute nodes is still a heavy task, which also puts a tremendous strain on the network topology of HPC systems. Therefore, there is a lot of research dedicated to optimizing the deployment of container images. Cider (Du et al., 2017) is a novel deployment system to enable rapid container deployment in a high concurrent and scalable manner. FID (Wang et al., 2017) is able to accelerate the speed of distributing Docker images by taking full advantage of the bandwidth of not only the Docker registry but also other nodes in the cluster. CFS (Liu et al., 2019) is a distributed file system for large-scale container platforms, which supports both sequential and random file access with optimized storage for both large- and small-size files. DADI (Li et al., 2020) is a block-level image service for increased agility and elasticity in deploying applications.

All of these works optimize the deployment of container images, which are difficult to maintain in HPC systems due to highly customized hardware and software. These methods are therefore not as useful in HPC as the TEES approach proposed here.

8 Conclusions

In this study, we proposed TEES, a topology-aware execution environment service, for fast and agile application deployment in HPC. TEES provided a private execution environment for each user in the HPC system, and realized rapidly automatic deployment of applications and their execution environments. We designed a topology-aware P2P approach in TEES to reduce deployment time. There was also a split-step transport and launch-in-advance mechanism in TEES to reduce the startup latency of the application. Experimental results showed that TEES is faster and can effectively reduce network load compared to traditional container-based application deployments.

Contributors

Mingtian SHAO designed the research. Kai LU, Wanqing CHI, Ruibo WANG, Yiqin DAI, and Wenzhe ZHANG improved the design. Mingtian SHAO and Wenzhe ZHANG implemented the system. Mingtian SHAO drafted the paper. Yiqin DAI helped organize the paper. Mingtian SHAO revised and finalized the paper.

Acknowledgements

The authors wish to thank Yong DONG and Hao HAN for their help in the system debugging, and Zhenwei WU for improving the paper.

Compliance with ethics guidelines

Mingtian SHAO, Kai LU, Wanqing CHI, Ruibo WANG, Yiqin DAI, and Wenzhe ZHANG declare that they have no conflict of interest.

References

Belkin M, Haas R, Arnold GW, et al., 2018. Container solutions for HPC systems: a case study of using shifter on blue waters. Proc Practice and Experience on Advanced Research Computing, Article 43. https://doi.org/10.1145/3219104.3219145

Boettiger C, 2015. An introduction to Docker for reproducible research. *SIGOPS Oper Syst Rev*, 49(1):71-79. https://doi.org/10.1145/2723872.2723882

Boyle PA, 2012. The BlueGene/Q supercomputer. Proc 30th Int Symp on Lattice Field Theory, Article 20. https://doi.org/10.22323/1.164.0020

Chen JY, Guan Q, Liang X, et al., 2018. Build and execution environment (BEE): an encapsulated environment enabling HPC applications running everywhere. IEEE Int Conf on Big Data, p.1737-1746. https://doi.org/10.1109/BigData.2018.8622572

de Velp GE, Rivière E, Sadre R, 2020. Understanding the performance of container execution environments. Proc 6th Int Workshop on Container Technologies and Container Clouds, p.37-42. https://doi.org/10.1145/3429885.3429967

di Nitto E, Gorroñogoitia J, Kumara I, et al., 2020. An approach to support automated deployment of applications on heterogeneous cloud-HPC infrastructures. Proc 22nd Int Symp on Symbolic and Numeric Algorithms for Scientific Computing, p.133-140. https://doi.org/10.1109/SYNASC51798.2020.00031

Djemame K, Carr H, 2020. Exascale computing deployment challenges. Proc 17th Int Conf on the Economics of Grids, Clouds, Systems, and Services, p.211-216. https://doi.org/10.1007/978-3-030-63058-4_19

Dongarra J, 2016. Report on the Sunway TaihuLight System. UT-EECS-16-742, University of Tennessee, Tennessee, USA.

Du L, Wo TY, Yang RY, et al., 2017. Cider: a rapid Docker container deployment system through sharing network storage. IEEE 19th Int Conf on High Performance Computing and Communications; IEEE 15th Int Conf on Smart City; IEEE 3rd Int Conf on Data Science and Systems, p.332-339.

Feng HH, Misra V, Rubenstein D, 2007. PBS: a unified priority-based scheduler. Proc ACM SIGMETRICS Int Conf on Measurement and Modeling of Computer Systems, p.203-214. https://doi.org/10.1145/1254882.1254906

Fu HH, Liao JF, Yang JZ, et al., 2016. The Sunway TaihuLight supercomputer: system and applications. *Sci China Inform Sci*, 59(7):072001. https://doi.org/10.1007/s11432-016-5588-7

Gerhardt L, Bhimji W, Canon S, et al., 2017. Shifter: containers for HPC. *J Phys Conf Ser*, 898:082021. https://doi.org/10.1088/1742-6596/898/8/082021

Godlove D, 2019. Singularity: simple, secure containers for compute-driven workloads. Proc Practice and Experience in Advanced Research Computing on Rise of the Machines, Article 24. https://doi.org/10.1145/3332186.3332192

Hardi N, Blomer J, Ganis G, et al., 2018. Making containers lazy with Docker and CernVM-FS. *J Phys Conf Ser*, 1085(3):032019. https://doi.org/10.1088/1742-6596/1085/3/032019

Haring R, 2011. The Blue Gene/Q Compute chip. IEEE Hot Chips 23 Symp, p.1-20. https://doi.org/10.1109/HOTCHIPS.2011.7477488

Harter T, Salmon B, Liu R, et al., 2016. Slacker: fast distribution with lazy Docker containers. Proc 14th USENIX Conf on File and Storage Technologies, p.181-195.

Höb M, Kranzlmüller D, 2020. Enabling EASEY deployment of containerized applications for future HPC systems. Proc 20th Int Conf on Computational Science, p.206-219. https://doi.org/10.1007/978-3-030-50371-0_15

Huang Z, Wu S, Jiang S, et al., 2019. FastBuild: accelerating Docker image building for efficient development and deployment of container. 35th Symp on Mass Storage Systems and Technologies, p.28-37. https://doi.org/10.1109/MSST.2019.00-18

Kurtzer GM, Sochat V, Bauer MW, 2017. Singularity: scientific containers for mobility of compute. *PLoS ONE*, 12(5):e0177459. https://doi.org/10.1371/journal.pone.0177459

Li HB, Yuan YF, Du R, et al., 2020. DADI: block-level image service for agile and elastic application deployment. USENIX Annual Technical Conf, p.727-740.

Liu HF, Ding W, Chen Y, et al., 2019. CFS: a distributed file system for large scale container platforms. https://arxiv.org/abs/1911.03001

Meizner J, Nowakowski P, Kapala J, et al., 2020. Towards exascale computing architecture and its prototype: services and infrastructure. *Comput Inform*, 39(4):860-880. https://doi.org/10.31577/cai_2020_4_860

Merkel D, 2014. Docker: lightweight Linux containers for consistent development and deployment. *Linux J*, 2014(239):2.

Shao MT, Lu K, Zhang WZ, 2022. Self-deployed execution environment for HPC. *Front Inform Technol Electron Eng*, early access. https://doi.org/10.1631/FITEE.2100016

Srirama SN, Adhikari M, Paul S, 2020. Application deployment using containers with auto-scaling for microservices in cloud environment. *J Netw Comput Appl*, 160: 102629. https://doi.org/10.1016/j.jnca.2020.102629

Verma A, Pedrosa L, Korupolu M, et al., 2015. Large-scale cluster management at Google with Borg. Proc 10th European Conf on Computer Systems, Article 18.

Wang KJ, Yang Y, Li Y, et al., 2017. FID: a faster image distribution system for Docker platform. IEEE 2nd Int Workshops on Foundations and Applications of Self* Systems, p.191-198. https://doi.org/10.1109/FAS-W.2017.147

Yoo AB, Jette MA, Grondona M, 2003. SLURM: simple Linux utility for resource management. Proc 9th Int Workshop on Job Scheduling Strategies for Parallel Processing, p.44-60. https://doi.org/10.1007/10968987_3

Zheng C, Rupprecht L, Tarasov V, et al., 2018. Wharf: sharing Docker images in a distributed file system. Proc ACM Symp on Cloud Computing, p.174-185. https://doi.org/10.1145/3267809.3267836