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

Self-deployed execution environment for high performance computing*

Mingtian SHAO, Kai LU‡, Wenzhe ZHANG

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

E-mail: shaomt@nudt.edu.cn; lukainudt@163.com; zhangwenzhe@nudt.edu.cn

Received Jan. 9, 2021; Revision accepted Feb. 14, 2021; Crosschecked Jan. 19, 2022; Published online Mar. 4, 2022

Abstract: Traditional high performance computing (HPC) systems provide a standard preset environment to support scientific computation. However, HPC development needs to provide support for more and more diverse applications, such as artificial intelligence and big data. The standard preset environment can no longer meet these diverse requirements. If users still run these emerging applications on HPC systems, they need to manually maintain the specific dependencies (libraries, environment variables, and so on) of their applications. This increases the development and deployment burden for users. Moreover, the multi-user mode brings about privacy problems among users. Containers like Docker and Singularity can encapsulate the job's execution environment, but in a highly customized HPC system, cross-environment application deployment of Docker and Singularity is limited. The introduction of container images also imposes a maintenance burden on system administrators. Facing the above-mentioned problems, in this paper we propose a self-deployed execution environment (SDEE) for HPC. SDEE combines the advantages of traditional virtualization and modern containers. SDEE provides an isolated and customizable environment (similar to a virtual machine) to the user. The user is the root user in this environment. The user develops and debugs the application and deploys its special dependencies in this environment. Then the user can load the job to compute nodes directly through the traditional HPC job management system. The job and its dependencies are analyzed, packaged, deployed, and executed automatically. This process enables transparent and rapid job deployment, which not only reduces the burden on users, but also protects user privacy. Experiments show that the overhead introduced by SDEE is negligible and lower than those of both Docker and Singularity.

Key words: Execution environment; High performance computing; Light-weight; Isolation; Overlay
https://doi.org/10.1631/FITEE.2100016 **CLC number:** TP315

1 Introduction

Traditional high performance computing (HPC) is designed mainly to provide super-high computing speed for scientific computing. It provides a standard preset environment to meet the needs of scientific computing. However, with the rapid development of use demand, modern HPC not only is being applied to scientific computing, but also supports various applications such as big data and artificial intelligence (Wang B et al., 2020). The needs of users are increasingly diverse. The traditional standard preset environment can no longer meet the needs of users.

The purpose of using HPC is for users to take advantage of its supercomputing performance to quickly obtain the execution results of their own programs. Because the HPC system is highly customized in terms of both hardware and software, users can develop and deploy applications only on the login node of the HPC system, not in their

‡ Corresponding author

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

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

desktop environment.

However, the preset environment of traditional HPC systems cannot meet the diverse needs of users. Thus, users of HPC systems need a lot of manual work to deploy their own applications and customize their own execution environment. This puts a heavy burden on users. In current HPC systems, some shortcomings can be identified:

1. Workload. Users need to develop and debug their program and configure the environment on the login node, but the job is ultimately running on the compute nodes. Some unique libraries and dependencies of user jobs do not exist on the compute nodes and need to be manually configured by users. For example, the user has a program that links to a special library. This user has to put this special library somewhere that he/she can access ("/usr/lib" is not writable for non-root users), and then he/she needs to recompile the program linked to that place. This process greatly affects efficiency and is a heavy burden on users. The user can also contact the system administrator to help configure the environment, but this process requires a lot of communication and an uncertain amount of time waiting for the administrator to complete the configuration.

2. Privacy. Multiple users may share the same login node, so user files and processes are visible to other users. For example, if user A and user B log in to the same login node, user A can see what user B is running (using the ps command) and user B's files. In this multiuser mode, user privacy cannot be protected. In most cases, users do not want their processes and files to be exposed.

3. System environment. Different users are likely to have different requirements for the system environment; for example, different users may use different versions of glic. Obviously, the preset environment of the system cannot be satisfactory for all users. If users are given high configuration privileges, it is likely to have an impact on the security and stability of the system, and because multiple users share the landing node, the configuration of some users may adversely affect other users. For example, if user A changes a library in the system, user B will have problems when using the same library.

The above shortcomings are well known, and we now have many tools to solve some of them, but not an integrated solution for the HPC scenario to solve them all.

The development of container technology makes it easier for users to deploy their job's execution environment, but it also introduces additional burdens. The container encapsulates the application and the libraries it needs to run in the image, and then deploys the application by deploying the container image (Huang et al., 2019). Examples of this are Docker (Merkel, 2014; Boettiger, 2015), Shifter (Gerhardt et al., 2017; Belkin et al., 2018), and Singularity (Kurtzer et al., 2017; Godlove, 2019). In this model, users are no longer directly deploying jobs and configuring environments on compute nodes. Instead, they have the additional task of maintaining the container image.

Furthermore, the container image cannot be deployed across different environments (details in Section 2.3), so users usually need to develop and compile their programs on the login node and then encapsulate them into a container image. It seems to be unnecessary for the user to explicitly build the container image in the HPC system and run the container-related instruction, which only increases the burden on the user.

After the user encapsulates his/her application and its libraries into an image, the system administrator will not be able to view the content or help maintain this particular environment. The maintenance of the image is then left to the user. At the same time, the image will bring some performance overhead to the running of the program (Casalicchio and Perciballi, 2017; Lingayat et al., 2018; Saha et al., 2018). In addition, this model does not solve the privacy problem of users. When different users are developing programs on the same login node, files and processes are still visible to each other.

To solve the problem of privacy, we can use large containers that contain the whole system, such as Linux container (LXC) (Bernstein, 2014; Beserra et al., 2015) and OpenVZ (Che et al., 2010; Kovari and Dukan, 2012), or use virtual machines directly, but the disadvantages are also obvious. On one hand, this approach is too cumbersome. It introduces significant performance overhead that is completely inappropriate for the HPC usage scenario, and it also introduces management overhead. It is completely a black box for system administrators. On the other hand, and more importantly, this model does not ease the user's burden of deploying the execution environment, but rather makes it more cumbersome

(Kwon and Lee, 2020).

In summary, the above problems are obvious in the HPC usage scenario, but there is no complete answer that solves them well. Therefore, to solve these problems, we propose a self-deployed execution environment (SDEE) for HPC that offers the following benefits:

1. Reduce the burden on users. The process of deploying the user's job and execution environment in the compute nodes is completely transparent to the user and completely automatic from the user's perspective, which greatly reduces the burden on the user. At the same time, SDEE is more efficient than manual user deployment and especially suitable for HPC systems. A comparison of the workload is shown in Fig. 1a.

2. Protect user privacy. Based on the currently maturing container-related support in the Linux kernel (namespace, overlay filesystem), SDEE implements isolation between users. The user logs in to the isolated environment on the login node for job development and environment configuration. The development environments of different users are isolated from each other, and the user cannot snoop on other users' files and processes.

3. Support user customization. SDEE supports independent customization of the user's own execution environment. The system administrator maintains and manages the underlying basic environment. The underlying base environment is shared among different users, while the user's customization has a higher priority. The development environment of each user is isolated from other users without affecting any user. For example, if user A upgrades Python 2.7 from the standard system environment to Python 3.7, he/she can do so directly without affecting the users of Python 2.7. Compared with the usage scenarios of traditional HPC systems, SDEE not only ensures that users can share the standard configuration of the system, but also provides users with higher configuration privileges and protects their privacy. At the same time, the user's independent customization will not affect the underlying basic system environment. This ensures the security of the system.

4. Be lighter than containers (Fig. 1b), with less performance and space overhead. SDEE uses only the overlay file system to achieve independent customization of the execution environment for users. SDEE uses just two layers of overlay, whereas Docker uses more layers. The SDEE overhead introduced by overlay is less than that of Docker. SDEE does not need to introduce the traditional container image, and the space overhead is relatively small. At the same time, SDEE is less isolated than traditional container technology. The isolation of SDEE is moderate. Process and file systems are isolated. Devices and the network are shared. This is more suitable for HPC scenarios and introduces less performance overhead while ensuring user requirements.

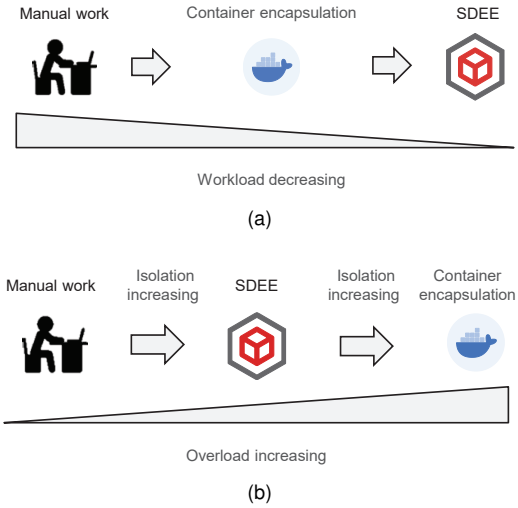

Fig. 1 The comparison of workload (a) and overhead (b) of three approaches

In summary, SDEE reduces the burden on users during the deployment of the execution environment, supports users doing their own customization, protects the privacy of users, and improves system security. At the same time, the performance and space overhead introduced by SDEE is very small. It is designed for HPC systems and brings great convenience to users.

2 Background and related works

Users run their jobs on the HPC system with the help of a job management system. This section introduces the job management system, two job execution modes, and the HPC container technology.

2.1 HPC job management system

All modern HPC systems support multiple users. Each user submits different jobs, some serial,

some parallel, some interactive, some batch, some computation-intensive, some I/O-intensive, which compete for the HPC system resources. Therefore, the HPC system needs management software for real-time response, scheduling, and management of users' jobs to ensure that the resources of the HPC system are fully utilized. This software is the job management system (Georgiou and Hautreux, 2013; Manco et al., 2017).

Currently, simple Linux utility for resource management (SLURM) (Christer, 2012; Azginoglu et al., 2017), portable batch system (PBS) (Feng et al., 2007) and load sharing facility (LSF) are widely used in HPC systems. A good job management system should provide optimized job scheduling policies to improve user job response speed and reduce execution time (Wang K et al., 2014).

Take SLURM as an example. SLURM is a highly scalable and fault-tolerant cluster manager and job scheduling system. It is widely used by HPC systems around the world. SLURM maintains a queue of pending jobs and manages the overall resource utilization of any single job. It manages available compute nodes in a shared or non-shared manner (depending on resource requirements) for users to run their jobs. SLURM allocates resources reasonably to the job queue and monitors jobs until they are completed.

Although the job management system can quickly handle a user's request to run a job, it is still very limited within the HPC system, because the job management system is not capable of configuring the job execution environment. If the user can issue a job run request to the job management system without considering the environment configuration of the compute node, it will greatly reduce the user's additional workload.

2.2 Job execution modes in HPC

In modern HPC systems, a large number of physical nodes are divided into two categories: login and compute nodes. The job management system links these two types of nodes together. After the user logs in to the login node, the development and debugging of the job and the configuration of the job execution environment are performed on the login node. A job run request is then issued through the job management system, which assigns compute nodes to the job to execute it (Fig. 2).

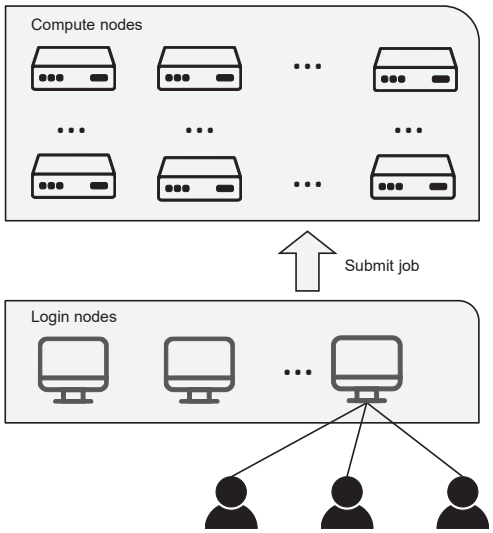

Fig. 2 Traditional job execution in HPC

In this process, however, the user has only configured the execution environment at the login node, such as the libraries required by the job. There is no such environment at the compute nodes, so the job cannot run successfully on the compute nodes. There are usually two ways to solve this issue.

In general, users can manually configure the execution environment of some compute nodes and then specify these nodes to run their own jobs through the job management system. However, when too many compute nodes are used, such a manual configuration wastes too much time. The second approach is to deploy the execution environment of the job to the shared file system of all compute nodes, so that the user needs only to specify the relevant path when running the job through the job management system. However, in the context of multiple HPC system users, this approach can easily cause chaos in the system environment, and changes to the shared file system may have a negative impact on other users. All users have permission to access files in the shared file system. If user A accidentally deletes or modifies another user's file, it can cause a lot of trouble. It is difficult for other users to find the cause of the problem, because they do not know whose operation caused the problem. If user A and user B use the same file in the shared file system, but they require different content, such as a configuration file for a tool, then user A's changes to the file will affect the use by user B. Ultimately, files of different users are visible to other users and lack isolation.

Another way relies on container technology. Container technology has been applied to HPC (Xavier et al., 2013; Gantikow et al., 2015; Herbein et al., 2016; Hale et al., 2017) and has facilitated the migration and deployment of applications. However, containers still expose a lot of problems in the specific HPC scenario. We describe this in detail in Section 2.3.

2.3 HPC container technology

Container technology is applied to achieve rapid deployment of the job execution environment in HPC systems. Docker and Singularity are good examples.

Docker was released in March 2013. It packages applications with all their dependences to facilitate application deployment and achieves a good software ecology through Docker Hub.

Singularity is a container technology developed at the Lawrence Berkeley National Laboratory specifically for large-scale, cross-node HPC and deep learning (DL) workloads. It supports conversion from the Docker image to the Singularity image, so users can package their local job execution environment and upload it to Docker Hub. Then they can use Singularity to run their jobs on HPC systems.

However, in the existing container schemes, there are still some obvious deficiencies in the HPC usage scenario:

1. The HPC system is highly customized. Docker and Singularity are designed for cross-environment deployment. Users develop and package containers in their desktop environment and then deploy them to the HPC system, effectively leveraging the Docker Hub ecosystem. However, this process is often hampered by HPC systems that have highly customized hardware and software.

The desktop environment on which users are building their Docker images may be quite different from the login and compute nodes in HPC systems. For example, Tianhe uses ARM instruction set architecture (ISA) and Taihu-light uses the self-developed 64-bit RISC ISA. However, the majority of desktop and Docker images are X86-based. Moreover, the software stacks in HPC systems are highly customized to fit their special hardware. Thus, users cannot deploy their images across the different environments. The common scenario is that users need to develop and compile their program on the login node, not on their own desktop.

To deploy the job's execution environment, the user can still package the job and its dependencies in an image on the login node, and then deploy the job and its environment by deploying the image, but the process always seems redundant. This is a complete additional burden for the user.

2. The container is completely owned by the user and it relies only on the user to maintain its environment, which makes it more difficult for the user to use. The HPC computing resources are highly customized, and it is complicated for users to maintain the image. Furthermore, the container image constructed by the user is a black box for the system administrator, so the system administrator is unable to help the user maintain and update the underlying common environment.

In summary, the current container technology has obvious deficiencies in HPC systems, and there are still many burdens for users to run their jobs in HPC systems. SDEE aims to solve these problems, is highly compatible with the HPC usage scenario, and offers great convenience to users.

3 Design and implementation

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

3.1 Process isolation

SDEE provides users with a light-weight, isolated execution environment. We use the Linux namespace mechanism (Biederman and Networx, 2006; Wright et al., 2006; Rosen, 2013) to achieve process isolation. When the user logs in to the login node, a new process is started at the login node. The process is isolated to a new PID namespace through the Linux PID namespace.

PID namespaces are used to isolate the process. Therefore, the process IDs in different PID namespaces can be repeated and do not affect each other. As shown in Fig. 3, PID namespaces can be nested, which means that there is a parent-child relationship. All new namespaces created in the current namespace are sub-namespaces of the current namespace, and all process information in the descendant namespace can be seen in the parent namespace. At the same time, process information in the ancestor or brother namespaces cannot be seen in the sub-namespaces. This makes it easy to start multiple

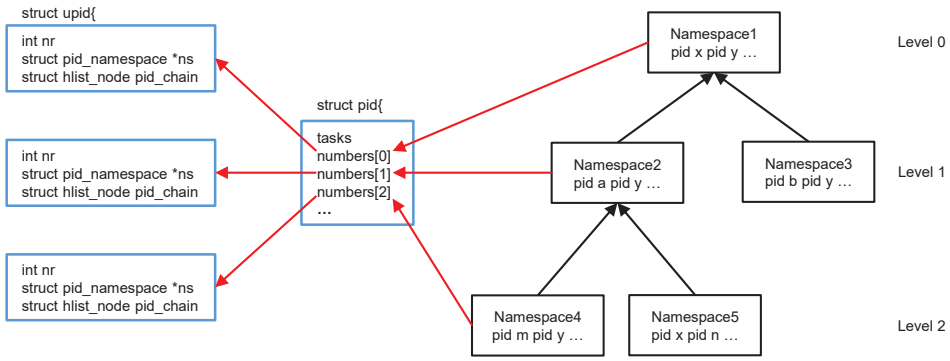

Fig. 3 Linux PID namespace

execution environments on a single physical node and protects user privacy.

Each process under Linux has a corresponding /proc/PID directory, which contains a great deal of information about the current process. For a PID namespace, the /proc directory contains only information about the current namespace and all of its descendants' namespace processes, so we remount the proc file system.

This process, then, becomes the root process of this new PID namespace, with a completed independent process tree as shown in Fig. 4.

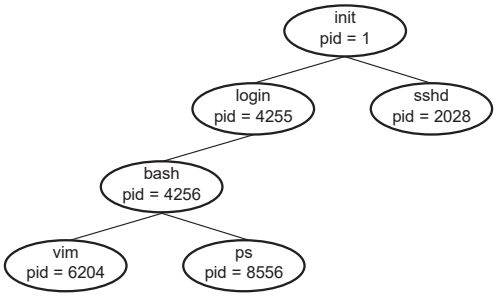

Fig. 4 The process tree of the new PID namespace

In this way, user privacy is protected. Multiple users log in to the same login node and cannot see the specific process of other users. We do the same on compute nodes when a job is deployed on them. This allows the job to run in an isolated environment.

In addition, this approach facilitates the management of user processes. For example, when a user logs out or a job is completed, we need only to kill the root process on the corresponding node, so the entire process tree will disappear. This operation cannot affect processes outside this PID namespace.

3.2 File system

SDEE allows all users to share the standard system environment of the HPC system, while customizing their own execution environment. This is achieved by designing an overlay file system.

SDEE uses two file system layers, the upper and the lower layers. The lower layer is a standard system environment (the system's "/" directory) that is managed and maintained by the system administrator. It contains the basic program libraries and common compilers, editors, and so on, which are shared by all users. An empty folder is then overlaid on the host "/" as the upper layer for the overlay file system. Finally, chroot is used to access the merged folder.

Based on the features of the overlay file system (Fig. 5), when a user reads or writes a file, he/she first looks in the upper layer. If the file does not exist in the upper layer, he/she goes through the lower layer. In other words, the upper layer has a higher priority. It is worth mentioning that the overlay file system uses the copy on write mechanism; that is, when a file is written, if it exists not in the upper layer but in the lower layer, it will be copied to the upper layer and then modified. This mechanism ensures that the lower layer will never be modified by the normal user.

This file system provides a good balance between sharing and customization. For example, if Python 2.7 is installed in the system environment maintained by the system administrator, but the user needs to use Python 3.7, then the user can update the Python version from 2.7 to 3.7 directly without any worry. Because this happens in the upper layer, there is no impact on the standard

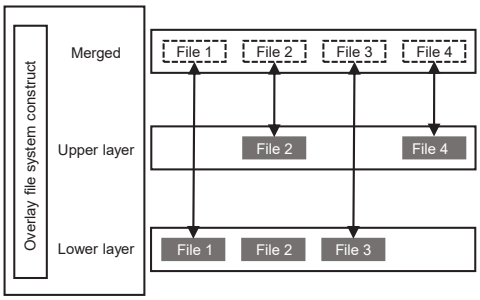

Fig. 5 The features of the overlay file system

system environment. Other users still use Python 2.7 in the system. If the system administrator changes Python 2.7 in the lower layer to Python 3.7, all users will immediately see this modification.

The design of the overlay file system not only ensures that users can share the standard system configuration, but also provides users with greater autonomy and protects user privacy. At the same time, the user's free customization will not affect the underlying system environment, which improves system security. The advantage of such a design is that the system administrator and the user can work together to maintain the environment. The existing container environment, such as Docker, can be maintained only by the user.

3.3 Deployment of the environment

To achieve self-deployment of the execution environment with the job, we need to preconfigure the login node to be ready for the user's login. At the same time, linkage with the job management system should be preset, which will be introduced in Section 3.4.

We need to run a daemon ahead of time on the login node. When a user asks to log in to a login node, he/she will actually log in to an isolated execution environment. The daemon does this. This daemon process continues to listen for login requests from users. When a login request from the user is detected, the daemon process is responsible for starting a new isolated process (introduced in Section 3.1) and overlaying an empty folder (introduced in Section 3.2).

The user then logs in to the environment. This is the environment in which jobs are developed, debugged, and configured.

3.4 Job execution process

Traditionally, a user issues a job run request to the job management system. The job management system assigns appropriate compute nodes to execute the program. To achieve self-development of the execution environment, we design a mechanism to combine self-deployment with the job management system.

As is shown in Fig. 6, there are three users sharing the same login node. We use different colors to distinguish them. When a user submits a job, the job management system assigns appropriate computing nodes.

Then the user's upper file system in the login node is automatically synchronized to the assigned compute nodes and becomes the upper file system of these compute nodes. The job and its special libraries and all other modifications made by the user are contained in the upper file system. Because the compute nodes and the login nodes have the same configuration in the underlying file system, they now have the same upper file system. This step completes the synchronization of the job execution environment.

Synchronization does not always make a complete copy of the upper file system. The user's environment configuration is cached on the compute nodes. If the user resubmits the job, only the modified parts will be synchronized. We determine which files need to be synchronized by checking the size of the files and the timestamp.

The document legend in Fig. 6 stands for the user's job, and the user and his/her job have the same color. The job and its special libraries are contained in the upper file system. We now have a complete execution environment of user's job on the compute nodes and start a new isolated process on the compute node. This is similar to what happens when a user logs in to a login node, and we obtain a completely separate process tree. The job runs in this automatically configured environment on the compute nodes, and the job management system will feed the results back to the user on the login node.

The above process is isolated for each different user. Users do not influence each other. The compute nodes that are assigned to different users may be completely different or overlap, depending on the job management system. In other words, the number

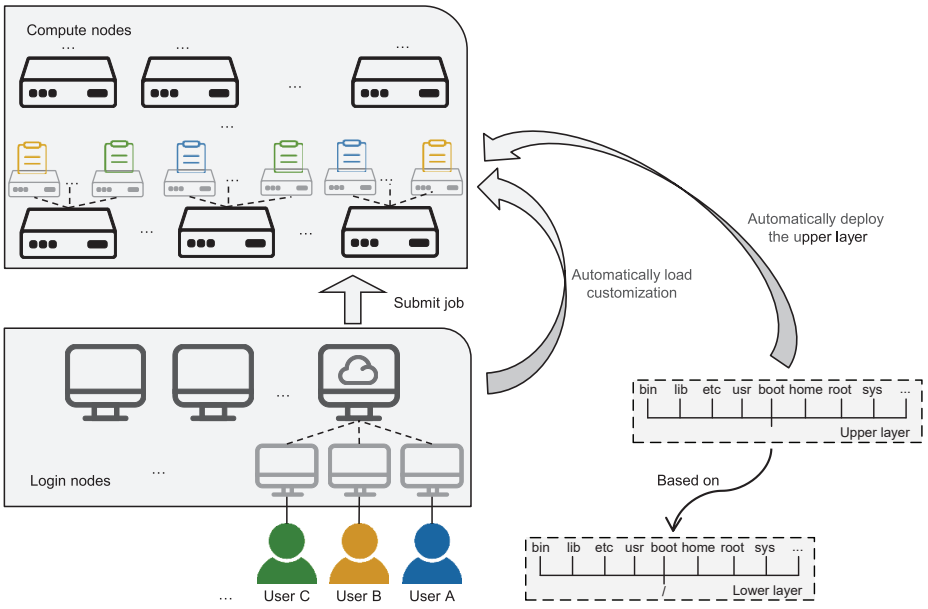

Fig. 6 The self-deployed execution environment (SDEE) framework

of upper file systems on a compute node depends on how many different user jobs are running there. For example, a compute node in Fig. 6 is being used by users A and B, while another compute node is used by users A and C. Of course, it is also possible for a compute node to be used simultaneously by users A, B, and C. It all depends on the job status of all users in the entire system.

Finally, when the results of the job are returned, the execution environment is cleared on the compute nodes. After the job is submitted, we start an isolated process on the compute node as the root process in the process tree. When the root process is killed, the entire process tree will cease to exist. Then the upper file system is cleared out. The overlay file system makes this work much easier.

The whole process is transparent to the user. From the user's point of view, he/she just issues the run command to the job management system to run the job at the login node and then obtains the results of the job after a period of time. The whole process is completed automatically, which greatly reduces the burden on users.

3.5 Security

SDEE supports users in customizing their own environment freely, which relies on the overlay file system. This file system ensures that user customization happens in the upper layer without impacting the underlying system environment, which supports system security.

SDEE lets the user be the root user in his/her own isolation environment. This is implemented through the SUID mechanism, and the user has root privileges, but we have limited the privilege of users based on the Linux capability. We drop privileges that may threaten security. For example, CAP_SYS_ADMIN may leave a door open to break into the kernel. CAP_SYS_BOOT may enable a user to do operations that affect others.

SDEE can effectively defend itself against the vulnerabilities that have been published so far.

4 Experiment and evaluation

SDEE reduces the burden on the user and does not require the user to manually deploy the environment. Because the execution environment is lightweight, we expect that the cost of performance will be very small. In this section we mainly test its overhead compared with bare metal.

The experimental platform is a server of the Tianhe system. It is equipped with FT-1500A 16-core CPU and 64 GB of memory.

First of all, we used Unixbench to test the overall performance of SDEE, which gave us a rough

estimate of SDEE. Then we did some further tests including CPU performance, file operation, and startup time.

Unixbench is not only a CPU, memory, and disk testing tool, but also a more general system-based benchmarking tool. The purpose of this test is to provide a basic performance evaluation of SDEE. The result of these tests is an index number. Finally, there is a comprehensive score that can easily be used for comparison with others. Our test results for bare metal, SDEE, Docker, and Singularity are shown in Tables 1 and 2.

Tables 1 and 2 show the average results of our 10 tests. Dhrystone 2 using register variables focuses on the string handling test. Double-precision whetstone focuses on testing the efficiency and speed of double-precision floating-point arithmetic. File copy focuses on file operation tests including read, write, and copy. Pipe throughput focuses on testing the communication between processes.

We calculated the overheads of SDEE, Docker, and Singularity based on the bare metal score. From the test results, we can see that SDEE had the lowest overhead. In the individual test samples though, SDEE's performance was not the best. The system contingency may lead to this. Overall, SDEE performed best. Docker and Singularity had their advantages and disadvantages, but Singularity was slightly better than Docker as a whole. We also noticed that most of the overheads for the three systems were in file copy.

Unixbench is a more comprehensive testing tool. To enhance the persuasive power, we conducted further targeted tests.

The benchmarks used in the experiment included NPB (Bailey et al., 1995), SysBench (Kopytov, 2012), and fio. We used these benchmarks to test the basic performance overhead, and also compared the Docker and Singularity test results. The tests included mainly CPU performance and file operation. We also did a special test on the startup deployment time.

4.1 CPU performance

We used the NPB and SysBench benchmarks to measure the overhead and reflect the impact of SDEE on CPU performance.

Figs. 7 and 8 show the SDEE overhead compared with bare metal on the NPB benchmark. Each NPB benchmark test case had seven problem sizes. We chose a scale large enough to avoid the contingency of the results. We chose bt, lu, sp, and ua benchmarks, and each benchmark was run 10 times. The average result is shown in the figure. We marked the fluctuating units of the test results.

We can see from the results that the overhead introduced by SDEE was very small, less than 1%. The main source of performance overhead was the overlay file system. We used only two layers, so we introduced only a small amount of overhead. At the same time, the benchmark did not operate on the file frequently, so the overlay file system had less impact.

Table 1 The comparison of Unixbench results (running one parallel copy of tests)

Benchmark	Score				Baseline	Unit	Overhead		
	Bare-metal	SDEE	Docker	Singularity			SDEE	Docker	Singularity
Dhrystone 2 using register variables	54 424 915.52	54 461 290.80	54 230 590.20	54 127 957.47	116 700	lps	−0.07%	0.36%	0.55%
Double-precision whetstone	3663.90	3657.17	3653.23	3647.17	55	MWIPS	0.18%	0.29%	0.46%
File copy 1024 bufsize 2000 maxblocks	1 625 214.80	1 621 733.70	1 601 870.70	1 618 400.37	3960	KBps	0.21%	1.44%	0.42%
File copy 256 bufsize 500 maxblocks	462 600.68	459 027.73	456 043.33	459 694.40	1655	KBps	0.77%	1.42%	0.63%
File copy 4096 bufsize 8000 maxblocks	3 605 343.52	3 591 148.63	3 593 464.83	3 557 815.30	5800	KBps	0.39%	0.33%	1.32%
Pipe throughput	3 064 019.30	3 046 674.73	3 033 976.87	3 053 341.40	12 440	lps	0.57%	0.98%	0.35%

Table 2 The comparison of Unixbench results (running four parallel copies of tests)

Benchmark	Score				Baseline	Unit	Overhead		
	Bare-metal	SDEE	Docker	Singularity			SDEE	Docker	Singularity
Dhrystone 2 using register variables	204 047 022.52	203 873 046.10	203 715 198.60	203 373 046.10	116 700	lps	0.09%	0.16%	0.33%
Double-precision whetstone	14 462.35	14 405.60	14 404.10	14 430.60	55	MWIPS	0.39%	0.40%	0.22%
File copy 1024 bufsize 2000 maxblocks	1 828 673.10	1 809 848.30	1 803 758.97	1 812 348.30	3960	KBps	1.03%	1.36%	0.89%
File copy 256 bufsize 500 maxblocks	491 701.37	487 144.18	484 199.23	484 394.18	1655	KBps	0.93%	1.53%	1.49%
File copy 4096 bufsize 8000 maxblocks	5 218 754.62	5 196 358.93	5 172 533.63	5 186 358.93	5800	KBps	0.43%	0.89%	0.62%
Pipe throughput	11 514 943.03	11 526 694.20	11 378 097.50	11 501 694.20	12 440	lps	−0.10%	1.19%	0.12%

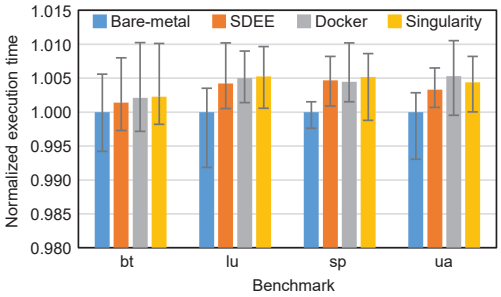

Fig. 7 Overhead of the NPB benchmark (serial)

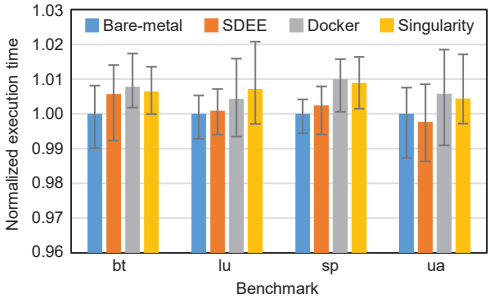

Fig. 8 Overhead of the NPB benchmark (massage passing interface (MPI) on 16 processes)

Compared with Docker and Singularity, SDEE introduced less performance overhead, because Docker and Singularity use more isolation mechanisms than SDEE. Although the overheads of SDEE, Docker, and Singularity were all within a small range, the overheads introduced by Docker and Singularity were more obvious.

The NPB benchmark includes both serial and parallel programs. The results were similar because the concurrency of the program is not the cause of the performance overhead.

To increase the completeness of the test results, we also used the SysBench benchmark. The SysBench benchmark executes prime addition operations, so the higher the selected upper prime limit, the more time the benchmark will naturally take for execution. To avoid the contingency caused by the short running time of the benchmark, we selected a high enough upper limit of prime number. A repeat test was also conducted to obtain an average result (Fig. 9). The results of the SysBench benchmark added to the evidence that SDEE introduces negligible overhead, which is a little better than those of Docker and Singularity.

4.2 File operation

When testing CPU performance in Section 4.1, the overhead was due mainly to the overlay file system (Mizusawa et al., 2017). However, the NPB and SysBench benchmarks do not focus mainly on the file operation, so this subsection is dedicated to test the overhead associated with file operations.

We used the fio benchmark. Fio provides six test modes which include sequential read, sequential write, sequential mixed read and write, random read, random write, and random mixed read and write. We ran each mode 10 times. Fig. 10 shows the average results. From the results, we can see that

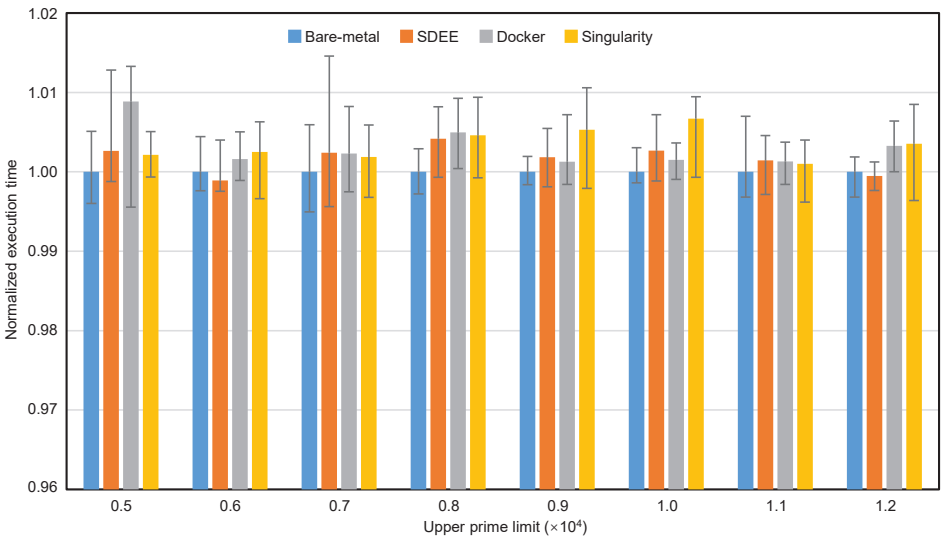

Fig. 9 Overhead of running the SysBench CPU test

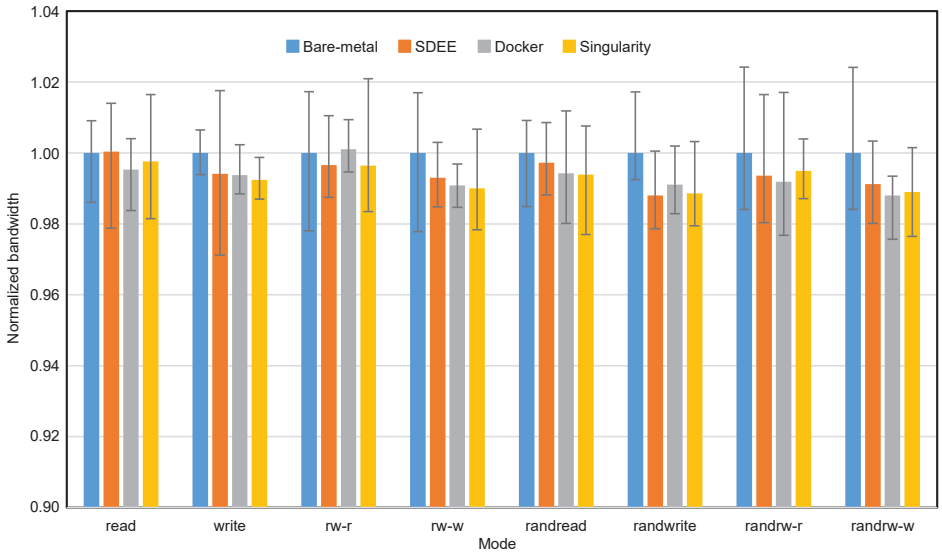

Fig. 10 Fio test overhead

the overhead in write, especially in randwrite mode, was relatively obvious, but only within the range of 2%. It is still acceptable and there are few such jobs in the HPC system.

The Docker and Singularity performance was slightly worse than that of SDEE, especially in terms of write operation overhead. This is because the number of file system layers that Docker and Singularity use is variable, and will increase due to the modification operation, thus creating more overhead.

4.3 Startup and deployment time

SDEE provides users with an isolated, customizable environment. SDEE is light-weight. When the job runs on the compute node, the environment must also be started on the compute node first, so there's a very high limit on this startup time. We compared the startup time of SDEE, Docker, and Singularity. This was done by starting 100 SDEEs in a row, 100 Docker containers in a row, 100 Singularity containers in a row, and recording the total startup time separately. The experimental results are shown in Table 3.

Table 3 Startup time of 100 SDEE, Docker, and Singularity environments

Environment	SDEE	Docker	Singularity
Total time (s)	2.848	16.930	12.489

As we can see from Table 3, SDEE had a very quick startup, which was 4.9 times faster than Docker and 3.4 times faster than Singularity. This startup speed is more suitable for the HPC usage scenario.

SDEE implements self-deployment of the job execution environment and is transparent to the user. In this subsection, we tested the execution environment's deployment time. Because the deployment process is designed to synchronize the upper file system to the corresponding compute nodes, the deployment time should increase with an increase in the size of the upper file system. We set the different upper file system sizes separately to test the time required for self-deployment. This is the time between when the user issues the run command and when the job actually starts running.

As can be seen from Fig. 11, the deployment can be completed in seconds. In most cases, for user jobs users need only to configure their unique libraries and dependencies. When the size was several megabytes, the job can start running at the compute node in less than one second.

The maximum size of the upper file system that we used in the test process was 2 GB, and the test result was 80 s. It looks like a long time, but the efficiency was greatly improved compared with the user's manual deployment. Moreover, such a large size of libraries and dependencies should rarely happen in actual application scenarios.

The deployment time of Docker and Singularity

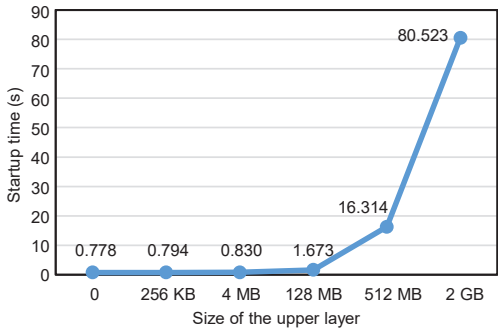

Fig. 11 SDEE deployment time

is hard to measure. This is more of an empirical value, because this process requires manual execution by the user. It is inefficient, and cannot be measured in exact time. In contrast, SDEE's deployment time is a definite advantage.

5 Discussions

SDEE is designed to solve the problems related to the user's job execution environment in the HPC system. Traditionally, users configure this environment manually, or the system administrator preconfigures it according to the user's requirements. This method not only greatly increases the burden on users, but also brings problems of system security and scenarios related to multiple users.

Deployment using containers can encapsulate the user's execution environment. However, because HPC is highly customized, it is likely that the user's container image cannot be used in the HPC system, and users still need to build and maintain the image in the HPC system, so the burden still exists. In addition, the container carries performance overhead.

SDEE solves most of the above problems, realizes the self-deployment of the execution environment, and introduces negligible overhead.

SDEE allows isolation of the job execution environment through the Linux PID namespace mechanism. When multiple users share the same nodes, processes are not visible among users, which protects user privacy. SDEE is a two-layer overlay file system that supports user environment customization and ensures the system administrator's maintenance of the underlying environment. This feature also guarantees system security, and all changes made by the user occur on the upper layer of the file system, with no impact on the real system. This hierarchical file system also facilitates the deployment of a job execution environment.

Finally, SDEE allows self-deployment of the execution environment of user jobs through linkage with the job management system. This process is transparent to users and greatly reduces their burden.

The experimental results show that the environment does not introduce significant overhead, which is better than Docker. In terms of the startup time of self-deployment, SDEE completes the startup in seconds.

SDEE ideas and methods also provide inspiration for future work.

6 Conclusions

In this paper, we have proposed the self-deployed execution environment (SDEE) for HPC jobs. Users run their jobs in an HPC system, which requires a lot of manual work because of the configuration of the environment. SDEE reduces the user's burden in this process. It implements self-deployment of the job's execution environment. At the same time, SDEE supports flexible user environment customization, which not only protects user privacy but also ensures system security. Experiments show that the overhead introduced by SDEE is negligible. The SDEE overhead is lower than both the Docker overhead and Singularity overhead.

SDEE is currently used on NUDT's Tianhe E prototype supercomputer and has good performance.

Contributors

Mingtian SHAO designed the research. Mingtian SHAO, Kai LU, and Wenzhe ZHANG implemented the system. Mingtian SHAO drafted the paper. Kai LU and Wenzhe ZHANG helped organize the paper. Mingtian SHAO revised and finalized the paper.

Acknowledgements

The authors wish to thank Yiqing DAI, Kun ZHANG, and Hao HAN for their help in system debugging. We would also like to thank Zhenwei WU and Yushuqing ZHANG for improving the paper.

Compliance with ethics guidelines

Mingtian SHAO, Kai LU, and Wenzhe ZHANG declare that they have no conflict of interest.

References

Azginoglu N, Atasever MU, Aydin Z, et al., 2017. Open source slurm computer cluster system design and a sample application. Proc Int Conf on Computer Science and Engineering, p.403-406. https://doi.org/10.1109/UBMK.2017.8093424

Bailey DH, Harris T, Saphir W, et al., 1995. The NAS Parallel Benchmarks 2.0. Technical Report.

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, p.1-8. https://doi.org/10.1145/3219104.3219145

Bernstein D, 2014. Containers and cloud: from LXC to Docker to Kubernetes. *IEEE Cloud Comput*, 1(3):81-84. https://doi.org/10.1109/MCC.2014.51

Beserra D, Moreno ED, Endo PT, et al., 2015. Performance analysis of LXC for HPC environments. Proc 9th Int Conf on Complex, Intelligent, and Software Intensive Systems, p.358-363. https://doi.org/10.1109/CISIS.2015.53

Biederman EW, Networx L, 2006. Multiple instances of the global linux namespaces. Proc Linux Symp, p.101-112.

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

Casalicchio E, Perciballi V, 2017. Measuring Docker performance: what a mess!!! Proc 8th ACM/SPEC Int Conf on Performance Engineering Companion, p.11-16. https://doi.org/10.1145/3053600.3053605

Che JH, Shi CC, Yu Y, et al., 2010. A synthetical performance evaluation of OpenVZ, Xen and KVM. Proc IEEE Asia-Pacific Services Computing Conf, p.587-594.

Christer E, 2012. Simple Linux Utility for Resource Management. Platform LSF. Technical Report.

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

Gantikow H, Klingberg S, Reich C, 2015. Container-based virtualization for HPC. Int Conf on Cloud Computing and Services Science, p.543-550.

Georgiou Y, Hautreux M, 2013. Evaluating scalability and efficiency of the resource and job management system on large HPC clusters. Workshop on Job Scheduling Strategies for Parallel Processing, p.134-156.

Gerhardt L, Bhimji W, Canon S, et al., 2017. Shifter: containers for HPC. *J Phys Conf Ser*, 898:082021.

Godlove D, 2019. Singularity: simple, secure containers for compute-driven workloads. Proc Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning), p.1-4. https://doi.org/10.1145/3332186.3332192

Hale JS, Li LZ, Richardson CN, et al., 2017. Containers for portable, productive, and performant scientific computing. *Comput Sci Eng*, 19(6):40-50. https://doi.org/10.1109/MCSE.2017.2421459

Herbein S, Dusia A, Landwehr A, et al., 2016. Resource management for running HPC applications in container clouds. Int Conf on High Performance Computing, p.261-278. https://doi.org/10.1007/978-3-319-41321-1_14

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

Kopytov A, 2012. SysBench Manual. MySQL AB.

Kovari A, Dukan P, 2012. KVM & OpenVZ virtualization based IaaS open source cloud virtualization platforms: OpenNode, Proxmox VE. Proc IEEE 10th Jubilee Int Symp on Intelligent Systems and Informatics, p.335-339. https://doi.org/10.1109/SISY.2012.6339540

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

Kwon S, Lee JH, 2020. DIVDS: Docker image vulnerability diagnostic system. *IEEE Access*, 8:42666-42673. https://doi.org/10.1109/ACCESS.2020.2976874

Lingayat A, Badre RR, Kumar Gupta A, 2018. Performance evaluation for deploying Docker containers on baremetal and virtual machine. Proc 3rd Int Conf on Communication and Electronics Systems, p.1019-1023. https://doi.org/10.1109/CESYS.2018.8723998

Manco F, Lupu C, Schmidt F, et al., 2017. My VM is lighter (and safer) than your container. Proc 26th Symp on Operating Systems Principles, p.218-233. https://doi.org/10.1145/3132747.3132763

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

Mizusawa N, Nakazima K, Yamaguchi S, 2017. Performance evaluation of file operations on OverlayFS. Proc 5th Int Symp on Computing and Networking, p.597-599. https://doi.org/10.1109/CANDAR.2017.62

Rosen R, 2013. Resource Management: Linux Kernel Namespaces and Cgroups. Huafix. Technical Report.

Saha P, Beltre A, Uminski P, et al., 2018. Evaluation of Docker containers for scientific workloads in the cloud. Proc Practice and Experience on Advanced Research Computing, p.1-8. https://doi.org/10.1145/3219104.3229280

Wang B, Chen ZG, Xiao N, 2020. A survey of system scheduling for HPC and big data. Proc 4th Int Conf on High Performance Compilation, Computing and Communications, p.178-183. https://doi.org/10.1145/3407947.3407977

Wang K, Zhou XB, Chen H, et al., 2014. Next generation job management systems for extreme-scale ensemble computing. Proc 23rd Int Symp on High-Performance Parallel and Distributed Computing, p.111-114. https://doi.org/10.1145/2600212.2600703

Wright CP, Dave J, Gupta P, et al., 2006. Versatility and Unix semantics in namespace unification. *ACM Trans Stor*, 2(1):74-105. https://doi.org/10.1145/1138041.1138045

Xavier MG, Neves MV, Rossi FD, et al., 2013. Performance evaluation of container-based virtualization for high performance computing environments. Proc 21st Euromicro Int Conf on Parallel, Distributed, and Network-Based Processing, p.233-240. https://doi.org/10.1109/PDP.2013.41