JKSCI

한국컴퓨터정보학회논문지
Journal of The Korea Society of Computer and Information
Vol. 25 No. 9, pp. 11-20, September 2020
https://doi.org/10.9708/jksci.2020.25.09.011

Comparative Analysis of Container for High Performance Computing

Jaeryun Lee*, Yunchang Chae*, Byungchul Tak**

*Student, School of Computer Science and Engineering, Kyungpook National University, Daegu, Korea
*Student, School of Computer Science and Engineering, Kyungpook National University, Daegu, Korea
**Professor, Dept. of Computer Science and Engineering, Kyungpook National University, Daegu, Korea

[Abstract]

In this paper, we propose the possibility of using containers in the HPC ecosystem and the criteria for selecting a proper PMI library. Although demand for container has been growing rapidly in the HPC ecosystem, Docker container which is the most widely used has a potential security problem and is not suitable for the HPC. Therefore, several HPC containers have appeared to solve this problem and the chance of performance differences also emerged. For this reason, we measured the performance difference between each HPC container and Docker container through NAS Parallel Benchmark experiment and checked the effect of the type of PMI library. As a result, the HPC container and the Docker container showed almost the same performance as native, or in some cases, rather better performance was observed. In the result of comparison between PMI libraries showed that PMIx was not superior to PMI-2 in all conditions.

▸ **Key words**: Container, HPC, PMI, Singularity, Shifter, Charliecloud

[요 약]

이 논문에서는 HPC 환경에서 컨테이너의 사용 가능성을 탐구하고 적절한 PMI 라이브러리를 선택하는 기준을 실험을 통해 제안한다. HPC 환경에서 컨테이너 기술의 필요성이 대두되고 있지만 Docker 컨테이너 경우 잠재적인 보안 문제가 있어 HPC 환경에 적합하지 않다는 문제가 있다. 이러한 문제를 해결하기 위해 여러 HPC 컨테이너들이 등장하였지만, 서로 다른 특징들 따른 성능 차이가 발생할 수 있다. 그리하여 본 논문에서는 각 HPC 컨테이너 및 Docker 컨테이너 사이의 성능 차이를 NAS Parallel Benchmark 실험을 통해 측정하고 PMI 라이브러리의 종류에 따른 성능 변화를 확인해보았다. 그 결과 HPC 컨테이너와 Docker 컨테이너는 거의 네이티브와 유사한 성능을 보이거나 경우에 따라 오히려 나은 성능이 관측되었다. 또한 PMI 라이브러리 간의 비교에서는 PMIx가 PMI-2에 비해 모든 조건에서 뛰어난 것은 아닌 것으로 관측된 것을 확인할 수 있었다.

▸ **주제어**: 컨테이너, HPC, PMI, Singularity, Shifter, Charliecloud

• First Author: Jaeryun Lee, Yunchang Chae, Corresponding Author: Byungchul Tak
*Jaeryun Lee (jrlee@knu.ac.kr), School of Computer Science and Engineering, Kyungpook National University
*Yunchang Chae (cyc@knu.ac.kr), School of Computer Science and Engineering, Kyungpook National University
**Byungchul Tak (bctak@knu.ac.kr), Dept. of Computer Science and Engineering, Kyungpook National University
• Received: 2020. 08. 03, Revised: 2020. 09. 11, Accepted: 2020. 09. 11.
• Equally contributed to this work.

http://www.ksci.re.kr pISSN:1598-849X | eISSN:2383-9945

I. Introduction

컨테이너화 혹은 운영체제 수준의 가상화는 하나의 호스트 머신에서 리소스의 격리하여 호스트 시스템 구성설정과의 충돌 없이 어플리케이션에 특화된 새로운 시스템 환경을 구축할 수 있다는 장점으로 기술의 활용이 널리 확산되어 현재는 컴퓨팅 산업에서 빠질 수 없는 위치를 차지하고 있다. 그 중 컨테이너는 다른 호스트 OS에서 새로운 게스트 OS를 설치하여 새로운 시스템을 구축하는 방법인 가상머신과 다르게 리눅스 네임스페이스를 사용한 격리를 통해 호스트와 다른 독자적인 시스템 구성 설정을 만들어 사용하고 cgroup을 통해 호스트의 리소스를 분리하여 활용할 수 있는 특징으로 가상화로 인해 발생할 수 있는 비용을 줄였다. 또한 컨테이너를 사용하는 개발자는 운영체제, 시스템 라이브러리, 환경변수, 사용자 정의 라이브러리 및 어플리케이션을 컴파일하는 방법을 직접 정의하여 하나의 이미지로 만들어 사용한다. 이러한 이미지는 대게 build manifest로 제작되고 이미지 저장소에 push가 가능하므로, Git과 같은 버전 통합 관리 시스템으로 이미지 버전 관리 및 공동 개발이 용이하며 이미지 저장소에서 이미지를 pull하여 쉽게 공유 및 배포할 수 있다. 이러한 장점들로 컨테이너 기술은 분산 시스템 분야에서 소프트웨어 스택을 관리하는 새로운 패러다임을 이끌어냈다.

Docker는 상용 컨테이너 중 가장 흔히 사용되는 컨테이너 솔루션이다. 자체 개발한 LXC를 사용하며, 네임스페이스 및 cgroup 격리를 제공하는 자체 libcontainer 패키지도 사용하고 있다. 하지만 선행 연구들을 통해서 빅데이터, 과학 연산과 같이 고성능 컴퓨팅이 이루어지는 슈퍼컴퓨터와 같은 환경에서는 Docker가 적합하지 않은 것으로 보고되었다 [1]. 보고된 문제점들은 HPC 환경에서 컨테이너 가상화 방식이 도입되기 위해서 검토해야할 여러 사항들이 존재한다는 것을 보여준다.

상용 컨테이너의 활용이 증가함에 따라 HPC 환경에서 컨테이너 도입의 필요성이 증가하면서 최근 상용 컨테이너의 문제점들을 해결하여 HPC 환경에서 사용할 수 있는 컨테이너 솔루션들이 등장했다. HPC 컨테이너들 중 가장 많이 거론되는 Singularity[2], Charliecloud[3], Shifter[4]는 서로 다른 방법론으로 Docker와 같은 기존 컨테이너 기술들의 보안 문제를 해결하는 것에 중점을 두고 개발되었으며 각 컨테이너 솔루션들의 제공자들은 향상된 보안에 대해 여러 문서를 통해 증명해왔다.

HPC 컨테이너들은 각자만의 설계 목표 및 특징, MPI 지원 등으로 인해 발생하는 성능차이가 존재할 수 있다. HPC 분야에서는 연산의 규모가 극한으로 크기 때문에 사용되는 기술들 사이의 미세한 성능 차이가 결국 큰 차이를 보일 수 있다. 그러므로 특히 변화에 민감한 HPC 환경에서 사용되는 기술들은 보안성뿐만 아니라 성능 측면에서도 여러 조사를 통해 기술에 대한 심도 있는 이해가 필요하다. 따라서 본 연구에서는 NPB(NAS Parallel Benchmark)에서 지원하는 MPI(Message Passing Interface) 벤치마크를 사용하여 HPC 컨테이너(Singularity, Charliecloud, Shifter)와 Docker 컨테이너 그리고 네이티브(Native)에서의 수행시간, 메모리 등을 측정하여 분석하였다. 추가로 Slurm workload manager와 같은 RM(Resource Manager)에서 MPI 어플리케이션을 수행하는 프로세스들을 관리하기 위한 통신 인터페이스로 사용되는 PMI(Process Management Interface) 라이브러리 종류에 따라 성능 차이가 발생할 수 있다고 판단하여 PMI-2[5]와 PMIx[6] 두 종류의 라이브러리를 모두 측정하여 이를 성능 측정 변수로 지정하였다.

실험결과 다음과 같은 현상들이 발견되었다. 첫째, HPC 컨테이너는 네이티브와 유사하거나 수행하는 어플리케이션에 따라 오히려 더 빠른 성능을 보이기도 한다. 둘째, 호스트 네트워크 사용 조건의 Docker 컨테이너와 HPC 컨테이너들은 대부분의 벤치마크에 대해 비슷한 성능을 보였다. 셋째, PMIx와 PMI-2의 성능 비교 시 보다 적은 프로세스 수의 조건에서는 PMI-2의 성능이 높으며, 많은 프로세스 수의 조건에서는 PMIx의 성능이 높은 것으로 나타났다. 넷째, 적은 프로세스 수의 조건에서 PMIx의 메모리 사용량이 PMI-2 보다 오히려 더 높은 수치가 나타났다.

본 논문은 2장에서 각 컨테이너와 PMI 등 배경 분석과 관련 선행 연구들을 소개한다. 3장에서는 진행된 성능 비교 실험을 설명하며 4장에서는 실험결과에 대한 분석을 한다. 마지막으로 5장에서는 본 논문의 결론을 정리한다.

II. Preliminaries

1. Background

1.1 Docker

Docker는 가장 널리 사용되는 컨테이너 런타임이지만 몇몇 이유로 인해 HPC 환경에서는 권장되지 않는다. 가장 큰 이유는 보안에 취약하다는 점이다. Docker는 기본적으로 루트 권한이 필요한 프로세스임에 따라 발생하는 여러 보안 문제가 존재할 수 있다. 대부분의 이슈들은 보안 패치를 통해 해결된 상태지만 루트 권한의 필요성은 잠재적

인 위험성이 존재하기 때문에 높은 보안성이 요구되는 HPC 환경에 적합하지 않다. 둘째로 Docker가 기존 HPC 환경에 통합이 쉽지 않다는 것이다. HPC 환경은 대부분 RM을 사용하여 클러스터의 리소스를 사용할 job에 할당하는 batch 시스템을 사용한다. 하지만 Docker의 경우 이러한 batch 시스템의 도입에 대한 고려 없이 설계되어 대부분의 RM들은 현재 Docker를 지원하지 않고 있다. 세 번째 이유는 분산 스토리지를 통합하는 기능의 부족한 것과 TCP/IP와 같은 상업용 네트워크의 사용한다는 점이다. 이로 인해 Docker를 사용 시 슈퍼컴퓨터의 하드웨어 및 소프트웨어 리소스를 온전히 활용하지 못하고 제한적으로 활용할 수밖에 없게 된다. 마지막으로는 가상화로 인한 오버헤드가 크다는 점이다. Docker에서는 HPC 환경에서는 불필요한 네임스페이스들이 과도하게 격리하여(Isolation) 어플리케이션에 필요한 리소스를 사용할 때 오버헤드가 발생하게 된다. 이러한 문제들을 해결하기 위해서 여러 HPC 컨테이너가 개발되게 되었다

1.2 Singularity

Singularity는 Docker 컨테이너를 관리하기 위한 특별한 권한을 필요로 하는 Docker와 달리 HPC 센터의 보안성을 충족시키기 위해 특정한 권한 없이도 컨테이너 사용을 할 수 있도록 개발되었다. Singularity는 기존 전통적으로 HPC 환경에서 사용하는 기술과의 호환을 위해 MPI, InfiniBand, Lustre 같이 HPC 환경에 필수적으로 사용되는 기술들을 지원하며, RM과의 호환성을 위해 자체적으로 Slrum 플러그인을 지원한다. Singularity는 일반적인 PC에 비해 비교적 낮은 버전의 OS가 사용되는 HPC 환경에서도 동작할 수 있도록 예전 환경에서의 호환성을 지원하며 리눅스 2.2버전까지 테스트 되었다. 또한 Singularity의 컨테이너 이미지는 타 컨테이너 환경과는 달리 단일 일반 파일로써 저장되어 이동 및 복사, 스냅샷 등 여러 작업에 용이하고 일반적인 파일로 취급하기 때문에 리눅스 표준 파일 엑세스 제어가 가능하여 접근성에 장점을 지닌다.

1.3 Charliecloud

Charliecloud는 HPC 환경에서 사용자 정의 소프트웨어 스택(User Defined Software Stack)을 지원하기 위해 리눅스 유저 네임스페이스와 마운트 네임스페이스를 사용하는 컨테이너 솔루션이다. Charliecloud는 Docker를 기반으로 하는 산업 표준 컨테이너를 추출하여 디렉터리로 변환한 뒤 디렉터리 내부에 위치한 바이너리 파일을 실행하는 방식으로 동작하기 때문에 쉽게 사용할 수 있다. 그리고 사용자 네임스페이스를 구분하여 사용자는 HPC 센터의 특별한 권한을 가질 필요 없이 컨테이너 내부에서 모든 작업을 수행할 수 있다는 것을 강조한다. 또한 500줄 가량의 짧은 코드로 구현된 점을 특히 강조하는데 필수적인 기능만 제공하여 어플리케이션의 복잡도를 줄여 더욱 안전하며 변화에 민감한 HPC 환경에서 시스템에 최소한의 변경으로 사용자 정의 소프트웨어 스택을 도입할 수 있다는 장점이 있다. 하지만 현재까지 Red Hat과 같은 다양한 리눅스 배포판을 지원하고 있지 않기 때문에 장기적인 안정성과 이식성은 보장되지 않는다는 단점이 존재한다.

1.4 Shifter

Shifter는 HPC 환경의 사용자에게 큰 규모의 작업을 효율적이면서도 안전하게 컨테이너 기술을 사용할 수 있도록 하는 것을 목적으로 하여 NERSC(National Energy Research Scientific Computing Center)에서 개발한 오픈소스 컨테이너 런타임이다. HPC 컨테이너의 목적으로 개발된 만큼 보안성 및 기존 HPC 환경의 RM과 호환성, MPI 지원을 고려하여 설계되었다.

Shifter는 주로 HPC 환경에서 성능 향상을 이루기 위한 몇 가지 특징들이 존재한다. 첫 번째 특징은 병렬 분산 파일 시스템을 위한 이미지 포맷이다. Shifter는 Docker 이미지를 마운트 가능한 공통 포맷으로 변환하며 이를 loop 마운트로 마운트 한다. 이를 통해 병렬 분산 파일 시스템의 중앙 메타데이터 서버에 의존하지 않고 노드에서 파일 조회와 같은 메타데이터 작업을 처리할 수 있다. 따라서 Shifter 이미지가 병렬 분산 파일 시스템에 저장되어 있음에도 단일 환경에서 Docker 인스턴스가 동작하는 것과 비슷한 성능을 보인다. 두 번째 특징은 I/O 최적화를 위한 임시 XFS 파일 시스템의 사용이다. HPC 환경에서는 병렬 분산 파일 시스템이 필수적으로 사용되는데 이러한 환경에서 I/O 성능을 향상시킬 목적으로 Shifter는 각 노드를 위한 임시 XFS 파일을 생성할 수 있다. 이 파일은 병렬 분산 파일 시스템에 생성되지만, 모든 메타데이터 작업이 단일 노드에서 이루어지기 때문에 IO 속도는 매우 빠르다. 따라서 사용자는 해당 XFS 파일을 컨테이너 내부로 마운트하여 자주 읽고 쓰는 파일 작업에 사용되는 디렉터리로 사용함으로써 I/O 성능 향상을 이룰 수 있다. 세 번째 특징은 공유 라이브러리 번들링이다. HPC 환경에서는 일반적으로 공유 라이브러리를 사용 시 클러스터의 컴퓨트 노드가 분산 파일 시스템에 접근하여 공유 라이브러리를 로드할 때 발생하는 추가적인 I/O 작업으로 인해 오버헤드가 생기게 된다. 하지만 Shifter는 컨테이너의 로드 타임에 모든 공유 라이브러리를 컨테이너 내부에 포함시키므로 컨테이너가 작업을 수행하는 동안 외부의 분산 파

일 시스템에 위치한 공유 라이브러리를 사용하는 대신 컨테이너 내부에서 추가 I/O 작업으로 인한 오버헤드 없이 공유 라이브러리를 사용할 수 있다. 선행 연구[4]에서는 이로 인해 Lustre 파일 시스템과 비교하여 Shifter 컨테이너를 사용 시 성능이 약 4배 향상된 것으로 알려졌다.

1.5 PMI

일반적으로 HPC 환경에서는 다수의 노드에 수많은 프로세스들을 생성하여 MPI를 통해 병렬적으로 계산을 수행하는 방식이 주로 사용된다. MPI 병렬 작업을 수행하기 위해 각 노드에 필요한 프로세스들을 생성과정이 필요하다. 이러한 프로세스의 생성 작업 및 종료를 담당하는 프로세스로 PM(process manager)가 존재하며, 이는 각 프로세스 간의 서로 통신을 위해 서로 간의 주소 및 MPI 정보 등 필수적인 정보를 교환하는 설정 과정 또한 수행하게 된다. 이 과정에서 사용환경에 따라 다른 PM과 MPI 라이브러리가 사용될 수 있는데, 이러한 작업들을 PMI 표준을 통해 다른 환경 간에도 정상적으로 구동할 수 있도록 하였다.

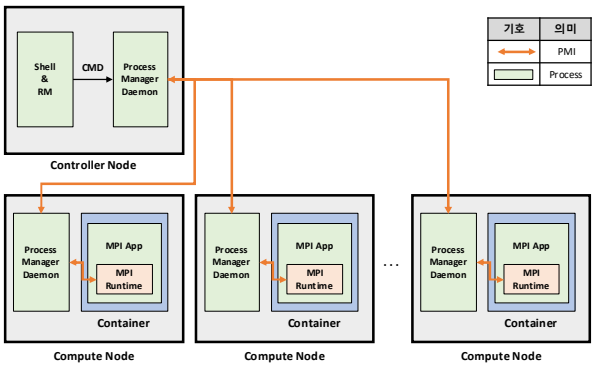

Fig. 1. Diagram of MPI applications workflow in container

Fig. 1은 컨테이너 환경을 사용한 MPI 작업 동작을 보여준다. 사용자는 Shell이나 RM을 통하여 원하는 MPI Job을 생성할 수 있다. 이후 해당 요청은 해당 호스트 내 PM daemon에 전달되며 PMI를 통해 대상이 되는 각 노드의 PM daemon에 요청이 전달된다. 요청을 받게 되는 노드의 PM는 컨테이너를 로드하게 되며 컨테이너 내부 MPI App의 MPI runtime은 PM daemon과 PMI를 통해 통신하며, MPI 통신 채널 설정 등의 작업을 처리한다.

현대에 들어서는 HPC 환경에서 수행하는 작업의 규모가 커져 PMI의 초기 개발 당시에 목표로 한 운영 프로세스의 수보다 큰 규모의 컴퓨팅 환경을 필요로 하게 되었다. 이에 따라 PMI는 초기 개발된 PMI-1에서 보다 뛰어난 편의성 및 성능, Scalability를 높인 PMI-2[5]가 개발되었고, 최근에는 Exascale의 프로세스 수를 관리하기 위해 MPI 라이브러리 함수들의 반응속도를 높인 PMIx(PMI Exascale)[6][7]가 개발되었다.

1.6 NAS Parallel Benchmark

NAS Parallel Benchmark[8]는 병렬 컴퓨팅 성능을 평가하기 위한 프로그램 세트이다. NPB 벤치마크 모음에는 MPI, OpenMP 등의 성능을 테스트할 수 있는 여러 종류의 벤치마크 알고리즘이 포함되어 있다. 그리고 그 둘을 동시에 테스트하는 Multi-Zone 벤치마크도 있다. 그 중 MPI 벤치마크 알고리즘에는 커널의 성능을 테스트하는 IS(Integer Sort), EP(Embarrassingly Parallel), CG(Conjugate Gradient), MG(Multi-Grid), FT(Fourier Transform) 5개의 연산들과 BT(Block Tri-diagonal), SP(Scalar Penta-diagonal), LU(Lower-Upper Gauss-Seidel) 3개의 문제 해결 알고리즘이 있다.

2. Related Work

컨테이너 기술이 보편적으로 널리 사용됨에 따라 HPC 환경에서도 컨테이너를 사용하려는 움직임들이 나타났다. 2007년 Stephen Soltesz et al.[9]은 HPC 환경에서 하이퍼바이저 기반의 가상화 기술에 비하여 컨테이너 기반의 가상화 기술이 적합함을 제시하였으며 이후 두 가상화 기술에 대한 성능 비교 연구 또한 다수 진행되었다.[10][11]

2017년 Maximilien de Bayser et al.[12]의 연구는 HPC 클러스터에 가장 보편적으로 사용되고 있는 Docker 컨테이너를 적용함에 있어 문제점으로 꼽히는 MPI framework와 Docker 컨테이너의 통합에 대한 문제에 대한 원인과 해결 방안을 제시하였다. 또한 동년에 Abdulrahman Azab et al.[13]은 Slurm과 같은 HPC 큐잉 시스템(≒RM)에서 Docker 컨테이너의 동작을 안전하게 실행할 수 있도록 개발된 secure wrapper인 Socker를 소개하였다. Socker는 HPC 환경에서 일반적으로 사용되는 Slurm과 같은 HPC 큐잉 시스템으로 Docker 컨테이너를 사용할 수 있도록 사용 사례를 통합 하였으며, 작업을 수행하는 컨테이너에서 내부의 루트 계정으로 작업을 하는 대신 작업을 제출한 유저의 계정으로 실행할 수 있도록 수정되었다. 그리고 HPC 큐잉 시스템에 의한 Docker 컨테이너의 리소스 사용제한등 여러 기능들을 지원하여 Docker 컨테이너가 HPC 환경에 적합하게 동작할 수 있도록 하였다. 하지만 2018년 Jonathan Sparks et al.[14]는 Docker 컨테이너가 HPC 환경에 완벽하게 적용되기에는 MPI 지원, 로컬 디스크에 대한 의존성, 호스트

리소스에 대한 접근, 사용자 인증 등 추가적인 문제들이 남아있는 것으로 판단하고 이러한 문제를 해결하기 위한 디자인된 Docker API 플러그인 세트를 소개하였다.

이처럼 Docker 컨테이너를 활용하는 연구 외에도 Docker 컨테이너가 가지는 보안적인 위험성 때문에 서로 다른 방법으로 보안 문제를 해결하기 위해 여러 HPC 컨테이너들이 개발되었다 [2][3][4]. 이에 따라 HPC 컨테이너들의 등장으로 인해 각 HPC 컨테이너 간의 성능 비교를 다루는 연구들 또한 등장하였다.

2017년에 Andrew J. Younge et al.[15]은 HPC 환경에서 컨테이너 도입의 가능성을 검토하기 위해서 Cray XC 시리즈 슈퍼컴퓨터에 로드된 컨테이너와 상용 클라우드 서비스 환경인 Amazon EC2에서 로드된 컨테이너의 성능 차이를 비교하는 연구를 진행하였다. 그들은 컨테이너 도입의 기준에 대해 컨테이너에서 수행되는 벤치마크의 수행시간을 주 평가요소로 삼고 네트워크 성능을 평가하기 위한 IMB(Intel MPI Benchmark) 벤치마크와 HPC 시스템에서 MPI 어플리케이션이 수행되는 성능을 파악하기 위한 HPCG 벤치마크를 컨테이너 안에서 실행하여 그 결과를 분석했다. 하지만 특정 라이브러리 함수의 성능을 측정하는 HPCG와 같은 벤치마크보다 더 다각화된 특징을 지닌 어플리케이션을 대상으로 하는 실험이 이루어질 필요가 있다고 판단하여, 본 실험에서는 NPB를 사용하였으며, MPI 라이브러리의 비교 뿐 아니라 MPI 어플리케이션에서 준필수적 요소인 PMI 라이브러리의 종류별 성능 차이에 대한 연구도 필요할 것으로 판단되어 PMI-2와 PMIx를 사용하여 실험을 수행하고 결과를 비교해보았다.

2019년 Oleksandr Rudyy et al.[16]는 기존 선행연구들에서는 HPC 환경에서 실제 상업용으로 배포되고 있는 코드를 컨테이너 내에서 실행한 성능 비교 연구가 부족하다고 판단하여, 생물학 분야에서 사용되고 있는 시뮬레이션 시스템을 Docker, Singularity, Shifter의 컨테이너 환경에서 수행하여 성능 비교를 진행하였다. 해당 연구에서는 Singularity가 가장 최선의 솔루션으로 선택되었고, 이에 따라 Singularity의 확장성(scalability)에 대해 보다 추가적으로 분석이 진행되었다. 하지만 해당 논문에서 Docker 컨테이너의 경우 별도의 오버레이 네트워크를 구성하여 실험이 진행되었기 때문에 타 컨테이너 환경 대비 낮은 성능을 보였다. 따라서 본 논문의 연구에서는 오버레이 네트워크의 사용으로 발생하는 오버헤드를 감소시키기 위하여 Docker 컨테이너에서 호스트의 네트워크를 직접 사용하도록 옵션을 적용하여 비교 대상 HPC 컨테이너들과 유사한 네트워크 환경으로 설정하여 실험을 진행하였다.

위에 소개된 HPC 환경에서의 성능 실험 외에도 컨테이너 자체 성능 비교를 위해 클러스터 단위가 아닌 단일 노드에서의 기본적인 성능들을 평가한 연구도 있었는데 2019년 Alfred Torrez et al.[17]은 Singularity, Charliecloud, Shifter 세 가지의 HPC 컨테이너 간의 성능 비교 실험을 진행하였다. 평가는 단일노드에 로드된 컨테이너에서 CPU 성능, 메모리 성능, 어플리케이션 성능, 컨테이너화로 인한 메모리 추가 사용량을 측정하고 HPC 컨테이너 기술별 차이점에 대해서 분석한 결과를 도출하였다. 해당 연구에서는 HPC 컨테이너 기술들 간에 성능 비교를 수행하였지만 가장 보편적으로 사용되고 있는 Docker 컨테이너와의 성능 비교는 이루어지지 않았다. 본 연구에서는 이러한 점들을 참고하여 평가에 반영하였다.

III. Experiment

1. Hardware Specification

실험을 위해 9개의 PC에 1대의 컨트롤러 노드와 8대의 컴퓨트 노드로 역할을 할당하여 실험을 진행하였으며 컨트롤러 노드는 컴퓨트 노드에게 작업 시작 명령 전달 및 결과 기록 작업만 하고 모든 벤치마크 수행은 컴퓨트 노드에서만 진행된다. 각 PC는 CPU로 intel i7-4770 3.50GHz를 사용하고 8GB의 메모리 용량을 가지며 스위치를 통해 1000Mbps의 하나의 네트워크로 연결되어 있다.

2. Software Specification

본 실험은 리눅스 커널 3.10.0-1062.18.1을 사용하는 CentOS 7.7.1908 운영체제를 사용하는 9대의 PC를 19.05 버전의 Slurm 워크로드 매니저를 사용하여 하나의 클러스터로 만들어 진행되었다. 사용된 컨테이너는 4종으로 각각 19.03.12 버전의 Docker, 3.5.2 버전의 Singularity, 0.14 버전의 Charliecloud, 18.03.0-1 버전의 Shifter로써 각 컨테이너 기술의 최신 버전을 사용하였다. MPI 어플리케이션을 컴파일하고 실행하는데 사용한 MPI 라이브러리는 4.02 버전의 OpenMPI를 사용하였다. 그리고 MPI 작업을 하는 프로세스를 관리하기 위한 PMI 라이브러리를 사용하였는데, PMI 라이브러리의 종류별 성능 비교를 위해 Slurm 패키지에 포함되어 있는 Slurm PMI-2와 3.1 버전의 PMIx를 사용하였다. 마지막으로 3.4 버전의 NAS parallel benchmark에 포함된 NPB-MPI 벤치마크 어플리케이션을 사용하여 성능 측정에 사용하였다.

3. Method

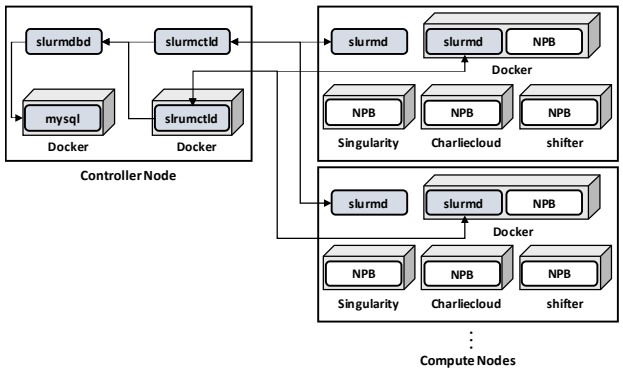

Fig. 2. Diagram of the experiment system and its Slurm service association

Fig. 2는 본 논문의 실험에 사용된 실험환경 구성도이다. 네이티브 환경, Singularity, Charliecloud, Shifter 컨테이너 환경에서 NPB를 동작시키는 경우 컨트롤러 노드의 slurmctld를 통해 각 컴퓨트 노드의 slurmd로 NPB 시작 명령이 전달된다. 그 후 slurmd는 각 호스트 혹은 각 호스트에 있는 컨테이너 내부에 벤치마크 실행 명령을 전달한다. 반면 Docker의 경우 Slurm 워크로드 매니저로부터 공식적인 지원을 받지 못하여 호스트에서 구축된 Slurm 클러스터가 Docker 컨테이너에 직접적으로 실행 명령을 내리지 못한다. 이를 해결하기 위해 호스트의 Slurm 프로세스를 사용하는 대신 Docker 컨테이너에 slurmctld와 slurmd를 설치하여 하나의 클러스터로 만들어 사용하였다. 그 결과 다른 컨테이너에서 벤치마크가 실행되는 방법과 동일하게 컨트롤러 노드에 위치한 Docker 컨테이너 내부의 slurmctld을 통해 컴퓨트 노드의 Docker 컨테이너 내부에 있는 slurmd로 벤치마크를 실행 명령을 내려 Docker 컨테이너 내부에서 벤치마크를 실행할 수 있게 되었다. 또한 Docker의 경우 타 컨테이너들과 마찬가지로 네트워크 오버헤드를 줄이기 위해 별도의 Overlay 네트워크 구성을 하지 않고 Host 네트워크를 사용하도록 설정하여 실험을 진행하였다,

성능 측정을 위한 NPB 벤치마크는 모든 MPI 벤치마크 종류별로 B 클래스, C 클래스, D 클래스의 워크로드를 갖도록 설정하고 각 컨테이너에서 직접 컴파일하였다. srun 명령어를 통해 Slurm 클러스터에 MPI 벤치마크 작업을 분배할 때 벤치마크를 병렬적으로 수행하는 프로세스들은 srun의 ntask 옵션의 값만큼 생성이 되며, 모든 컴퓨트 노드에 1개씩 순차적으로 할당하도록 설정하였다.

4. Tracking System Resource Usage

NPB 실험의 리소스 사용량은 Slurm 작업에서 사용된 리소스를 수집한 데이터를 SlurmDB 서비스로 mysql에 저장한 뒤 sacct 명령어를 통해 출력하여 사용했다. 실험 결과를 저장한 파일에는 수행한 실험 대상과 조건을 포함하여 벤치마크를 수행하는데 걸린 시간, CPU 사용량, 메모리 사용량, 네트워크 사용량, 디스크 사용량, 전력 사용량을 기록하였다. 본 논문에서는 주로 벤치마크의 수행 시간 그리고 메모리 사용량과 관련된 메모리를 관찰하여 특징을 분석했지만, 추후 다른 메트릭도 분석할 계획이다.

IV. Result

1. Analysis Performance of Containers

1.1 Benchmark Execution Time

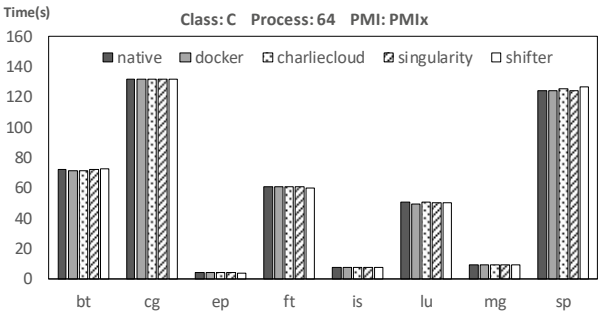

Fig. 3. Comparison chart of NPB execution times on different container within type of benchmark

Table 1. Table of container NPB execution Time

Bench	Min	Max	NT/Min	DK/Min	CC/Min	SG/Min	SF/Min
BT	DK	SF	+0.7%	-	+0.3%	+0.8%	+1.4%
CG	CC	DK	+0.1%	+0.2%	-	+0.1%	+0.1%
EP	SF	SG	**+12.2%**	**+12.0%**	**+17.4%**	**+17.6%**	-
FT	SF	NT	+1.5%	+1.0%	+0.7%	+0.8%	-
IS	SG	DK	+1.6%	+1.9%	+1.2%	-	+0.4%
LU	DK	NT	+2.6%	-	+2.1%	+1.8%	+1.4%
MG	SF	SG	+0.4%	+0.3%	+0.1%	+1.3%	-
SP	SG	SF	+0.1%	+0.1%	+1.0%	-	+2.2%

* NT(native), DK(docker), CC(charliecloud), SG(singularity), SF(shifter)

Fig. 3 은 8개의 노드에 총 16개의 프로세스를 각 노드당 2개씩의 할당하여 수행한 전체 벤치마크 셋의 결과를 나타낸다. 실험결과를 통해 알 수 있는 사실들은 다음과 같다. 첫째, 컨테이너에서의 어플리케이션 수행은 EP의 경우 네이티브 보다 최대 17.4% 빠른 수행시간이 나타났다. 각 컨테이너 별 성능적 특징에 비롯한 결과로 해석될 수

있는데, 특히 Shifter의 경우 컨테이너 공유 라이브러리를 모두 컨테이너 내부에 번들링하여 기존 대비 파일 엑세스에 대한 오버헤드를 줄이는 공유 라이브러리 번들링 특징 때문에 네이티브보다 빠른 수행시간이 나타난 것으로 판단된다. 둘째, Docker 컨테이너는 대부분의 경우에서 HPC 컨테이너들이나 네이티브와 유사한 성능을 보였다. 실험결과에서 Docker 컨테이너는 매우 적은 수행시간으로 기록된 EP를 제외한 나머지 대부분의 경우에서 타 컨테이너들과 2.58% 정도의 성능 차이를 보여 전체적으로 비슷한 성능이 기록되었다. 본 실험에서 사용된 Docker 컨테이너는 호스트 네트워크를 사용하도록 환경을 구성한 결과이며, Overlay 네트워크를 사용하는 등 네트워크 오버헤드가 있는 조건에서는 성능 차이가 발생할 수 있다.

1.2 Container Load Time

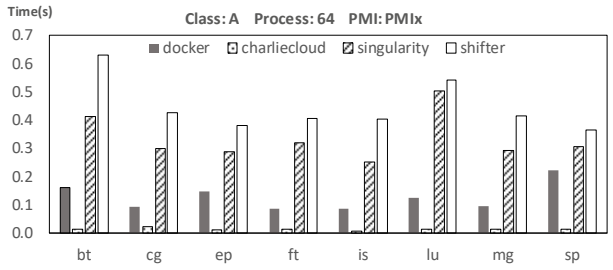

Fig. 4. Comparison chart of container load times on different container within type of benchmark

Table 2. Table of container load times

Bench	Min	Max	MinVal	DK/Min	CC/Min	SG/Min	SF/Min
BT	CC	SF	0.013 s	12.58	-	31.94	**49.06**
CG	CC	SF	0.024 s	3.84	-	12.63	**17.98**
EP	CC	SF	0.011 s	13.34	-	25.90	**34.30**
FT	CC	SF	0.012 s	7.09	-	25.99	**33.19**
IS	CC	SF	0.007 s	12.43	-	36.00	**57.29**
LU	CC	SF	0.012 s	10.18	-	41.03	**44.05**
MG	CC	SF	0.014 s	7.15	-	21.74	**30.78**
SP	CC	SF	0.013 s	16.48	-	22.97	**27.17**

* NT(native), DK(docker), CC(charliecloud), SG(singularity), SF(shifter)

Fig. 4는 컨테이너 별 초기화 시간을 측정하기 위하여 진행한 실험의 결과로써, “1.1 Benchmark Execution Time”의 실험에서 벤치마크에서 발생하는 수행시간의 비중을 보다 줄이기 위해 워크로드를 A로 변경하고 나머지 조건은 동일한 실험을 진행하였다. 또한 MPI 초기화 작업 등의 컨테이너 초기화 시간과 관련이 없는 요소들을 제외하기 위하여 각 경우의 각 컨테이너의 수행시간에서 네이티브 수행시간 값을 제외한 뒤 평가하였다.

Table. 2를 통해 Charliecloud의 경우 모든 벤치마크에 대해 컨테이너 로드타임이 대체로 0.01초 전후의 시간이 소요된 것을 확인할 수 있으며 다른 컨테이너들과 비교하여 가장 짧은 컨테이너 로드 타임을 보였다. 컨테이너 로드 타임이 가장 길었던 컨테이너인 Shifter는 최소값을 보인 Charliecloud와 비교하여 최대 49.06배 더 긴 시간을 컨테이너 로드에 사용했다. 그 이후로 Singularity, Docker 순으로 컨테이너 로드 타임이 짧은 것을 볼 수 있었다. 본 실험에서 Charliecloud는 단일 파일 구조의 이미지 파일 형태로 존재하는 타 컨테이너와 달리 각 컴퓨트 노드의 로컬 스토리지에 디렉터리 형태로 저장이 되어 있다. 이러한 이유로 컨테이너 이미지로부터 컨테이너를 로드하는 데에 소요되는 시간이 최소화되어 Charliecloud의 컨테이너 로드 타임이 가장 적게 나타난 것으로 판단된다.

1.3 Memory Usage

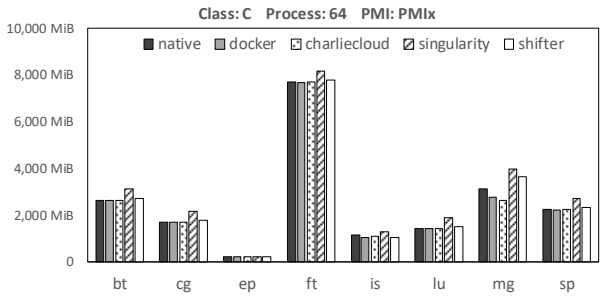

Fig. 5. Comparison chart of memory usages on different container within type of benchmark

Table 3. Table of memory usages

Bench	Min	Max	NT/Min	DK/Min	CC/Min	SG/Min	SF/Min
BT	DK	SG	+0.2%	-	+0.6%	**+18.2%**	+3.8%
CG	NT	SG	-	+0.0%	+0.5%	**+28.1%**	+5.5%
EP	SF	DK	+2.1%	**+2.3%**	+2.0%	+0.3%	-
FT	DK	SG	+0.1%	-	+0.2%	**+6.2%**	+1.3%
IS	SF	SG	+10.8%	+2.0%	+4.5%	**+24.9%**	-
LU	DK	SG	+0.1%	-	+0.9%	**+33.9%**	+6.9%
MG	CC	SG	+17.3%	+4.6%	-	**+50.4%**	+38.0%
SP	DK	SG	+0.3%	-	+0.7%	**+21.5%**	+4.4%

* NT(native), DK(docker), CC(charliecloud), SG(singularity), SF(shifter)

Fig. 5는 “1.1 Benchmark Execution Time”과 동일한 조건에서의 실험 결과 중 각 컨테이너별 메모리 사용량을 나타낸다. 해당 결과는 각 컨테이너의 구현이나 아키텍처에 따라 메모리 사용량 차이가 나타났고 그 차이는 Singularity, Shifter, Docker, Charliecloud 순으로 많은 메모리를 사용하는 것으로 나타났다. Singularity는 MG 연산에서 최소 메모리 사용량을 보인 Charliecloud에

비해 50.4%를 많은 메모리를 사용하였으며, EP의 경우 반대로 모든 컨테이너의 메모리 사용량의 차이가 최대 2.3% 이내로 매우 유사한 수치가 관측되었다.

2. Analysis Performance of PMI Library

2.1 Benchmark Execution Time

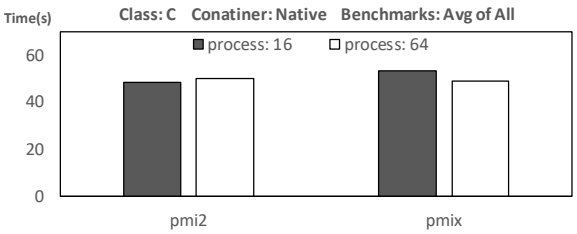

Fig. 6. Comparison chart of NPB execution times on different PMI library

Fig. 6은 프로세스 수에 따른 PMI-2와 PMIx 간의 MPI 어플리케이션 수행시간 비교를 나타내는 그림이다. 실험결과 보다 적은 프로세스 수의 환경(16개 프로세스)에서는 PMIx의 수행시간이 PMI-2의 수행시간에 비해 10.0% 느린 것으로 나타났으나 사용된 프로세스 수가 커짐(64 프로세스)에 따라 기존 결과와 달리 PMI-2의 수행시간이 PMIx에 비해 2% 느린 결과가 나타났다. 본 실험결과를 통해 사용된 프로세스 수가 적을 때는 PMIx보다 PMI-2가 더 나은 성능을 보이며 사용된 프로세스 수가 늘어남에 따라 PMIx가 더 높은 성능을 보인 것을 알 수 있다.

Table 4. Comparison chart of Benchmark Excution times two different PMI libraries within type of benchmark and container

	Charliecloud	Docker	Native	Shifter	Singularity
BT	Equal (-0.68%)	Equal (-0.38%)	Equal (0.62%)	Equal (0.29%)	Equal (0.22%)
CG	Equal (0.05%)	Equal (0.13%)	Equal (0.03%)	Equal (0.09%)	Equal (0.06%)
EP	Equal (-0.08%)	Equal (-0.19%)	Equal (0.08%)	Equal (-0.62%)	Equal (-0.09%)
FT	Equal (0.95%)	Equal (0.92%)	Equal (0.36%)	Equal (0.54%)	Equal (0.25%)
IS	Equal (0.33%)	Equal (-0.31%)	Equal (-0.38%)	Equal (0.62%)	PMIx (-1.10%)
LU	PMI-2 (1.11%)	Equal (0.36%)	Equal (0.81%)	Equal (-0.21%)	Equal (0.52%)
MG	Equal (-0.11%)	Equal (-0.33%)	Equal (-0.50%)	Equal (0.25%)	PMI-2 (1.12%)
SP	PMI-2 (1.22%)	Equal (0.92%)	Equal (-0.13%)	PMI-2 (2.46%)	Equal (-0.44%)

* The number in parentheses indicates the difference of PMIx compared to PMI-2

Table. 4는 추가적으로 PMI 라이브러리가 사용된 컨테이너 종류에 따라 성능 차이를 보이는지 확인하기 위한 표다. 실험결과 대부분의 컨테이너에서는 PMI-2와 PMIx 간의 성능 차이는 나타나지 않았음을 알 수 있다.

2.2 Slurm Job + Container INIT Time

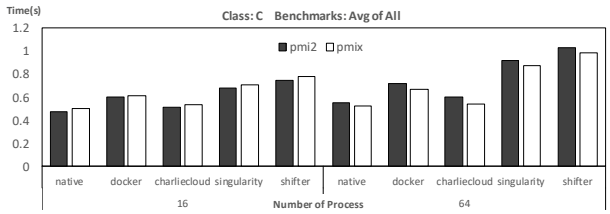

Fig. 7. Comparison chart of Slurm Job + Container INIT within type of container

Table 5. Comparison table of with Slurm Job + Container INIT times on different PMI libraries

time (s)	16			64		
	PMI-2		PMIx	PMI-2		PMIx
NT	0.476	<	0.502	0.554	>	0.530
DK	0.601	<	0.613	0.716	>	0.669
CC	0.516	<	0.532	0.603	>	0.540
SG	0.682	<	0.709	0.921	>	0.880
SF	0.746	<	0.780	1.026	>	0.989

* NT(native), DK(docker), CC(charliecloud), SG(singularity), SF(shifter)

Fig. 7은 PMI 라이브러리 간의 성능차이를 비교하기 위한 모든 벤치마크 수행시간의 평균을 나타낸다. 이 값들은 외부 관측으로 측정한 총 작업 수행시간 측정치에서 벤치마크 자체 수행시간을 제외한 값이다. 벤치마크 자체 수행시간은 벤치마크 프로세스가 초기화된 이후 측정된 정보이므로 프로세스 매니저에 의한 프로세스 초기화 및 종료시간과 컨테이너 초기화 시간 등 초기화 과정의 모든 시간이 포함되지 않는다. 이를 통해 전체 수행시간에서 벤치마크 과정에서 소모된 시간을 제외한 값을 알 수 있다.

실험 결과 앞선 2.1 절의 어플리케이션 수행결과와 비슷한 양상의 결과가 나타났다. 더 적은 프로세스 수의 환경(16 프로세스)에서는 PMI-2의 수행시간이 PMIx에 비해 최대 5.1% 빠른 결과가 나타났으며, 사용된 프로세스 수가 커짐(64 프로세스)에 따라 전체적으로 PMIx가 PMI-2에 비해 더 적은 수행시간을 보이며 그 차이는 최대 11.5% 차이를 보였다. 따라서 어플리케이션 수행시간을 제외한 총 수행시간 결과에서도 PMIx는 사용된 프로세스의 수가 적으면 PMI-2와 성능 차이를 크게 보이지 않지만, 더 많은 수의 프로세스 수가 사용되면 PMIx에 비해 좋은 성능이 나타날 것으로 예측된다.

2.3 Memory Usage

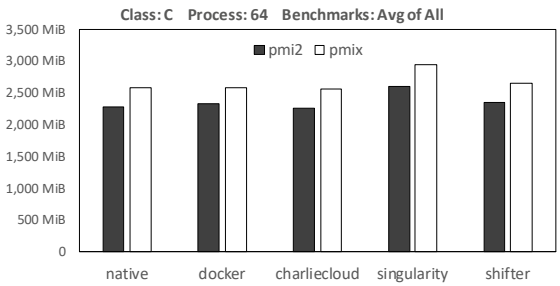

Fig. 8. Comparison chart of average memory usages on different PMI library

Fig. 8는 2.1절의 실험과 동일한 실험의 결과로 메모리 사용량을 비교할 수 있다. 그 결과 기존 선행 연구들이 수백 개 이상의 프로세스를 사용한 결과와 달리 대부분의 경우에서 모두 PMIx의 메모리 사용량이 PMI-2의 사용량에 비해 최대 13.1% 많은 소모량을 보였다. 이는 PMIx의 기본적인 아키텍처에 따른 초기 메모리 사용량이 높기 때문에 발생한 현상이다.

V. Conclusions

본 연구에서는 여러 HPC 컨테이너와 Docker 컨테이너 사이의 성능 차이와 PMI 라이브러리 간의 성능차이를 수행 시간과 컨테이너 초기화 시간 그리고 메모리 사용량을 기준으로 하여 비교실험을 진행하였다. 그 결과 HPC 컨테이너와 Docker 컨테이너의 환경에서 어플리케이션을 수행한 경우 모두 네이티브 환경과 유사한 수행결과를 보이거나 경우에 따라 오히려 네이티브보다 다소 빠른 수행시간 결과를 보였다. 따라서 컨테이너 기술이 제공하는 이식성, 배포 용이성, 편의성 등 다양한 장점들과 함께 성능적인 측면이나 HPC와의 호환성까지 충족시키는 다양한 HPC 컨테이너를 채택하여 기존 HPC 환경에도 컨테이너 기술을 도입하는 것이 좋을 것으로 판단된다.

함께 진행한 PMI 라이브러리 성능 비교실험 결과로 다음과 같은 사항들을 발견하였다. 첫째, 적은 프로세스 수가 사용된 환경에서는 PMI-2가 PMIx에 비해 좋은 성능을 보인다. 둘째, 적은 프로세스 수가 사용된 환경에서는 PMIx가 PMI-2에 비해 많은 메모리를 사용하였다. 따라서 모든 경우에서 PMI-2의 개선 버전인 PMIx를 사용하는 것은 권장되지 않고 사용되는 환경을 고려하여 PMI 라이브러리를 선별하여 채택할 것이 권장된다.

향후 연구 방향으로는 본 논문에서 사용된 NPB 외 I/O Intensive 특징을 보이는 벤치마크 등 다양한 벤치마크 어플리케이션을 사용하여 보다 다각화된 결과를 얻을 예정이며, PMI의 성능 차이를 파악하기 위해 더 많은 노드 및 쓰레드 수의 환경에서 작업할 예정이다.

ACKNOWLEDGEMENT

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. NRF-2019 R1C1C1006990) and by the BK21 Plus project (SW Human Resource Development Program for Supporting Smart Life) funded by the Ministry of Education, School of Computer Science and Engineering, Kyungpook National University, Korea (21A20131600005).

REFERENCES

[1] D.M. Jacobsen and R.S. Canon. “Contain this, unleashing docker for hpc,” Proceedings of the Cray User Group, pp. 33-49, Apr. 2015.

[2] G.M. Kurtzer, V. Sochat and M.W. Bauer. “Singularity: Scientific containers for mobility of compute,” PloS one, Vol. 12, No. 5, e0177459, May. 2017. DOI: 10.1371/journal.pone.0177459

[3] R. Priedhorsky and T. Randles, “Charliecloud: unprivileged containers for user-defined software stacks in HPC," Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017, pp. 1-10, Nov. 2017. DOI: 10.1145/3126908.3126925

[4] L. Gerhardt, W. Bhimji, S. Cannon, M. Fasel, D. Jacobsen, M. Mustafa, J. Porter and V. Tsulaia, “Shifter: Containers for HPC," Journal of Physics: Conference Series, Vol. 898, No. 8, pp. 082021, Nov. 2017. DOI: 10.1088/1742-6596/898/8/082021

[5] P. Balaji, D. Buntians, D. Goodell, W. Gropp, J. Krishna, E. Lusk and R. Thakur, “PMI: A scalable parallel process-management interface for extreme-scale systems," European MPI Users' Group Meeting, pp. 31-41, Nov. 2010. DOI: 10.1007/978-3-642-15646-5_4

[6] R.H. Castain, J. Hursey, A. Bouteille and D. Solt , “PMIx: process management for exascale environments," Parallel Computing, Vol. 79, pp. 9-29, Nov. 2018. DOI: 10.1016/j.parco.2018.08.002

[7] S. Chakraborty, H. Subramoni, J. Perkins and D.K. Panda, “SHMEMPMI -- Shared Memory Based PMI for Improved Performance and Scalability," 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp.60-69, May. 2016 DOI: 10.1109/CCGrid.2016.99

[8] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan and S.K. Weeratunga, “The nas parallel benchmarks," International Journal of High Performance Computing Applications, VOL. 5, No. 3, pp. 63-73, Sep. 1991.

[9] S. Soltesz, H. Pötzl, M.E. Fiuczynski A., Bavier, and L. Peterson, “Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors," ACM SIGOPS Operating Systems Review, p.p. 275-287, Mar. 2007. DOI: 10.1145/1272996.1273025

[10] M.G. Xavier, M.V. Neves, F.D. Rossi, T.C. Ferreto, T. Lange and C.A.F De Rose, “Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments," 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, p.p. 233-240, February 2013. DOI: 10.1109/PDP.2013.41

[11] W. Felter, A. Ferreira, R. Rajamony and J. Rubio, “An updated performance comparison of virtual machines and linux containers," 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), p.p. 171-172, March 2015. DOI:10.1109/ISPASS.2015.7095802

[12] M. de Bayser, and R. Cerqueira, "Integrating MPI with Docker for HPC", 2017 IEEE International Conference on Cloud Engineering (IC2E), p.p. 259-265, April 2017. DOI: 10.1109/IC2E.2017.40

[13] A. Azab, “Enabling Docker Containers for High-Performance and Many-Task Computing,” 2017 IEEE International Conference on Cloud Engineering (IC2E), p.p. 279-285, April 2017. DOI: 10.1109/IC2E.2017.52

[14] J. Sparks, “Enabling Docker for HPC,” Concurrency and computation, Vol. 31, No. 16, e5018, July 2018. DOI: 10.1002/cpe.5018

[15] A.J. Younge, K. Pedretti, R.E. Grant and R. Brightwell, “A Tale of Two Systems Using Containers to Deploy HPC Applications on Supercomputers and Clouds,” 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), p.p. 74-81, Dec. 2017. DOI: 10.1109/CloudCom.2017.40

[16] O. Rudyy, M. Garcia-Gasulla, F. Mantovani, A. Santiago, R. Sirvent, and M. Vázquez, “Containers in hpc: A scalability and portability study in production biological simulations,” 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), p.p. 567-577, May 2019. DOI: 10.1109/IPDPS.2019.00066

[17] A. Torrez, T. Randles and R. Priedhorsky, “HPC container runtimes have minimal or no performance impact," 2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC), pp.37-42, Nov. 2019 . DOI: 10.1109/CANOPIE-HPC49598.2019.00010

Authors

Jaeryun Lee received the B.S. degree in Computer Software from Daegu University, Korea, in 2019. He is currently doing a M.S Course in the Department of Computer Science and Engineering at Kyungpook National University from 2020. He is interested in cloud computing, and distributed systems.

Yunchang Chae received the B.S. degree in Computer Science and Engineering from Kyungpook National University, Daegu, Korea in 2020 and currently doing a M.S course from 2020 in Kyungpook National University. His interested fields are cloud computing, performance modeling and distributed systems.

Byungchul Tak received his B.S. degree from Yonsei University in 2000, M.S. from KAIST in 2003 and Ph.D. degrees in Computer Science and Engineering from the Pennsylvania State University at University Park in 2012. Dr. Tak joined the faculty of the Department of Computer Science at Kyungpook National University, Dague, Korea, in 2017. He is currently an Assistant Professor in the Department of Computer Science. His research interests are in cloud computing, distributed systems, operating system and big data analytics.