Performance evaluation of MEGADOCK protein–protein interaction prediction system implemented with distributed containers on a cloud computing environment

Kento Aoyama¹,², Yuki Yamamoto¹, Masahito Ohue¹, and Yutaka Akiyama¹

¹ Department of Computer Science, School of Computing, Tokyo Institute of Technology, Tokyo, Japan
² AIST-Tokyo Tech Real World Big-Data Computation Open Innovation Laboratory (RWBC-OIL), National Institute of Advanced Industrial Science and Technology, Ibaraki, Japan

Email: {aoyama, y_yamamoto}@bi.c.titech.ac.jp, {ohue, akiyama}@c.titech.ac.jp

Abstract—Container-based virtualization, a lightweight virtualization technology, has begun to be introduced into large-scale parallel computing environments. In the bioinformatics field, where various dependent libraries and software tools need to be combined, the container technology that isolates the software environment and enables rapid distribution as in an immediate executable format, is expected to have many benefits. In this study, we employed Docker, which is an implementation of Linux containers, and implemented a distributed computing environment of our original protein–protein interaction prediction system, MEGADOCK, with virtual machine instances on Microsoft Azure cloud computing environment, and evaluated its parallel performance. Both when MEGADOCK was directly performed on the virtual machine and also when is performed with Docker containers of MEGADOCK on the virtual machine, the execution speed achieved was almost equal even if the number of worker cores was increased up to approximately 500 cores. On the standardization of portable and executable software environments, the container techniques have large contributions in order to improve productivity and reproducibility of scientific research.

Keywords: container-based virtualization, cloud computing, MPI, Docker, MEGADOCK, protein–protein interaction (PPI)

1. Introduction

In the field of bioinformatics and computational biology, various software are utilized for research activities. Management of software environments such as dependent software libraries is one of the most challenging issues in computational research. Recently, as a solution to complication of the software environment, introduction of the container-based virtualization technology, which is an approach using virtualization technologies with lightweight and excellent performance, is advancing [1], [2]. Particularly in the field of genome research, pipeline software systems consisting of multiple pieces of software are commonly used, which tend to complicate the environment. For this reason, case studies have been reported, including those on environmental management and distributed processing using the container-based virtualization technology [3], [4].

In the container-based virtualization, a software execution environment, including dependent software libraries and execution binaries, is isolated as a container, and immediate software distribution as an executable format can be realized [5], [6]. This feature facilitates the management of the software environment and distribution, thus introducing new software. It has also been reported that the container-based virtualization performs better than the hypervisor-based virtualization, which is used to implement common virtual machines (VMs), and when properly configured, performs almost as well as running on a physical machine [7].

Container-based virtualization has developed in areas such as dynamic load balancing in parallel distributed platforms on cloud environments because it enables rapid environment building and application abstraction [8]. Although the introduction has not advanced in the computer environments at research institutes and universities due to the concern regarding performance degradation by virtualization, there is a tailwind through excellent benchmark results on parallel computing environments and application research case reports. Recently, the container-based virtualization technology has begun to be adopted in supercomputing environments.

As an example, the National Energy Research Scientific Computing Center (NERSC) in the United States, which has a large-scale supercomputer Cori, developed an open-source software for the container-based virtualization of high-performance computing called Shifter [9]. There are reports on the utilization of the container-based virtualization in various application researches from NERSC [10]. In addition, another container implementation, Singularity[11], is available on the TSUBAME 3.0 supercomputer of Tokyo Institute of Technology and AI Bridging Cloud Infrastructure (ABCI) of National Institute of Advanced Industrial Science and Technology (AIST), both of which are world’s first-tier supercomputing environments in Japan [12]. From the above, correspondence to the container-based virtualization is urgent even in the bioinformatics and computational biology fields.
In this study, we focused on MEGADOCK [13], [14], a protein–protein interaction (PPI) prediction software, as an example of bioinformatics software, that can predict PPIs between various proteins by parallel computing. We introduced distributed processing on MEGADOCK using Docker containers [1], an implementation of the container-based virtualization, and then evaluated its computational performance on the Microsoft Azure public cloud computing environment [15] by comparing it with a simple parallel implementation with message passing interface (MPI) [16].

2. Overview of container-based virtualization

There are two major concepts of virtualization approaches in the context of applications running on a cloud environment: hypervisor-based and container-based.

2.1 Hypervisor-based virtualization

In hypervisor-based virtualization, the virtual environment is provided by a higher-level “hypervisor” that further manages the OS (Supervisor) that manages the application (Fig. 1, left). The “virtual machine” (VM) widely used in general cloud environments is provided by the hypervisor-based virtualization that enables users to use various operating systems such as Windows and Linux OS as Guest OS, which is managed by a hypervisor running on the Host OS or hardware.

There are various types of implementations for the hypervisor-based virtualization, such as Kernel Virtual Machine (KVM) [17], Hyper-V [18] used in Microsoft Azure, XEN [19] used in Amazon Web Service, and VMware [20].

2.2 Container-based virtualization

In container-based virtualization, containers are realized by the isolation of the namespaces of user processes running on the host OS (Fig. 1, right). These virtualizations are mainly implemented by the namespace [5], one of the Linux kernel features. The namespace can isolate user processes from global namespaces to individual namespaces, and enable us to use different namespaces for mounting points, processes, users, networks, hostnames, etc. Therefore, users can touch an isolated application environment that is separate from the host environment. The container-based virtualization is sometimes called as kernels-sharing virtualization because the containers running on the same host commonly use the same kernel.

According to a previous study, performance overheads of a container in various aspects are smaller than those of the VMs because the resource management in containers is under the direct control of its host kernels [7]. Moreover, the data size of the container images tends to be smaller than that of the VMs. This offers a significant advantage on the application deployment.

2.3 Docker

Docker [1] is the most popular set of tools and platform for managing, deploying, sharing of Linux containers. It is an open-sourced software on the GitHub repository, operated by Moby [21] project, written in Golang, contributed by worldwide developers. There are several related toolsets and services of Docker ecosystems, such as Docker Hub [6], the largest container image registry service to exchange user-developed container images. Docker Machine [22] provides container environments to Windows and MacOS using a combination of Docker and the hypervisor-based approach.

2.3.1 Sharing container image via Docker Hub

A container image can include all the dependencies necessary to execute the target application: code, runtime, system tools, system libraries, and configurations. Thus, it enables us to reproduce the same application environment in the container as we build it, and deploy onto the machine with other specifications. Users easily share their own application environment with each other through uploading (push) of container images via the Docker Hub [6], the largest container image registry service for Docker containers, and downloading (pull) of the same container image onto a different machine environment (Fig. 2).

2.3.2 Filesystem of Docker container image

Docker adopts a layered file system as the file system in the container image to reduce the total file size of the images. Every image layer corresponds to a batch of user operations or differential changes of the file system in the container and each has a hashed id. Its layered file system has great benefits on reproducing the operations, rollbacks, reuse of the same container layer that has the same hash id; these contribute to the reproducibility of the application and also of the research. Note that such a layered file system does not show good performance at the latency of the file I/O due to its differential layer management such that we usually
3. MEGADOCK

MEGADOCK is a PPI prediction software for a large-scale parallel computing environment, which was developed by Ohue et al. [13], [14]. MEGADOCK supports MPI, OpenMP, and GPU parallelization, and has achieved massive parallel computing on TSUBAME 2.5/3.0, K computer, etc. The MPI parallel implementation on Microsoft Azure [15] public cloud (MEGADOCK-Azure [16]) as well as the predicted PPI database MEGADOCK-Web [23] have been developed to promote the use of this software in more general environments.

3.1 MEGADOCK-Azure [16]

MEGADOCK-Azure has three main functions: client, resource group management, and task distribution on Microsoft Azure VMs. Fig. 4 shows a diagram of the system architecture of parallel processing infrastructure on Microsoft Azure VMs using MEGADOCK-Azure.

The client function uses the Azure command line interface (AzureCLI) to locate the necessary compute resources on Microsoft Azure, including VMs and virtual networks. It can directly perform the data I/O through the mount point where the target data are stored and attached to the container.

3.2 MEGADOCK with container-based virtualization

The versatility of a software dependent environment with deployment/management problems and improvement of its execution performance are still pressing issues. The use of a cloud computing environment, such as the case of MEGADOCK-Azure, which is based on VMs, is one of the solutions, but there are still concerns about the complexity to reuse the entire existing local computing resource, vendor lock-in problems, and performance overheads due to the hypervisor-based virtualization.

In this study, we tried to solve these problems by using Docker containers. Introduction of container techniques into MEGADOCK has the following advantages:

- Docker containers are able to run on almost all environments over various cloud computing infrastructure using the same container image as well as on our local environments.
The container-based virtualization approach generally shows superior performance than the hypervisor-based virtualization approach both in running and deploying. There are compatible container environments available on several high-performance computing (HPC) environments such as TSUBAME 3.0 and ABCI supercomputers, such that it can even be a model of standard application package in HPC environments.

To maintain compatibility between different environments, we implemented the MEGADOCK system using Docker containers running on VM instances of Microsoft Azure by referring to the MEGADOCK-Azure architecture. The system overview is shown in Fig. 5.

The Docker container image used for the system is generated from a build recipe (Dockerfile) and can run the same MEGADOCK calculation as VM images. The containers over different VM instances are connected on an overlay network using Docker networking functions and are able to communicate with each other using the MPI library. Thereby, we can run the docking calculation of MEGADOCK on the cloud environment using containers, and the system is compatible with other environments.

4. Performance evaluation

We present two experiments to evaluate the parallel performance of the container-distributed MEGADOCK system using Docker features.

4.1 Experiment 1. Parallel performance on a cloud environment

Firstly, we measured the execution time and its parallel speed-up ratio of distributed MEGADOCK system by changing the number of worker processor cores of the VM instances in Microsoft Azure under the master-worker model on the MPIDP framework.

4.1.1 Experimental setup

We selected Standard_D14_v2, a high-end VM instance on Microsoft Azure. The specifications of the instance are listed in Table 1, and the software environment is shown in Table 2.

| Table 1: Experiment 1. Specifications of Standard_D14_v2 |
|------------------|---------------------------------|------------------|
| CPU              | Intel Xeon E5-2673, 2.40 [GHz] × 16 [core] |
| Memory           | 112 [GB]                          |
| Local SSD        | 800 [GB]                          |

The measured data were obtained from the result of the time command, and the median of 3 runs of the calculations was selected. To avoid slower data transfer time between nodes, all output results of docking calculation were generated onto local SSDs attached on each VM instance. On the Docker container case, to avoid the unnecessary performance degradation due to the layered file system of the container, all output files are stored to a data volume on local SSD which was mounted on the inside of the container.

For MPI and OpenMP configurations, the number of processes is selected such that every node has four processes, and the number of threads is fixed to four in all the cases (OMP_NUM_THREADS=4). On the container case, each node runs one container for MEGADOCK calculation, and it performs as same as when it directly runs on a VM.

4.1.2 Dataset

Protein hetero-dimer complex structure data of the protein–protein docking benchmark version 1.0 [26] were used for performance evaluation. Whole 59 hetero-dimer protein complexes were used and all-to-all combinations of each binding partner (59 × 59 = 3,481 pairs) were calculated to predict their possible PPIs.

4.1.3 Experiment result

The execution time of MEGADOCK running with the Docker containers on the VM instances and MEGADOCK directly running on the VM instances (MEGADOCK-Azure) is shown in Fig. 6. Each bar shows the execution time on the number of VM instances, and the error-bars show the standard deviation in the measurements.

Fig. 7 shows the scalability in strong-scaling for the same results. The label “ideal” indicates the ideal linear scaling to the number of worker cores.

According to the result of scalability, both of them achieved a good speed-up of up to 476 worker cores. It was ×35.5 speed-up in the case of directly running on the VMs, and ×36.6 in the case of the Docker containers on the VMs.
The speed-ups were almost equivalent in this experiment and they indicate that the performance overhead on the Docker containers is small when running on the VM instance of the cloud environment.

The MEGADOCK execution load is mainly composed of independent 3D-fast Fourier transform (FFT) convolutions on each single node even in the MPI version such that it tends to be a compute-intensive workload, not a data I/O or network-intensive; therefore, similar to the situation mentioned in the Linux container performance profile reports [7], MEGADOCK calculation on the distributed containers environment also performs well.

4.2 Experiment 2. Execution performance on a GPU-attached bare-metal node

Additionally, to investigate the overhead of the container-based virtualization, we measured the execution time of MEGADOCK running on a local node with various conditions and compared the results of performance.

4.2.1 Experimental setup

We used a bare-metal (not a virtual machine) GPU-attached node for this experiment. The specifications are listed in Table 3, and environmental settings are shown in Table 4.

<table>
<thead>
<tr>
<th>OS (image)</th>
<th>CentOS</th>
<th>library/ubuntu</th>
<th>nvidia/cuda</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version (tag)</td>
<td>7.2.1511</td>
<td>14.04</td>
<td>8.0-devel</td>
</tr>
<tr>
<td>Linux Kernel</td>
<td>3.10.0</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>GCC</td>
<td>4.8.5</td>
<td>4.8.4</td>
<td>4.8.4</td>
</tr>
<tr>
<td>FFTW</td>
<td>3.3.5</td>
<td>3.3.5</td>
<td>3.3.5</td>
</tr>
<tr>
<td>OpenMPI</td>
<td>1.10.0</td>
<td>1.6.5</td>
<td>N/A</td>
</tr>
<tr>
<td>Docker Engine</td>
<td>1.12.3</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>NVIDIA Driver</td>
<td>8.0.44</td>
<td>N/A</td>
<td>8.0.44</td>
</tr>
<tr>
<td>NVIDIA Docker</td>
<td>1.0.0 rc.3</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>NVIDIA Driver</td>
<td>367.48</td>
<td>N/A</td>
<td>367.48</td>
</tr>
</tbody>
</table>

We used the physical machine and measured the execution time of the MEGADOCK calculation for 100 pairs of pdb data, for each of the following conditions (Fig. 8):

(a) MEGADOCK using MPI library,
(b) MEGADOCK using MPI library in a container,
(c) MEGADOCK using GPU, and
(d) MEGADOCK using GPU in a container.
We used NVIDIA Docker [27] to invoke NVIDIA GPU device from inside of the Docker container. All the result data of MEGADOCK calculation were output to the network-attached storage (NAS) on the same network in all the cases, and we used the “volume” option to mount the path to the NAS when using Dockers.

The data were obtained by the time command to measure the duration from start to end of the execution, and we selected the median of six repeated runs.

As it had the same configuration as that of Experiment 1, the number of Docker containers per node was one, the number of processes was four, and the number of threads per process was fixed as four.

### 4.2.2 Dataset

PPI predictions were performed for 100 pairs of pdb data randomly obtained from the KEGG pathway [28].

### 4.2.3 Experiment result

Fig. 9: Experiment 2: Execution time comparison between MEGADOCK running on bare-metal and with Docker container

Fig. 9 show the result for the execution time of the MEGADOCK calculation for 100 pairs in each condition. As the result of experiment, MEGADOCK calculation using the MPI library in the Docker container (b) was approximately 6.3% slower than the same calculation using the MPI library on the bare-metal environment (a). Note that this experiment was performed on a single node of a bare-metal environment such that the communication cost can be made sufficiently small because there was no inter-node MPI communication.

On the other hand, MEGADOCK calculation using GPU in the Docker container (d) performed almost the same result as calculations using the GPU on the bare-metal environment (c). This means that the MEGADOCK-GPU, which does not use the MPI library, performs at the same speed of execution even when it runs in the Docker container.

## 5. Discussion

In Experiment 1, the execution on a single VM instance with the MEGADOCK-Azure (VM) particularly consumed time but the reason was unknown, and that affected the result of scalability. This irregularity increases the scalability more than expected. The reason should be investigated; however, it is difficult because it is time-consuming calculation.

We did not achieve multiple GPU nodes parallelization using the MPI library in this study; however, now the VM instances attached to NVIDIA GPU devices are generally available. Moreover, there are more sophisticated VM instances for the HPC applications on Microsoft Azure that is connected by the InfiniBand each other and supports low-latency communication using remote direct memory access (RDMA). We have already performed experiments over multiple VM instances with GPUs [16] or HPC instances; therefore, further experiments with the Docker containers is our future challenge.

Additionally, there is a possibility to introduce an alternative approach for task distributions of MEGADOCK. In this study, we used the MPIDP framework, which uses the MPI library to realize a dynamic task distribution over multiple nodes because it is the built-in-function of MEGADOCK; however, that can be alternated by another framework such as MapReduce. Moreover, our current implementation lacks the functionality to recover from unpredictable failures caused by the MPI processes, containers, VM instances, or hardware such that we should introduce a more fault-tolerant framework that has functions of auto-recovery from failure and redundancy of executions. We are considering introducing container orchestration frameworks such as Kubernetes [29] and Apache Mesosphere [30] to resolve the issues.

## 6. Conclusion

We implemented a protein–protein interaction prediction system, MEGADOCK, using Docker containers and its networking functions on the VM instances of Microsoft Azure. We confirmed that the performance is almost equivalent to
the same calculation directly performed on the VM instances through the benchmark experiment of protein docking calculations. Both when MEGADOCK directly runs on the virtual machine and when it runs with the Docker containers of MEGADOCK on the virtual machine, the execution speed achieved was almost equal even when the number of worker cores increased up to approximately 500 cores.

In the second experiment, we performed MEGADOCK calculation with MPI/GPU on a single bare-metal machine both in a Docker container and in a bare-metal environment, to investigate the effects of performance overhead of container virtualization. Results showed a small performance degradation of approximately 6.3% on the MPI version in the container case compared with the bare-metal; however, it was almost equal in the GPU version with the container and bare-metal.

The containers enable us to isolate software dependencies and the system software stacks, which offer a great advantage to the users in sharing software packages through platforms, thereby making it easy to distribute the latest research achievement. Virtualization technologies have been evolved in the context of the general cloud computing environment; however, in the current era, many research institutions have introduced container environments into their HPC infrastructures. To improve productivity and retain scientific reproducibility, it is necessary to introduce such software engineering techniques into research activities.

## Code Availability

The entire source code of MEGADOCK is open sourced on the GitHub repository. There is also a build recipe (Dockerfile) for building Docker container images for performing the PPI prediction calculations in various environments.

https://github.com/akiyamalab/MEGADOCK

## Acknowledgment

This work was partially supported by KAKENHI (Grant No. 17H01814 and 18K18149) from the Japan Society for the Promotion of Science (JSPS), the Program for Building Regional Innovation Ecosystems “Program to Industrialize an Innovative Middle Molecule Drug Discovery Flow through Fusion of Computational Drug Design and Chemical Synthesis Technology” from the Japanese Ministry of Education, Culture, Sports, Science and Technology (MEXT), the Research Complex Program “Wellbeing Research Campus: Creating new values through technological and social innovation” from JST, Microsoft Business Investment Funding from Microsoft Corp., and Leave a Nest Co., Ltd.

## References