Containerization Support for Multi-Agent Spatial Simulation

Jiashun Gou

A report
submitted in partial fulfillment of the
requirement for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington Bothell

Spring 2021

Project Committee:
Munehiro Fukuda, Ph.D., Committee Chair
Kevin Sung, Ph.D., Committee Member
William Erdly, Ph.D., Committee Member
Abstract

Containerization Support for Multi-Agent Spatial Simulation

Jiashun Gou

Chair of the Supervisory Committee
Dr. Munehiro Fukuda
Computing and Software Systems

The Multi-Agent Spatial Simulation (MASS) library is a parallel-computing library designed to execute applications over a cluster of computing nodes. In this capstone project, we aim to improve MASS developers’ experience by adding containerization support to the MASS library and its applications as well as adding a Continuous Integration/Continuous Delivery (CI/CD) pipeline to all related code repositories. First, we designed and implemented containerization support for two versions of MASS libraries and three sample containerized MASS applications. Second, we added a CI/CD pipeline to each code repository of containerized MASS library and MASS applications. Third, we evaluated implementation of the containerized MASS library and applications from five aspects, including reliability, usability, efficiency, maintainability, and portability. In comparison to the original MASS, the containerized MASS library and its applications demonstrates a noticeable increase in usability and maintainability. The project successfully carries out two achievements: (1) containerized MASS and applications provides MASS developers a consistent developing environment and (2) the CI/CD pipeline simplifies MASS developers’ workload, especially testing and releasing procedures.
# TABLE OF CONTENTS

List of Figures .............................................................................................................................. ii
List of Tables ............................................................................................................................... iii
List of Listings .............................................................................................................................. iv

## Chapter 1. Introduction ............................................................................................................ 1
  1.1 Background ........................................................................................................................... 1
  1.2 Project Goals ........................................................................................................................ 2
  1.3 Project Overview .................................................................................................................. 3

## Chapter 2. Related Work ......................................................................................................... 5
  2.1 Why Docker with Docker Swarm ......................................................................................... 5
  2.2 Similar Frameworks .............................................................................................................. 6
  2.3 Current Challenges ............................................................................................................... 8

## Chapter 3. Containerization and Pipeline of MASS library .................................................. 10
  3.1 Containerized MASS Library and its Applications .............................................................. 10
  3.2 Deploy Containerized MASS Applications ........................................................................... 16
  3.3 Execute Containerized MASS Applications ......................................................................... 19
  3.4 Streamlined CI/CD Experience ............................................................................................ 22

## Chapter 4. Evaluation of MASS Containerization ................................................................. 25
  4.1 Evaluation Environment and Procedures ............................................................................. 25
  4.2 Evaluation on Software Quality .......................................................................................... 26
  4.3 Summary of Evaluation ........................................................................................................ 36

## Chapter 5. Conclusion ............................................................................................................ 38

References .................................................................................................................................... 40
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 (a)</td>
<td>Basic structure of a containerized MASS application</td>
<td>11</td>
</tr>
<tr>
<td>3.1 (b)</td>
<td>Procedure to build a containerized MASS application</td>
<td>12</td>
</tr>
<tr>
<td>3.2 (a)</td>
<td>Procedure to deploy a containerized MASS application on one computing node</td>
<td>17</td>
</tr>
<tr>
<td>3.2 (b)</td>
<td>Procedure to deploy a containerized MASS application on a computing cluster</td>
<td>18</td>
</tr>
<tr>
<td>3.3 (a)</td>
<td>Generate “nodes.xml” for execution on One Computing Node</td>
<td>21</td>
</tr>
<tr>
<td>3.3 (b)</td>
<td>Generate “nodes.xml” for execution on Multiple Computing Nodes</td>
<td>21</td>
</tr>
<tr>
<td>3.4 (a)</td>
<td>Original Workflow</td>
<td>22</td>
</tr>
<tr>
<td>3.4 (b)</td>
<td>New Workflow with CI/CD Pipeline</td>
<td>24</td>
</tr>
<tr>
<td>4.2 (a)</td>
<td>Execution performance of Range Search</td>
<td>28</td>
</tr>
<tr>
<td>4.2 (b)</td>
<td>Downed and automatically revived container</td>
<td>32</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2 Comparison among Apache Spark, COMP Superscalar, Docker, and Singularity</td>
<td>8</td>
</tr>
<tr>
<td>4.2 (a) Usability comparison between Original and Containerized Range Search</td>
<td>31</td>
</tr>
<tr>
<td>4.2 (b) Maintainability comparison between Original and Containerized Range Search</td>
<td>35</td>
</tr>
</tbody>
</table>
# LIST OF LISTINGS

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 (a)</td>
<td>Dockerfile of containerized MASS Java</td>
<td>14</td>
</tr>
<tr>
<td>3.1 (b)</td>
<td>“docker-compose.yml” of containerized MASS Java</td>
<td>14</td>
</tr>
<tr>
<td>3.1 (c)</td>
<td>Dockerfile of containerized MASS Range Search</td>
<td>15</td>
</tr>
<tr>
<td>3.1 (d)</td>
<td>“docker-compose.yml” of containerized MASS Range Search</td>
<td>15</td>
</tr>
</tbody>
</table>
Chapter 1. Introduction

1.1 Background

MASS is a parallel-computing library that stands for Multi-Agent Spatial Simulation whose design is based on the parallelization of agent-based modeling, the concept of which has been highlighted for the past two more decades. However, years of development not only strengthened the functionality but also increased the complexity of the MASS library. Since the MASS is inherently a parallel-computing library, which has a much higher likelihood than other types of software to generate inconsistent results. MASS developers, defined as people who either contribute to the MASS library or invoke the MASS library in their applications, are experiencing an increasingly steep learning curve from three difficulties. First, when developers start learning the mechanism of the MASS library, they are required to manually set up their development environment on machines. Second, when developers encounter unfamiliar errors, other developers need to make sure that the environment is consistent among them and then try to reproduce the error. Third, when developers finish testing their application on their local machine, they need to set up the network environment and move their applications into a computing cluster. These kinds of activities deviate from the product software development life cycle and greatly reduce the development efficiency of MASS developers. To further facilitate the MASS development, this project will introduce containerization, Continuous Integration (CI), and Continuous Delivery (CD) in MASS.

Containerization is a word that is derived from freight transportation. In the world of software engineering, containerization is defined as using a virtualized operating system to wrap
up code into independent containers, which is a standalone unit that includes all its dependencies. Since a container includes every dependency to execute an application, developers can obtain consistent results across different platforms. While other techniques (e.g., virtualization) can achieve the same result, containerization is a more well-rounded approach that can set up the development environment with minimal effort even deploying on a cluster of computing nodes.

We bring back CI in the form of pipelines, which is a series of steps need to conduct after committing code. Some contributions on CI for MASS were made, but they were hard to maintain and forced to abandon years ago. Learned from the experience, we pay close attention to the maintainability and usability in this project.

Continuous Deployment (CD) is a brand-new topic for the MASS. It stands for an automated procedure to verify correctness and to publish code in a production environment. This technique can free MASS developers from manually conducting tests and uploading verified software to the public.

1.2 Project Goals

The overall goals of this project are:

1. Improve development experience with the MASS

There are multiple versions of the MASS library. This project plans to containerize two MASS libraries and a few sample applications and introduce the CI and CD process, alleviating
MASS developers from some redundant manual labor. In addition, MASS application developers can follow sample containerized applications to quickstart their projects.

2. Demonstrate scalability of the MASS

   Containerization can provide MASS, a parallel computing library in nature, a solid foundation to demonstrate its capabilities under intensive communication workload.

3. Explore portability of the MASS

   This project demonstrates the capability of the MASS library in working on different platforms.

4. Building a platform that supports other research interests.

   This project provides developers a uniform experience on containerized MASS applications across complex environments just like a platform, which alleviates developers from fiddling with environments so they can focus on problems they value.

1.3 Project Overview

   In order to achieve these goals, we containerize existing MASS libraries as the first step. Thereafter, using the containerized MASS library, we containerize a few MASS applications. Next, we add pipelines into MASS developers’ workflow since containerization helps us adding pipelines more efficiently. Finally, we evaluate the design and implementation with some relating metrics from ISO/IEC 25010 standard.
This project was designed to keep iterating forward according to feedback from users. Then, collect a resulting score from a survey at the end of this project as a major aspect of evaluation. Particularly, we strengthened attention on usability and maintainability from the beginning of this project since this project targets to providing a better developing experience for MASS developers.
Chapter 2. Related Work

This chapter discusses why we choose Docker as the tool to add containerization support for the MASS library and its applications, as well as other approaches that could increase developer experience while interacting with parallel execution libraries.

2.1 Why Docker with Docker Swarm

Docker [1] is an open-source containerization tool that packages code and all its dependencies. There are other alternative approaches, such as LXD, Linux VServer, Windows Containers. However, comparing to Docker, they all lack on community support. This is a major drawback since containerization is still a new technique that most developers need to learn and we want to minimize amount of effort required.

In addition, virtualization, another tool that packages code and all its dependencies, is functionally comparable to Docker in this project. Although the virtualization technique emerged in 1964 [2], which precedes containerization by 15 years [3], and proved to be a solid approach, research[4] concludes that Docker containers do not show considerable performance deterioration as compared to virtual machines. Also, they proclaim that containers are a great solution for a cybersecurity lab, which is one of our project goals: building a platform that supports other research interests.

Docker Swarm [5] is an open-source containerization container orchestration platform that focuses on managing containerized services. Although Kubernetes [6] is interchangeable

5
with Docker Swarm in this project and it is a production-ready tool with widely available community supports, we decided to adopt Docker Swarm since Kubernetes’s functionality and adaptability comes at the price of learnability and efficiency. Kubernetes has a very steep learning curve due to its robust industrial design. Although Kubernetes allows developers to deploy applications on multiple clusters, both Kubernetes and Docker Swarm use Docker as the foundation for containerization, allowing MASS developers to switch to Kubernetes swiftly when necessary. Also, according to experimentation [7] on Kubernetes against Docker Swarm and bare metal execution on High-Performance Computing (HPC) infrastructure at SUNY Binghamton University, Kubernetes shows four-times larger overhead than Docker Swarm in TCP/IP latency test, which is crucial for MASS as a communication intensive parallel-computing library.

2.2 Similar Frameworks

Researchers have experimented containerization support on various parallel-computing libraries but not on MASS. This section discusses three comparable approaches that are capable of improving developers’ experience in HPC workflow with containerization. Table 2.2 shows a summary of differences among three approaches as well as Docker with CI.

Apache Spark [8] is an open-source distributed computing library, that supports submitting containerized applications to Kubernetes since version 2.3 released in 2018. [9], [10] Kubernetes is a powerful tool that offers incredible reliability and scalability. [11]

COMP Superscalar (COMPSs) [12] is a framework targeting to alleviate stress from developers while they are working with parallel-processing applications. Developers working
with COMPSs proposed a method to containerize parallel application [13]. In their method, they upload Docker images of parallel-computing libraries and applications to DockerHub and deploy these Docker images on cluster nodes via Docker Swarm. They tried to minimize the amount of data needed to upload and to download by separating a parallel program into two layers: a parallel-computing library and the application. Also, they incorporated Docker-Compose into their architecture to help defining configuration for complex applications.

Singularity [14] is a container platform, similar to Docker. And Jenkins is a CI tool that can help developers build, test, and deploy applications. Researchers proposed an approach, in combination of Singularity and Jenkins, to improve developers’ developing experience while working with HPC in 2018 [15]. They used Singularity as a containerization tool as well as Jenkins with Puppet as a CI tool, and demonstrated a performance increase over time with weekly updating code.

Although researchers have conducted plenty experimentations and discovered several techniques to improve developers’ experience via containerization, all these techniques have their own limitations. Apache Spark solely based on Kubernetes, which has an excessive communication overhead [7] and is hard to install [11]. Also, Apache Spark and COMPs mainly focuses on containerizing COMPs applications while we also want to provide even better developing experience on the MASS with CI/CD pipeline. Comparing to Docker, Singularity does not have as much community supports, which can be directly observed from a number of results from Google search. “Singularity container” has 1,760,000 results while “docker container” has 51,700,000 results, around 29 times more results. This is a vital drawback since we want to minimize amount of effort to learn containerization.
While Docker with CI seemingly a best combination for the MASS, it has a potential problem. If the Bitbucket Pipelines changes the coding syntax of the pipeline configuration file, MASS developers may need to spend extra effort to correct the CI/CD pipeline.

Table 2.2 Comparison among Apache Spark, COMP Superscalar, Docker, and Singularity

<table>
<thead>
<tr>
<th></th>
<th>Strengths</th>
<th>Challenges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Spark</td>
<td>Kubernetes is powerful</td>
<td>Kubernetes is overly complicated, Lack of CI</td>
</tr>
<tr>
<td>COMP Superscalar</td>
<td>Improved usability</td>
<td>Lacks community support, Lack of CI</td>
</tr>
<tr>
<td>Docker + CI</td>
<td>Docker is well-documented, Improved usability, Combines with CI</td>
<td>CI/CD pipeline could require maintenance</td>
</tr>
<tr>
<td>Singularity + CI</td>
<td>Combines with CI</td>
<td>Lacks community support, CI/CD pipeline could require maintenance</td>
</tr>
</tbody>
</table>

2.3 Current Challenges

MASS itself has some unique challenges, including a tedious configuration file set-up, a dispersed logging mechanism, and a tremendous verification workload. Comparing to Docker, Singularity does not have as much community supports, which is a vital drawback.

MASS has its own requirement on environment configuration. It requires a “nodes.xml” file that records information about all the nodes that will involve in the computation. Inside the file, a MASS developer needs to specify a list of attributes precisely for each node, including the hostname, the path to java executable, the path to the generated MASS application, the name of...
an authenticated user, and a private key that can authenticate the user. This information is confusing and requires a lot of manual labor in the original MASS development. Also, researchers found out that configuration errors are the major cause of application downtime and IT management cost dominates the total cost of ownership [16], which is one important problem we want to mitigate through this project.

The MASS library also has its own approach on logging debugging or error messages. Since MASS is a parallel-processing library in nature, execution on every node generates its own copy of log in the local hard disk. The current design of MASS does not collect any log from every computing node, so that MASS developers have to retrieve logs from every computing node manually. MASS developers tried to mitigate the difficulty retrieving logs by installing and maintaining a Network File System (NFS). However, NFS takes effort to install, operates in polynomial time [17], and creates excessive messages when developers are not experimenting with all nodes in the cluster.

MASS developers have difficulties validating code they committed to code repository. The MASS library needs a uniform development platform to conduct unit tests and integration tests. Grady Booch introduced CI in 1991 [18] and it has been evolved since then with commendation from other researchers. [19] They showed “70% of most popular projects on GitHub heavily use CI”. In addition, they stated cost to developers for writing and maintaining the configuration for their CI service, especially when they are unfamiliar, is a major factor preventing them from using CI in their projects.
Chapter 3. Containerization and Pipeline of MASS Library

This chapter describes the design on how containerization fits into two versions of MASS libraries: MASS Java and MASS C++, as well as three MASS sample applications: Range Search, Tuberculosis (TB) simulation, and Shortest Path. In addition, this chapter covers the original workflow for MASS developers to contribute their work and how we optimize it with the CI/CD approach.

3.1 Containerized MASS Library and its Applications

3.1.1 Basic Structure of Containerized MASS Library

This project containerizes MASS using Docker as backend. Containerized MASS with docker is structured as shown in Figure 3.1(a). Comparing to an application that invokes the MASS library directly above the operating System, there is one more layer of abstraction named Docker Engine in Figure 3.1(a).

As Docker is designed to use a server-client architecture, this layer of abstraction is made up of two parts, including Dockers Daemon and Docker Client. A Docker Daemon process, at server side, listens to instructions sent from Docker Clients and follows them accordingly. A Docker Client, clearly at client side, could connect with multiple Docker Daemons situated on different machines at the same time. Most work inside this abstraction layer is concentrated inside each of the Docker Daemon processes, which manages system resources, builds Docker containers, and launches Docker containers.
Before building and launching Docker containers, Docker Daemon requires a Docker image, which can be seen as the blueprint while constructing a building. As shown in Figure 3.1(b), we need two-to-five steps to build a Docker image of MASS application from scratch depending on whether we want to upload Docker images to cloud. If we want to upload the Docker image of both MASS library and MASS applications to cloud, we need all five steps, which we will describe in next section below. If MASS developers want to upload the Docker image of MASS library to cloud and use it to build a containerized MASS application on another machine, they need steps one to four. If MASS developers just want to build the MASS application locally, which is the most common practice, they need two steps.

Figure 3.1(a) Basic structure of a containerized MASS application on two nodes with four containers
3.1.2 How to Containerize MASS library and MASS Application

We mentioned there are a maximum of five steps to build a Docker image for MASS application. The following gives the details of the five steps to build a Docker image for MASS applications.

Step 1. Building a Docker image for MASS libraries:

We need an additional “Dockerfile” aside from the existing MASS library. As shown in Listing 3.1(a), a Dockerfile containerizes the MASS Java library. The first line inherits another Docker image from the cloud, clearly defines Java and maven versions for building the MASS library, allowing developers to upgrade or downgrade dependencies easily. On line 7, we define
path to compiled library in environment, which can be directly used while building containerized
MASS applications. To further coordinate the name of the generated Docker image and simplify
the building process, we need an additional “docker-compose.yml” file as shown in Listing 3.1(b).

Step 2. Uploading the Docker image:

This is straightforward with one Line Of Input (LOI) in terminal.

Step 3. Building a Docker image of MASS application:

This step requires an additional Dockerfile aside from the existing MASS application. Shown in Listing 3.1(c) is a Dockerfile to containerize the MASS Range Search. The first line, inherits a Docker image of MASS library and clearly defines the version of MASS library, allowing developers to work on a stable version with minimum work. Line 3 inherits a Docker image of Java. We define the Java version to 11. It might be confusing why and how we need to inherit from two Docker images. This act is to minimize the Docker image building time and to prevent hidden packages from the Docker image of MASS library. On line 9, we further define a list of required dependencies. On line 17, we require MASS developers to create a directory named “dependencies”. This directory is intended to store all files that will not change after building the Docker image of MASS application. For all the files that may change and need to access outside containers, like input data and “nodes.xml”, are required to store in a directory named “resources”. Like the first step, we need “docker-compose.yml”, as shown in Listing 3.1(d), to name the generated Docker image and to simplify the building process.

Step 4. Downloading the Docker image of MASS library from cloud:
This step is work done by Docker engine in the background when MASS developer tries to build a Docker image of MASS application with a Docker image of MASS library as foundation. The Docker engine will automatically obtain the Docker image of MASS library whether it is available in local hard disk or on cloud.

Step 5. Uploading the Docker image of MASS application to cloud:

This is straightforward with one LOI in terminal just like the second step.

Listing 3.1(a) Dockerfile of containerized MASS Java

```
FROM maven:3-openjdk-11

# build the MASS library with Hazelcast multicast discovery enabled
ENV HAZELCAST_USE_MULTICAST_DISCOVERY=true

# define the path of the JAR file will be generated
ENV MASS_JAR_PATH=/mass/target/mass-core.jar

# set working directory
WORKDIR /mass

# add all necessary files to build MASS library
COPY pom.xml .
COPY ./src ./src

# build the MASS library
RUN ["mvn", "package"]
```

Listing 3.1(b) “docker-compose.yml” of containerized MASS Java

```
version: '3.4'
services:
  massjavacore:
    image: dslabcse/mass_java_core
    build: .
```
Listing 3.1(c) Dockerfile of containerized MASS Range Search

```
1. FROM dslabcss/mass_java_core:1.3.0 as MASS
2. FROM adoptopenjdk:11-jdk
3. # install gradle for the RangeSearch compilation
4. # install ssh for connection among containers
5. # install nano as basic text editor
6. RUN apt-get update \
7.   && apt-get install gradle nano ssh -y --no-install-recomends \
8.   && apt-get clean \
9.   && rm -rf /var/lib/apt/lists/
10. # configure ssh connection among containers
11. RUN mkdir /var/run/sshd \
12.   && chmod 0755 /var/run/sshd
13. # setup public/private keys with proper permissions
14. COPY ./dependencies/key /root/.ssh/id_rsa
15. COPY ./dependencies/key.pub /root/.ssh/authorized_keys
16. RUN chmod 700 /root/.ssh/id_rsa && chmod 700 /root/.ssh/authorized_keys
17. # expose necessary ports for MASS library
18. # exposed all ports from 1025 to 65535 because Hazelcast uses random port to bind
19. EXPOSE 22
20. EXPOSE 3400
21. EXPOSE 1025-65535
22. # add necessary runtime dependency
23. COPY --from=MASS /mass /mass
24. # set the working directory in container to /app
25. WORKDIR /app
26. # add execution script
27. COPY ./run.sh  .
28. # compile the MASS Java application
29. COPY ./src ./src
30. COPY ./build.gradle .
31. RUN gradle build
32. # waiting for user to attach to the container then execute the MASS application
33. CMD ["/usr/sbin/sshd", "-D"]
```

Listing 3.1(d) “docker-compose.yml” of containerized MASS Range Search

```
1. version: '3.4'
2. services:
3.   agent:
4.     image: dslabcss/rangesearch
5.     build:
6.       context: ./MASS/
7.     deploy:
8.       replicas: 4
9.     volumes:
10.  - ./resources:/resources
```
3.2 Deploy Containerized MASS Applications

In this section, we will discuss two ways to deploy containerized MASS application. The first way is only deploying on one computing node whereas the second way is deploying on multiple computing nodes. We designed to let MASS developers just need one LOI in the terminals for deployment in both scenarios by including all the necessary information for deployment in the “docker-compose.yml” file as shown in Listing 3.1(d).

3.2.1 Deploy Containerized MASS Application on One Computing Node

Figure 3.2(a) shows a deployment to a single computing node. Only the first step requires MASS developers’ input. The second step is the work done by Docker engine in the background.

In the first step, a MASS developer invokes Docker-Compose to build containers with the application’s “docker-compose.yml”. For example, if we use the file from Listing 3.1(d), we can define the number of containers that will be populated by Docker engine on line 9. This is the number of replicas. In addition, we can define places to store persisting data generated by and used by Docker containers, as volumes, on line 11.

In the second step, while Docker-Compose tries to invoke Docker engine to create Docker containers, docker engine will fetch MASS application’s Docker image from the local hard disk.
3.2.2 Deploy Containerized MASS Application on Multiple Computing Nodes

If we want to deploy on multiple computing nodes, that has the Docker image in the cloud, it only requires 1 step, shown as step 1 in Figure 3.2(b).

In the first step, a MASS developer invokes Docker Swarm to build containers with the application’s “docker-compose.yml” across multiple computing nodes. Similar to deploying with only one node, Docker Swarm will fetch the number of replicas, in the “docker-compose.yml” file, to populate same number of containers.
The second and third steps are work done by Docker Swarm in the background. While the master node in Docker Swarm instructs worker nodes to construct Docker containers, worker nodes will try to fetch MASS application’s Docker image from cloud and to launch containers.

Figure 3.2(b) Procedure to deploy a containerized MASS application on a computing cluster
3.3 Execute Containerized MASS Applications

In this section we will discuss two ways to connect to Docker containers and execute the program. The first way is connecting to containers deployed on one node whereas the second way is connecting to containers deployed on multiple nodes.

3.3.1 How to Execute Containerized Application

MASS developers have to set up a configuration file before executing their MASS applications. The configuration file, “nodes.xml”, is required due to the design of the MASS library. With containerization support in place, we can use a script to retrieve all required data, including the hostname, the path to java executable, the path to the generated MASS application, the name of an authenticated user, and a private key that can authenticate the user. Although current implementation forces MASS developers to install Python with version higher than 3.4, Python3.4 is easily to install. Also, if Python3.4 is not available on the machine, this script could be transformed to other languages, such as Java and Go. In addition, this script provides verbose guidance on each step that requires MASS developers to input.

3.3.1 Execute Containerized MASS application on One Computing Node

MASS developers are required to input twice to obtain the correct “nodes.xml” file. The first choice is either “y” or “n”. In the example used in Figure 3.3(a), the developer should input “n” as it is executing with only one computing node. The second choice is depending on name of
the MASS application’s Docker image. As shown in Figure 3.3(a), we input “5”, which is the index number of “rangesearch_default”.

3.3.2 Execute Containerized MASS application on Multiple Computing Nodes

Similar to executing application on one node, we need to generate the “nodes.xml” file using a Python script, before connecting to the container, and executing the run script.

As shown in Figure 3.3(b), MASS developers are required to input three times to obtain the correct “nodes.xml” file. The first choice is either “y” or “n” and should input “y” since we are executing on multiple nodes. The second and third choices are depending on name of the MASS application’s Docker image and we input “6” and “0”, as shown in Figure 3.3(b).
Figure 3.3(a) Generate “nodes.xml” for execution on One Computing Node

Figure 3.3(b) Generate “nodes.xml” for execution on Multiple Computing Nodes
3.4 Streamlined CI/CD Experience

3.4.1 Original Workflow

In the original workflow, MASS developers are required to do everything manually before and after committing code to the master branch in git repository. As shown in Figure 3.4(a), they need to execute unit tests locally, to conduct some integration tests with existing MASS applications, and to upload the generated MASS library for a release.

![Figure 3.4(a) Original Workflow](image)

3.4.2 New Workflow with CI/CD Pipeline

We chose to add CI/CD pipeline through Bitbucket Pipelines since all code repositories that hosts the MASS library and MASS applications are situated on Bitbucket. This would present
least amount of effort for future MASS developers to extend and modify the behavior of each CI/CD pipelines.

To add CI/CD pipeline support, we added a configuration file, “bitbucket-pipelines.yml”, to each code repository that hosts MASS library and MASS applications. Content in each configuration file has to follow the syntax as required by Bitbucket Pipelines. Although content in each configuration file differs, procedures of pipeline are similar. There are four steps as shown in Figure 3.4(b). But steps 2, 3 and 4 are all automated after MASS developer committing code to git repository in the first step.

The first step is straightforward. It is MASS developer committing code to git repository on developer’s feature branch, but not master or develop branch.

The second step will be automatically triggered by code committing from the first step. The behavior of CI/CD pipeline differs according to the content of the “bitbucket-pipelines.yml” file and branch of the new committed code. In current implementation, code committed to any branch will trigger the pipeline start building the updated MASS library or MASS applications, where the building tool will execute all enabled unit tests. After successfully building and testing the generated files, a code linter will examine the content of each Dockerfile.

Only if code is committed to either master or develop branch, integration tests will be triggered since Bitbucket has limit on amount of pipeline computation. These integration tests allow MASS developers to verify correctness of the MASS library and MASS application while executing with multiple nodes in a parallel scenario.
If pipelined execution failed for any reason, either while compiling the code, executing unit tests, or executing integration tests, the third step will be triggered. In this step, the pipeline records this commit with a red exclamation mark as well as automatically sends an email to the MASS developer, informing errors and where they happened. The red exclamation mark and a sample email can be accessed in Appendix A. This feedback loop helps MASS developers acknowledge the faulty code and prevents MASS developers from merging faulty code into the develop or master branch.

If the pipelined execution completes in success, the fourth step will be automatically triggered. In this step, the pipeline will upload the Docker image of the MASS library or MASS application to the cloud and release to public. And MASS developer can read a green check mark behind their commit message as shown in Appendix A.

![Figure 3.4 (b) New Workflow with CI/CD Pipeline](image)
Chapter 4. Evaluation of MASS containerization

This chapter presents the evaluation environment and evaluates the design quality of containerization support for the MASS library and its applications as well as CI/CD pipeline.

4.1 Evaluation Environment and Procedures

4.1.1 Input Data Format

The input dataset for each application varies depending on how the application was implemented before adding containerization support. Range Search requires a text file, containing a list of 2D points, and five arguments, defining the number of points inside the text file and an enclosed area to search. TB simulation requires two arguments, including the number of simulation cycles and the size of a square grid. Shortest Path requires three arguments, including the total number of nodes in a graph, the source vertex id, and destination vertex id.

4.1.2 Evaluation Criteria

The evaluation on the design quality of containerization support for the MASS library and its applications as well as CI/CD pipeline is based on (International Organization for Standardization/ International Electrotechnical Commission (ISO/IEC) 25010 standard. Only five out of eight metrics were selected, including efficiency, usability, reliability, maintainability, and portability. We do not include the other three criteria, including functionality, compatibility, and
security, since the containerized MASS does not differ from the original MASS on these three attributes.

4.1.3 Evaluation Environment

Execution performance was evaluated on four Linux clusters, which have similar specification. Each has an eight-core Intel Xeon E5410 CPU with sixteen gigabytes of memory. Execution time is an averaged value taken from at least three independent executions.

4.2 Evaluation on Software Quality

This section divides into five sub-sections, each covering one aspect of design quality, including reliability, usability, efficiency, maintainability, and portability. Each section includes analysis on difference between original MASS and containerized MASS as well as a survey collected from MASS developers in Distributed Systems Laboratory.

Two out of the 11 MASS developers answered the survey. The survey is consisting of seven questions. Each question requires participants to give a score, from one-to-five, and to provide a comment, optionally, for comparison between the original MASS and the containerized MASS. The seven questions includes five questions on design quality of containerization, an overall comparison between the original MASS and the containerized MASS, and a comparison between developing experience with and without CI/CD pipeline.
4.2.1 Evaluation on Efficiency

ISO/IEC 25010 defines efficiency as “characteristic represents the performance relative to the amount of resources used under stated conditions”. [20] This subsection focuses on the execution time for three MASS applications, including (1) Range Search, (2) Shortest Path, and (3) TB simulation, as well as (4) feedback from survey.

(1) As shown in Figure 4.2(a), execution of Range Search with MASS Java on both bare metal and containers demonstrates performance increase with more computing nodes when input file has 100k points. Execution on containers have more overhead as expected but at a low ratio comparing to the total execution time. Docker’s additional execution time with one, two, and four computing nodes is 2.5%, 5.9%, and 9.4% respectively. Also, we benchmarked execution of Range Search when input file has 100 or 10k points. Executions with 100 or 10k points demonstrate performance decrease with more computing nodes, in which total computation time was affected more by communication overheads than actual computation.

(2) We benchmarked execution of MASS Shortest Path in MASS Java, with 3000 points and only one node since original implementation could not execute on multiple nodes. The performance was lowered by 5.1%.

(3) TB simulation is the only application we containerized with MASS C++. We benchmarked TB simulation with 30 iterations, on a 5x5 grid, and with only one node since original implementation could not execute on multiple nodes. The performance was lowered by 22%. We conclude the irregular performance loss comes from overly small dataset while this is the only input we could verify correctness at the moment. The computation time for the TB
simulation is much shorter than Range Search. TB simulation averaged to take 94 seconds, for orignial MASS, and 115 seconds, for containreized MASS. In contrast, Range Search averaged to take 396, for orginal MASS, and 406 seconds, for containreized MASS. Furthermore, the execution time for MASS TB simulatiuon is quite unstable. It has standard deviation of 43.3, for orignal MASS, and 60, for containerized MASS. In contrast, the computation times for MASS RangeSearch is stable. It has standard deviation of 8.3, for orignal MASS, and 5.3, for containerized MASS.

(4) In the survey, both MASS developers think containerized MASS does better than original MASS on efficiency. One developer gives four, for containerized version, two, for original version. The other MASS developer gives five, for containerized version, four, for original version.

Figure 4.2(a) Execution performance of Range Search
4.2.2 Evaluation on Usability

ISO/IEC 25010 defines usability as “degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use”. [20] This subsection focuses on different amount of effort to use the original and containerized MASS library and MASS applications with CI/CD pipeline from six perspectives in a software development cycle. The six perspectives include (1) amount of guidance, (2) amount of effort to build, (3) amount of effort to execute, (4) amount of effort to verify, and (5) amount of effort to release the MASS library and MASS applications as well as (6) feedback through survey.

(1) Containerized MASS applications have more trackable documentation and easier to set up comparing to the original MASS applications. Taking MASS Range Search as an example, the original MASS Range Search has zero Line Of Code (LOC) describing the proper way to set up the environment and to launch itself. In contrast, containerized Range Search provides a dedicated wiki page with detailed description on each Line Of Inputs (LOI) required for MASS developer to execute applications. Furthermore, the wiki page can be copy-and-paste across different containerized MASS applications since containerized MASS applications share the same commands to build, to launch, and to execute.

(2) Building process for the containerized MASS Java library or MASS C++ library has been greatly simplified. To build the original MASS Java library and MASS C++ library, developers need to install the same version of dependencies across all machines and platforms for maximizing consistency and minimizing potential errors. For MASS Java, the environment requires Java
version higher than 11 and maven version higher than 3. For MASS C++, the environment requires gcc version higher than 10 and libssh higher than 1.4.3. Without containerization, MASS developers had to manually install dependencies and update them across machine if MASS deprecates support for old dependencies. In contrast, developers just need to guarantee Docker version higher than 17.09 has been installed across all computing nodes.

(3) Containerized MASS application requires much less steps to execute. As shown in Table 4.2(b), containerized MASS Range Search always requires less the LOI than the original version. When Range Search execute on one node, containerized version requires less than half amount of LOI. Furthermore, original version of Range Search requires more than five times of LOI that depends on environment. This LOI includes data that needs to look up by MASS developers, such as path to generated executable. Notably, when containerized MASS application is deploying onto multiple nodes, LOI does not change depending on the number of computing nodes involved. In contrast, LOI for the original MASS Range Search showed an abrupt increase mainly due to cumbersome configuration file, “nodes.xml”.

(4) Pipeline verifies the correctness of existing code with unit tests and integration tests before a commitment. With comprehensive and robust unit tests and integration tests in place, MASS developers can worry less about bugs hidden inside the MASS library.

(5) Pipeline automatically publishes tested Docker image of the MASS library and MASS applications to the DockerHub. This allows MASS developers to directly interact the pre-built and tested MASS library and MASS applications, which minimizes human errors in environment setup.
In the survey, both MASS developers think containerized MASS does better than original MASS on usability. One developer gives four, for containerized version, three, for original version. The other MASS developer gives five, for containerized version, three, for original version. In addition, one MASS developer comments that running program created by Docker images is way easy than the original MASS. Though this comment was incorrectly commented by the participant in the evaluation on efficiency section.

<table>
<thead>
<tr>
<th>Scenarios</th>
<th>Metrics</th>
<th>Original</th>
<th>Containerized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute MASS Range Search on 1 node</td>
<td>LOI</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>LOI depends on environment</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>Execute MASS Range Search on a cluster with 4 nodes</td>
<td>LOI</td>
<td></td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>LOI depends on environment</td>
<td></td>
<td>33</td>
</tr>
<tr>
<td>Execute MASS Range Search on a cluster with 8 nodes</td>
<td>LOI</td>
<td></td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>LOI depends on environment</td>
<td></td>
<td>65</td>
</tr>
</tbody>
</table>

4.2.3 Evaluation on Reliability

ISO/IEC 25010 defines reliability as “degree to which a system, product or component performs specified functions under specified conditions for a specified period of time”. [20] This subsection focuses on the difference of system status when one or more computing nodes stop working in a cluster.

Docker Swarm provides a reliable environment for the containerized MASS executing on multiple nodes by automatically arranging containers. When a container crashes for any reason on one computing node, the swarm manager tries to construct a new container to replace the
crashed container. Figure 4.2(b) demonstrates a recovered container. First, we confirm four containers were working normally. Then we intentionally terminate the third container to simulate a crashed computing node. When examining the service status, Figure 4.2(b) shows the third container was re-constructed, launched, and ready for MASS computation after “nodes.xml” getting updated. In contrast, original MASS applications could not revive a crashed computing node and MASS developers have to update “nodes.xml” manually.

In the survey, two MASS developers do not notice any difference on reliability between the containerized and the original MASS. They gave both the original MASS and the containerized MASS a same score, three out of five.

<table>
<thead>
<tr>
<th>Container ID</th>
<th>Name</th>
<th>Desired State</th>
<th>Current State</th>
<th>Error</th>
<th>Ports</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RangeSearch Cluster agent 1</td>
<td>Running</td>
<td>Running</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>RangeSearch Cluster agent 2</td>
<td>Running</td>
<td>Running</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>RangeSearch Cluster agent 3</td>
<td>Running</td>
<td>Running</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>RangeSearch Cluster agent 4</td>
<td>Running</td>
<td>Running</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>Docker service as RangeSearch Cluster agent</td>
<td>Running</td>
<td>Running</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.2(b) Downed and automatically revived container

4.2.4 Evaluation on Maintainability

ISO/IEC 25010 defines maintainability as “This characteristic represents the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements.”. [20] This subsection focuses on different amount of effort to add new features to the original and containerized MASS library and applications with CI/CD pipeline from five perspective. The five perspectives include
(1) amount of effort to introduce new dependencies, (2) amount of effort to add new features, (3) amount of effort to verify functionalities, and (4) amount of effort to verify correctness as well as (5) feedback through survey.

(1) If a MASS developer wants to introduce a new dependency to the MASS, all MASS developers have to manually install new dependencies on all machines that needs to execute the MASS. In contrast, containerization support wraps all required dependencies inside and always requires same two dependencies, Docker and Docker-Compose. In addition, the containerized MASS library has consistent versions for all dependencies, which reduces potential errors (e.g., inconsistent Java installation across computing nodes) and time to debugging errors.

(2) To compare different amount of effort to add new features to the MASS, a comparison on time to add a new feature would be ideal. However, the time is hard to measure since developers are reluctant to accomplish the same goal twice, the comparison would not be fair for the first feature being developed, and skill for developers varies greatly. Hence, we selected LOC to quantify the amount of effort. Although containerized MASS applications always have more LOC than original MASS applications due to additional files for containerization support, containerized MASS applications reduce the LOC that requires manual inputs, which dominates amount of time to add a new feature for inexperienced MASS developers. As shown in Table 4.2(a), to build the MASS Shortest Path from scratch, we need 444 LOC and containerized version requires 58 more LOC. However, most contents in these LOC can simply copy-and-paste from existing containerized MASS applications since all MASS applications share the same building procedure. In combination with the fact that containerized MASS applications inherit the containerized MASS library, MASS library and containerized MASS Shortest Path has less LOC
requires manual input. MASS developers do not need to look up environment variables, such as path to Java and generated library, to manually input them into build-script and to run-script anymore. Similarly, building containerized MASS Range Search requires 58 more LOC but only has one more LOC requires manual input.

(3) The MASS is a parallel-computing library that requires MASS applications to deploy on multiple computing nodes to verify functionalities. Containerized MASS applications allow MASS developer to use one machine to simulate execution on multiple nodes. In contrast, MASS developers have to set up environment for original MASS applications on all computing nodes.

(4) To verify correctness of original MASS applications, MASS developers have to manually conduct unit tests and integration tests for every new commit they contribute. In contrast, CI/CD pipeline frees MASS developers from repetitive manual testing. Once unit tests and integration tests are appended into the CI/CD pipeline. Every future code commit will automatically include these tests, which prevents code smell [21]. In addition, CI/CD pipeline automatically notifies MASS developers the status of their code commits. The notification expedites the process for MASS developers to fix corrupted code.

(5) In the survey, both MASS developers think containerized MASS does better than original MASS on maintainability. One developer gives four, for containerized version, three, for original version. The other MASS developer gives five, for containerized version, four, for original version. Also, one MASS developer comments that the steps to build containerized MASS applications are very intuitive.
Table 4.2(b) Maintainability comparison between Original and Containerized Range Search

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Metrics</th>
<th>Original</th>
<th>Containerized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build MASS Shortest Path</td>
<td>LOC</td>
<td>444</td>
<td>502</td>
</tr>
<tr>
<td></td>
<td>LOC requires manual input</td>
<td>416</td>
<td>414</td>
</tr>
<tr>
<td>Build MASS Range Search</td>
<td>LOC</td>
<td>762</td>
<td>820</td>
</tr>
<tr>
<td></td>
<td>LOC requires manual input</td>
<td>732</td>
<td>733</td>
</tr>
</tbody>
</table>

4.2.5 Evaluation on Portability

ISO/IEC 25010 defines portability as “degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another”. [19] This subsection focuses on ability for the MASS to execute on different platforms.

Docker works as an abstraction layer between application and operating system, supporting execution across platforms. Although we are not demonstrating Docker’s capability of deploying applications across different architecture such as x86 and ARM, we demonstrate Docker’s capability of deploying applications across different operating system in Appendix B. MASS Range Search could execute on a Windows and MASS Multiplane Search could execute on an Apple Mac with Docker installed. In contrast, original MASS requires MASS developers to install every dependency, which requires them to find pre-built executable or compile by themselves for every dependency.

In the survey, both MASS developers think containerized MASS does better than original MASS on portability. One developer gives four, for containerized version, two, for original version. The other MASS developer gives five, for containerized version, four, for original version.


### 4.3 Summary of Evaluation

Overall, by comparing with the current MASS library and MASS applications, the containerized MASS library versions and its applications greatly improve usability and maintainability. Although the containerized MASS and its applications sacrifice efficiency by around five percent, it is negligible to MASS developers.

In the survey, both MASS developers think containerized MASS does better than original MASS overall. One developer gives four, for containerized version, three, for original version. The other MASS developer gives five, for containerized version, three, for original version.

**Strengths and challenges**

The strengths and challenges of the MASS containerization are summarized below from the three viewpoints: library containerization, application containerization, and CI/CD pipeline:

1. **Library containerization**

   **Strengths:** The containerized MASS library packages up all required dependencies into a container, providing a consistent environment and easing the effort to be reused in containerized MASS application and CI/CD pipeline.

   **Challenges:** The generated executables from containerized MASS C++ may not usable outside containers since compiled C++ files are environment specific. Making it hard for MASS developers to prepare a same environment for comparison between execution performance on bare metal and within containers.

2. **Application containerization**
Strengths: The containerized MASS application simplifies procedure to execute.

Challenges: Since containerization adds overhead before execution, MASS applications suffer around 2.5% to 10% of performance loss. In addition, to maximize the benefit of containerized MASS applications, MASS developers need to learn basic knowledge about containerization.

(3) CI/CD Pipeline

Strengths: The CI/CD pipeline enables a feedback loop to notify MASS developers of the correctness of their code commits and enables automated testing and code release.

Challenges: If the pipeline changes the coding syntax, MASS developers may have to modify the configuration files accordingly. In addition, if a MASS developer commits corrupted code to the develop or master branch and another MASS developer pulls from that branch, the following MASS developer could receive warning from email and misunderstand his code is corrupt.
Chapter 5. Conclusion

The project completed its goals through the implementation of the following four implementation tasks: (1) containerizing two versions of MASS libraries, (2) containerizing three sample MASS applications, (3) adding CI/CD pipeline to the code repository for the MASS libraries and MASS applications, (4) and verifying the implementation of the containerized MASS library and its applications and compared against current MASS from multiple aspects.

Below we discuss the potential areas where future work can be done based on the outcome of this work.

1. Security: MASS developers are sharing the same access permission when they try to deploy their containerized MASS applications to a cluster system. It could arouse a potential hazard if a MASS developer removes another MASS developer’s container. Singularity may provide a potential solution, but we need further research.

2. Diversity: Containerization serves for the MASS library and its applications as an exceptionally reliable platform, which covers up some corrupt code only appear in inconsistent scenarios, such as drastically different datetime. To increase the robustness and correctness of the containerized MASS, it is beneficial to extend tests with more diversity.

3. Containerization file generation: Although MASS developers just need to copy-and-paste containerization files, including Dockerfile, from their previously containerized applications to a new MASS application with a few lines of edit, it would be nice to present them a script that generates containerization files with verbose guidance.
4. Integration test generation: MASS developers need to combine input files, source code of integration test, and expected output for each additional integration test. If MASS developers can coordinate the testing procedure, we may have a better experience adding integrations tests.

5. Debugger: MASS developers are using debuggers to pinpoint errors and the current containerization presents an abstraction layer as barrier between debugger and execution process. Visual Studio Code [22] may already presents a solution, but we need further investigation.

6. Robustness: Comparing to Kubernetes, Docker Swarm is not an industrial-level production-ready tool for deployment. If MASS needs to deploy with highest standard, we may need to investigate Kubernetes or other solutions.
[16] T. Eilam, M. H. Kalantar, A. V. Konstantinou, G. Pacifici, J. Pershing, and A. Agrawal, “Managing the configuration complexity of distributed applications in Internet data


Appendix A

Succeed and failed commits on Bitbucket Pipelines

- Jiashun Gou 7568c35 4 branches 2021-04-22
- Jiashun Gou 7993f1b 4 branches 2021-04-22

Received email of a code commit failed to build
Appendix B

Containerized MASS Range Search Execute on Windows

Containerized MASS Multiplane Search Execute on Apple Mac