



南京大學

本科畢業論文

院 系 _____ 計算機科學與技術系
專 業 _____ 計算機科學與技術
題 目 _____ 基於 Mesos 的 MPI 容器化封裝技術研究
年 級 _____ 四 _____ 學 號 _____ 141220074
學生姓名 _____ 路萍
指導教師 _____ 曹春 _____ 職 稱 _____ 副教授
提交日期 _____ 2018 年 6 月

南京大学本科生毕业论文（设计、作品）中文摘要

题目：基于 Mesos 的 MPI 容器化封装技术研究

院系：计算机科学与技术系

专业：计算机科学与技术

本科生姓名：路萍

指导教师（姓名、职称）：曹春 副教授

摘要：

MPI（消息传递接口）是在 HPC（高性能计算环境）中开发并行应用的标准通讯协议。由于它具备较好的可移植性与较高的性能，所以时下仍是高性能计算的主要模型。容器技术属于操作系统级虚拟化技术的一种，由于它非常轻量级、也易于部署，所以可以大幅度降低系统中的资源占用。作为分布式系统的重要组成部分，以 Docker 为代表的容器管理工具迅速成为学术界和工业界的研究热点。Mesos 是一种开源分布式资源管理框架，它通过抽象集群中各主机的计算资源来搭建一套具有高效性、高容错性的弹性分布式系统。伴随着容器技术的崛起，Mesos 开始支持 Docker，国内外的许多开发者也纷纷尝试使用 Mesos 管理集群。

但由于 MPI 在版本以及厂商之间存在较大的兼容性差异，所以在 HPC 中开发部署 MPI 程序是一个具有挑战性的问题。另外，虽然时下有多种资源管理平台，但是提高资源利用率依然是一个需要考虑的问题。针对这两个问题，本文主要讨论并研究了如何使用 Docker 容器在 Mesos 平台中部署运行 MPI 任务，并将此方案与不使用 Docker 部署进行比较，以证明其可行性。

本文的具体工作包括：

- 针对使用容器部署分布式并行计算任务的需求，选用 Docker 容器技术，以镜像封装任务运行时所需的环境。
- 针对在集群中运行 MPI 任务的需求，选用 Mesos 作为分布式资源管理框架，通过自定义 Scheduler 对资源进行合理分配。
- 在以上两点研究的基础上，结合前期的参照实验，将本文提出的解决方案与现有概念进行对比，从不同角度验证该解决方案的可行性。

关键词：容器；分布式系统；并行计算；资源管理

南京大学本科生毕业论文（设计、作品）英文摘要

THESIS: Research on Technologies for Mesos Based Containerization of MPI

DEPARTMENT: Computer Science and Technology

SPECIALIZATION: Computer Science and Technology

UNDERGRADUATE: Lu Ping

MENTOR: Associate Professor Cao Chun

ABSTRACT:

MPI (Message Passing Interface) is a standard communication protocol developed for parallel applications in the HPC (High Performance Computing Environment). Because it has good portability and high performance, it is still the main model of high-performance computing. Container technology is a kind of operating system-level virtualization technology. Because it is very lightweight and easy to deploy, it can significantly reduce resource consumption in the system. As an important part of distributed systems, container management tools represented by Docker have quickly become a research hotspot in academia and industry. Mesos is an open source distributed resource management framework. It abstracts the computing resources of each host in the cluster to build an elastic distributed system with high efficiency and high fault tolerance. With the rise of container technology, Mesos began to support Docker. Many developers at home and abroad have also tried Mesos management clusters.

However, due to the large differences in compatibility between MPI versions and vendors, developing and deploying MPI programs in HPC is a challenging issue. In addition, although there are many kinds of resource management platforms nowadays, improving resource utilization is still a problem that needs to be considered. For these two problems, this article mainly discusses and studies how to use the Docker container to deploy and run the MPI task on the Mesos platform, and compare this solution with the use of the Docker deployment to prove its feasibility.

The specific work of this article includes:

- For the need to deploy distributed parallel computing tasks using containers,

use Docker container technology to encapsulate the environment required for task execution.

- For the requirement to run MPI tasks in the cluster, Mesos is selected as the distributed resource management framework, and the scheduler is used to rationally allocate resources.
- On the basis of the above two points of research, combined with previous reference experiments, the proposed solution is compared with existing concepts, and the feasibility of the solution is verified from different perspectives.

KEY WORDS: Container, Distributed System, Parallel Computing, Resource Management

目录

第一章 绪论.....	3
1.1 研究背景.....	3
1.2 研究现状.....	4
1.3 本文工作.....	5
1.4 本文组织.....	5
第二章 相关工作和技术.....	7
2.1 MPI 消息传递接口.....	7
2.2 Docker 容器技术.....	8
2.3 Mesos 资源管理框架.....	10
2.3.1 Scheduler 调度器.....	14
2.3.2 Executor 执行器.....	17
2.4 本章小结.....	18
第三章 基于 Mesos 的 MPI 容器化运行研究.....	19
3.1 方案整体描述.....	19
3.2 MPI 的容器化封装.....	20
3.2.1 使用 Docker 制作镜像.....	20
3.2.2 容器间跨主机通信.....	23
3.3 使用 Mesos 进行资源管理.....	26
3.3.1 NFS 网络文件共享系统.....	26
3.3.2 自定义 Mesos Scheduler.....	29
3.4 本章小结.....	35
第四章 对比验证与实验.....	36
4.1 不使用 Docker 部署 MPI 任务.....	36
4.1.1 集群环境.....	36
4.1.2 对照实验.....	36
4.2 实验结论.....	38
4.3 本章小结.....	38

第五章 总结与展望.....	39
5.1 工作总结.....	39
5.2 研究与展望.....	39
参考文献.....	I
致谢.....	IV

第一章 绪论

1.1 研究背景

MPI[1]是消息传递编程模型之一，主要定义了消息传递应用程序的接口，以及语义说明和协议。在实际场景中，我们使用的是 MPI 的某个具体实现，比如 openmpi、mpich 等，为了简化，在本文中统一称为 MPI。消息传递方式广泛应用于各种并行机，尤其是分布式存储并行机，尽管各个厂商和版本的 MPI 在具体实现上有些许不同，但进程间通过消息传递进行通信的基本概念是相似的。MPI 通过在核心库里定义程序的语法和语义，可以有效和可移植地实现一个消息传递系统。自从 MPI 产生以来，该类消息传递编程模型在计算应用领域里获得了实质性的进步。

因为 MPI 有针对高性能通信作出良好的优化，所以在高性能计算环境中它几乎是标配。但是，MPI 的主流版本（比如 Openmpi，mpich，Intel MPI）大小都不小，而且由于厂商和版本的差异，在集群的不同主机中配置起来会比较麻烦，甚至如果支持 MPI 会使原本轻型的框架系统变得冗余沉重。

因此，在本文的研究中，首先考虑采用时下流行的 Docker 技术，对 MPI 进行封装，以期解决在集群中部署 MPI 产生的版本适配问题。

容器技术是一种操作系统级虚拟化技术。Docker[2]是容器管理工具中的翘楚，在 2013 年开源。Docker 容器技术具有轻量化的显著特性[3][4]。一方面，启动一个 Docker 容器的耗时，与在普通操作系统中启动单个进程的耗时大致相当；另一方面，由于 Docker 容器和宿主机共享操作系统内核，所以其镜像只需要包括应用以及应用运行需要的依赖[5]。虽然 Docker 技术不能像虚拟机技术那样支持带有不同操作系统的宿主机，但它的轻量化有非常显著的优势，不论是在开发、测试还是最后的生产环境中都可以平滑顺畅地运行[6]。因为使用 Docker 容器就不再需要单独为客户机安装操作系统，可以直接使用宿主机的操作系统内核，并获取宿主机操作系统内核提供的资源管理机制等一系列运行支撑。

伴随着 Docker 技术的成熟，Docker、Google、Mesosphere 等公司都推出了各自开发的容器计算平台。诸如 Docker 公司自己推出的原生容器集群编排管理

工具 Docker Swarm[7], Google 公司开发的名为 Kubernetes[8]的容器计算平台, 以及 Mesosphere 基于 Mesos 资源管理器开发的 marathon[9]框架等。

在这多种集群编排管理框架之中, Mesos 可以达到框架混布的目的, 并由此灵活调度资源, 提高利用率。所以本文研究工作中, 将依靠 Mesos 平台, 来对资源进行管理调度。

Mesos[10]是由 Mesosphere 开发的一种用于管理集群资源的工具, 现在被 Apache 收购, 并将之开源。起初, Mesos 并不是为了适应 Docker 而产生的, 甚至还拥有自己的原生容器隔离。它诞生的初衷是为 Spark 做集群资源的管理。但是一方面, 随着 Docker 技术的不断崛起, Mesos 也顺势而为, 从 0.20 版本开始为 Docker 容器提供支持。这样, 由 Mesos 与 Docker 强强联手合成的平台, 可以为在集群环境中部署应用和服务的场景创造强大的支撑, 也带来了莫大的便利。另一方面, Google 发了一篇比较 Mesos 与 Google 内部的 borg 和 Yarn 三个集群资源管理平台性能的论文, 并指出 Mesos 和 borg 有着异曲同工之妙[11][12]。通过这两方面的帮助, Mesos 在工业界被越来越多地投入到了生产实践中去。例如国外的世界级大科技公司 Apple、Twitter、Uber 等, 还有国内知名的公司诸如豆瓣网、爱奇艺、去哪儿网、360、数人科技等, 都早已开始使用 Mesos 平台做资源管理了。

1.2 研究现状

传统方式部署 MPI 应用时, 往往会要求集群中对 MPI 的配置具备高度一致性。一旦生产环境变得庞大, 那么对 MPI 的版本更新就会变慢, 常常会出现牵一发而动全身的情况。

结合容器技术的便利性, 可以考虑使用容器封装 MPI 任务运行环境。一般在使用 Docker 容器部署应用时, 大部分都是单宿主机, 很少有类似承载 MPI 任务的多宿主机的集群系统环境。但是, 关于使用 Docker 容器在集群中运行任务的优点也已经有相关研究证明了[13]。

另外, 在 MPI 并行计算、容器技术和各种分布式资源管理框架浪潮迭起的大环境下, 涌现出一批多式多样的资源管理框架, 它们可以针对资源进行切割、分配、管理, 比如 Docker Swarm Mode、Mesos、Slurm、yarn 等。

目前已经有研究基于 Docker Swarm Mode 实现的使用 Docker 部署运行 MPI 任务[14]。但是，想要实现框架混布，还需要采用 Mesos 平台进行调度。

1.3 本文工作

由于 MPI 在版本以及厂商之间存在较大的兼容性差异，所以在高性能计算集群中开发部署 MPI 程序是一个具有挑战性的问题。其次，MPI 的主流版本大小都不小，在轻量级系统中如果支持 MPI，可能会破坏原有的设计理念，使整个框架系统都变臃肿。

另外，针对 Mesos 资源管理框架，如何对资源进行灵活调度，使得运行在集群上的所有框架与应用可以根据需要，相互隔离地交错运行，也是本文需要考虑的问题。针对这些问题，本文主要研究了使用 Docker 容器在 Mesos 平台中部署 MPI 任务的解决方案，以期对以上不足做出改进。

本文完成的主要工作有以下几点：

- 针对使用容器部署分布式并行计算任务的需求，选用 Docker 容器技术，以镜像封装任务运行时所需的环境。
- 针对在集群中运行 MPI 任务的需求，选用 Mesos 作为分布式资源管理框架，通过自定义 Scheduler 对资源进行分割、调度和管理等操作[15]。
- 在以上两点研究的基础上，结合前期的参照实验，将本文提出的解决方案与现有概念进行对比，从不同角度验证该解决方案的可行性。

1.4 本文组织

本文的各个章节内容安排如下：

第一章绪论，概括性地介绍了 MPI 的特性、Docker 容器的优点以及 Mesos 资源管理框架的应用现状，简明阐述了本文主要工作和内容结构。

第二章详细介绍了本文研究中做过的相关工作和涉及的关联技术。首先介绍了 MPI 并行编程标准，以及它在实际应用场景中的不足之处；然后通过与虚拟技术的比较，总结了 Docker 技术的优势以及对改进 MPI 在实际应用中不足的可适用性；最后使用 Mesos 平台做资源管理和调度，以容器为资源分割的单位进行调度管理，并提出使用 Mesos 管理资源的优势与适用性。

第三章主要阐述本文的研究方案。针对两方面问题：一是 **MPI** 在集群间的版本适配问题；二是采用何种资源管理框架才能实现任务混布。提出了基于 **Mesos** 平台对 **MPI** 进行容器化封装的解决方案。并从容器镜像、共享存储、网络结构、资源调度策略等方面来介绍本文的工作内容。

第四章进行对比实验与验证。将不使用 **Docker** 部署与本文的方案进行对比，验证本方案具有的一定有效性和可行性。

第五章总结了本文的研究工作，分析本文目前的不足之处，并展望进一步的完善方向。

第二章 相关工作和技术

2.1 MPI 消息传递接口

在高性能计算机集群中，为了达到各主机间能够有效地通信、协调计算机集群的目的，如表 2-1 中所示，现在已经有很多种并行编程的模型。其中一种流行的模式就是名叫 **Message Passing** 的消息传递并行编程模型。这个模型中又有很多套规范，如图 2-1 所示。因为 MPI（**message passing interface**）的易用性，目前在 HPC 中应用最广泛的就是它。用户在编写程序的时候，只需要学习 MPI 定义的一些标准接口，然后根据需要自行去实现标准库中的函数，而不用花时间和精力去关心消息传递这件事的本身是如何实现的。

表 2-1：并行编程模型

特征	消息传递	共享存储	数据并行
典型代表	MPI, PVM	OpenMP, Pthreads, Click	进程级细粒度
并行粒度	进程级大粒度	线程级细粒度	进程级细粒度
数据存储模式	分布式存储	共享存储	共享存储
学习入门难度	较难	容易	偏易
可扩展性	好	差	一半

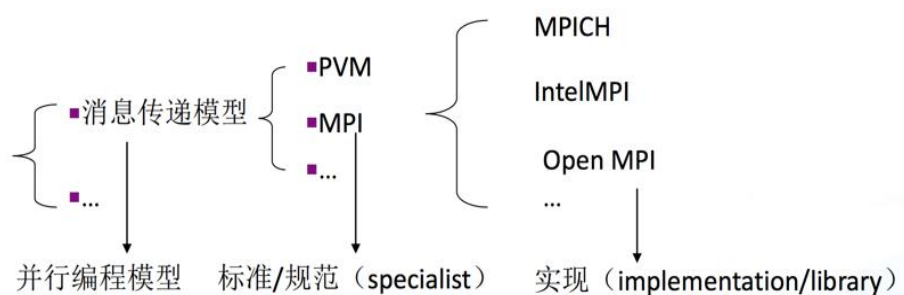


图 2-1：消息传递模型

MPI 作为一个可以跨语言的通讯协议，适用于并行计算的编程。通过 MPI 标准化定义的消息传递接口，能够给生产商提供定义清晰的程序库，以便他们高效地实现这些库里的函数，或者为库程序提供一些硬件支持，这样一来，在易于使用的同时还加强了 MPI 的可扩展性。同时，共享存储也可以被实现。组合共享和分布式存储体系结构的流行，使这种模式不会过时，反而可以在大范围的计算机集群中使用。除此之外，建立消息传递标准的优点还有它显而易见的可移植性，特别是在以消息传递为基础的应用所构成的分布式集群环境中，对消息传递接口进行标准化的效益极其显著。

综上所述，可以总结归纳为，MPI 是以为编写消息传递应用程序而开发的、并被广泛使用的标准。它同时具有易用性、可扩展性、可移植性、和有效性等优势[16]。

2.2 Docker 容器技术

Docker 是当前最具有代表性的开源容器管理工具，通过创建 Docker 镜像等方式，它可以为任何一个应用创建出又轻量又高度可移植的 Docker 容器。Docker 容器技术的思想源自于集装箱[17]。举个例子，在一艘正在海洋里航行的大型货船上，如果可以把各种各样的货物按照类别分开规整地放在集装箱里，而且保证集装箱和集装箱之间互不影响，那么就不需要再为不同类的货物准备不同的船只。只要这些货物完全被封装在集装箱内，我们就可以使用一艘大船把这些装有不同货物的集装箱全部运走。Docker 的思想就可以照此例来理解，时下兴起的云计算平台就相当于一艘大货轮，Docker 就对应于货轮上的集装箱。

与虚拟机技术重叠的是，Docker 旨在为用户提供独立的包括进程空间、文件系统和网络栈等运行环境。但和虚拟机技术的不同之处在于服务器的负载方面，单独开一个虚拟机会占用服务器中的空闲内存，但假设换成使用 Docker 容器来部署应用，这些内存就会被重新利用起来；在环境配置方面，在虚拟机内配置服务器环境会遇到各种麻烦的问题，然而 Docker 容器可以通过制作包含应用运行依赖的镜像，一键部署，这样便减少了许多不必要的麻烦。也正是因为这一点，容器技术比虚拟机技术具有更加轻量化的显著优势。另外，由于 Docker 直接使用宿主机操作系统内核提供的运行支持和资源管理机制，本身不再需要在客户机

上单独地安装操作系统，如图 2-2 所示，所以启动一个容器的耗时就相当于启动一个操作系统的普通进程用时。这就直接影响了容器的启动速度，对比虚拟机而言会更快、更高效。

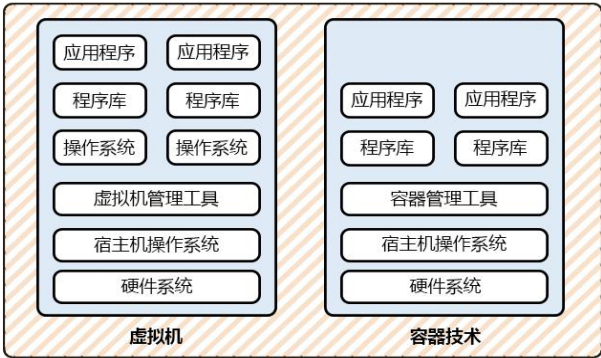


图 2-2：虚拟机技术与容器技术的对比

Docker 的架构是使用了 C/S 模式。用户只需要通过 Docker 的客户端就能与 Docker 服务端的守护进程（即 Docker 引擎）进行通信[18]，Docker 服务端守护进程运行在一台主机上。Docker 服务端守护进程会提供一组被称为 Docker Remote API 的 REST API[19]。它主要负责处理一些繁复的重型任务，比如建立容器、启动运行容器和发布自己制作的 Docker 容器等。另外，Docker 客户端和 Docker 引擎相互之间是通过 RESTful API 或 socket 来进行通信的。因此，尽管我们作为一个用户表面上好像是在本地的机器上执行各种 Docker 功能，但事实上，这一切的操作都是使用远程调用的形式，转换在服务端完成的[20]。

Docker 内部由镜像（Docker images）、仓库（Docker registries）、容器（Docker containers）三个部件构成[21]。Docker 镜像可以看成是 Docker 容器运行时的一个只读模板，每一个 Docker 镜像都是由一系列不同的层（layers）组成的。Docker 通过联合文件系统把这一系列不同的层联合到每个单独的镜像中。Docker 仓库分为公有和私有的，是用来保存和发布自己创建的 Docker 镜像的地方。一个从 Docker 镜像创建的 Docker 容器内应该集成了某个应用运行时需要的所有依赖和运行环境。Docker 提供了 Restful 风格的编程接口，支持多种编程语言。可以进行例如容器的创建、启动、销毁、状态查看等操作。当用户启动运行一个 Docker 容器时，Docker 只需在只读的镜像上再增添一个读写层，然后将这个读写层通

过联合文件系统挂载为一个容器内部完整的文件系统。用户对容器文件系统作出的所有更改，都保存在读写层中，不会影响到只读的镜像[5]。同时，用户也可以将修改后的读写层提交成为一个新的 Docker 容器镜像。另外，还有一种方式就是可以通过编写 Dockerfile，在已有镜像的基础上重新构建形成一个新的镜像。也因此，Docker 镜像具有非常强大的可重用性、可扩展性。

2.3 Mesos 资源管理框架

时下正处于一个互联网行业飞速蓬勃发展的时代，由于需要满足层出不穷的各种需求，基于数据密集型应用各类计算框架因此兴盛，从支持离线处理的 MapReduce[22]，到支持在线处理的 Apache Storm，从迭代式计算框架 Spark[23]，到流式处理框架 S4[24]……各种框架都有自己的特性，满足特定的需求，很难说出高下之分[25]。而在实际的生产实践活动中，由于复杂的环境和不断变更的需求，这几种框架可能都会同时采用。如果将资源利用率、运维成本、数据共享等因素纳入考虑范围，生产者都会希望能把所有这些框架部署进一个公共的集群，共享集群的资源，统一使用集群中的资源。这样，也就顺势而产生了集群资源统一的管理调度平台，比较有影响力的代表就是 Mesos 和 Yarn。本文将介绍和采用的就是 Mesos 资源管理框架。

Mesos 主要的用途是搭建高效可扩展的分布式系统，并以此来支持各式各样不断增长的计算框架。总体上看，作为一个有鲜明特点的资源统一管理和调度平台，Mesos 具有如下特点：

- 可以支持很多种计算框架：Mesos 提供了一个全局资源管理器，所有接入 Mesos 集群的计算框架，都必须要先向该全局资源管理器申请资源。成功申请到资源之后，再由资源调度器决定该资源将会交由哪个任务使用。
- 高扩展性：Mesos 不会对计算框架水平扩展有制约影响。
- 高容错性：Mesos 通过数据本地持久化，具备良好的容错性。
- 隔离性：采用 Linux Container[26]对内存和 CPU 两种资源进行隔离。
- 高资源利用率：假如集群是采用静态的资源分配机制，即为每个计算框架都分配一个属于各自的集群，如图 2-3 所示。这种机制下，由于作业

自身的特点或者提交频率之类的原因，集群整体利用率将会变得很低。但当用 Mesos 进行统一的资源管理和调度时，各种作业互相交错运行，作业的提交频率也大幅度提高，这样，集群的资源利用率顺理成章地得到了一定提升。

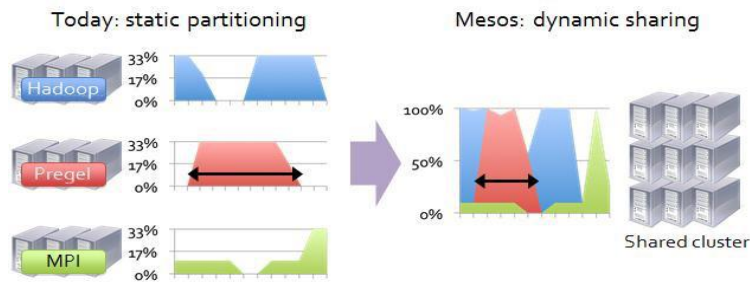


图 2-3: Mesos 的高资源利用率

- 资源分配细粒度：如图 2-4 所示，Mesos 不是像 MapReduce 那样以槽作为资源分配的单位，而是会按照实际任务的需求合理分配资源。这样做使作业互相交错运行，大大提高了集群中的资源利用效率。

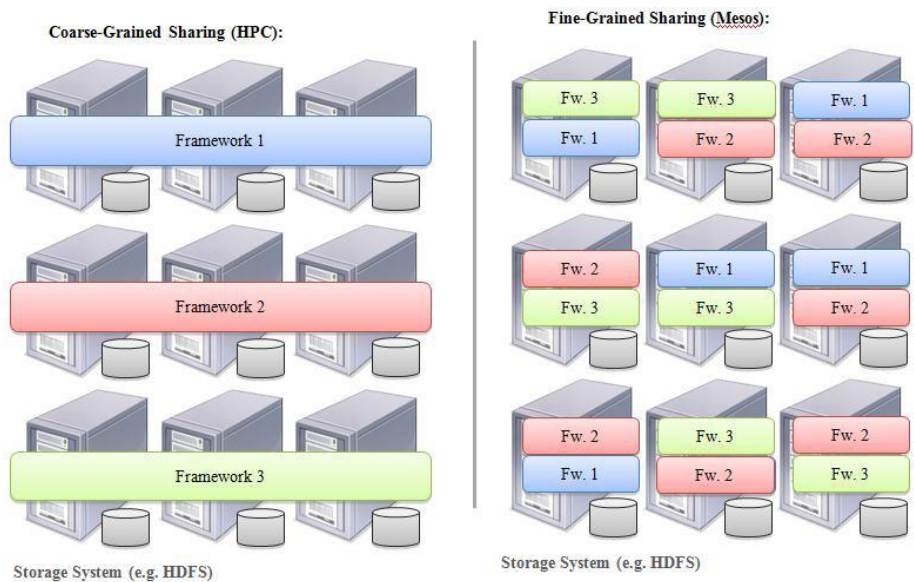


图 2-4: Mesos 的细粒度资源分配

在上图 2-4 中，假设左边是三个不同的集群，每个集群中有三台服务器，分别安装有三种分布式计算平台。例如，最上面一排装有 Hadoop，中间装有 Spark，

最下面三台装的是 Storm，这三个框架分开进行管理。而右边，使用 Mesos 框架对集群中的九台服务器统一进行调度管理。不管是 Hadoop、Spark 还是 Storm 的任务，都在这九台服务器上混合运行。这样一来，很明显能看出，使用 Mesos 进行集群统一管理可以大大提高资源冗余率。粗粒度的资源管理和分配方式必然会导致一定的浪费，但是细粒度的资源反而可以精细地提高资源管理能力。

总体上看，Mesos 是一个具有主从关系的分布式 Master/Slave 架构，如图 2-5 所示，Mesos 一共包含有四个组件，它们分别是：Mesos Master、Mesos Slave、Framework 和 Executor。一个典型的 Mesos 集群是通过若干个运行的 Mesos Master 服务器以及运行 Mesos Slave 的集群服务器组成的。在图 2-5 中的 zookeeper 是用来选举首领 Master 的，每个 Master 在被唤醒之后都要向 zookeeper 的 quorum 去注册 znode。但在本文的实验中，只需要一个节点作为 Master，所以 zookeeper 的功能在此没有启用。

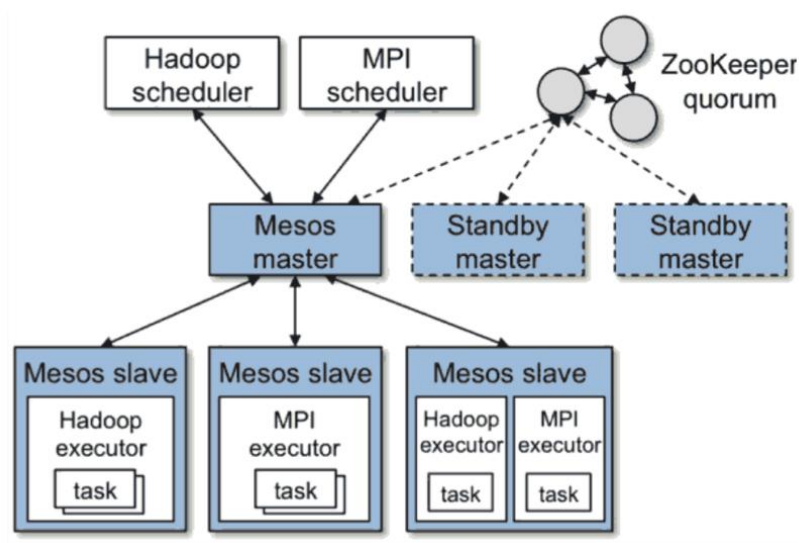


图 2-5: Mesos 框架结构

应用程序跑在 Framework 上，它也是分布式的软件[27]，包括调度器 scheduler 和执行器 executor，如图 2-5 所示。Framework 通过 Mesos Scheduler Driver 接入 Mesos，即可以简单理解为每个在 Mesos 框架里跑的应用，都首先需要注册到 Mesos Master 上。有两种不同的 Mesos Framework，一种对资源的需求是动态的，

类似 Hadoop 和 Spark；另一种，就是本文中出现的状况，对资源大小的需求是固定的，例如本文涉及到的 MPI。

Mesos 通过 Framework 的方法，提供了两级调度机制。上层是一个轻量的中央调度器，下层是具体某个应用程序自身携带的资源调度器，诸如 Hadoop、MPI、Spark 等等。图 2-6 良好地反应了 Mesos 的两级调度机制。图中蓝色方框部分是 Mesos 本身，白色部分即是待开发的 Framework。图中只简单展示了 Hadoop 和 MPI 两个框架，也可以替换成其他现有框架或者自定义框架。

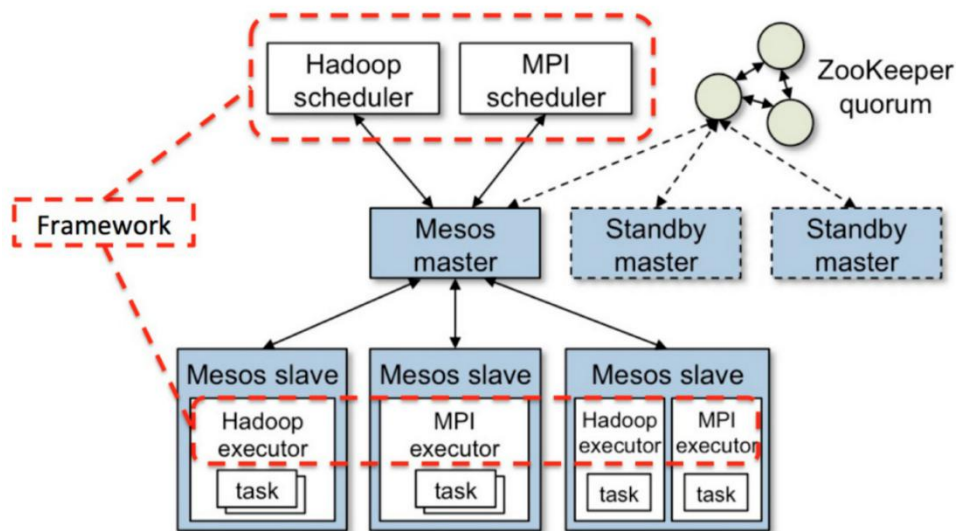


图 2-6: Mesos Framework

Mesos Master 也是极其轻量的，在这之上仅存有 Framework 和 Slave Node 的状态，它是一个全局的资源调度器。Mesos Slave 需要注册到 Master 上，并提供用于运行任务的资源，Master 负责与被部署的 Framework 进行交互，将 Slave 提供的那些资源交给 Framework，或接受指令运行任务，或委托这些指令给 Mesos Slave，整个流程如图 2-7 所示。

从图 2-7 可以看出，首先 Mesos 主服务器去查询现有的可用资源，提供给调度器。第二步调度器再向主服务器发出需要加载的任务，主服务器把这些任务都统一传达给从服务器，从服务器收到任务之后，给执行器发出命令加载任务并执行，执行器执行任务以后，再将任务状态反馈上报给从服务器，最终上报给调度

器。每个从服务器下会管理多个执行器，每个执行器都是一个容器，诸如 Linux 容器 LXC 和 Docker 容器等。

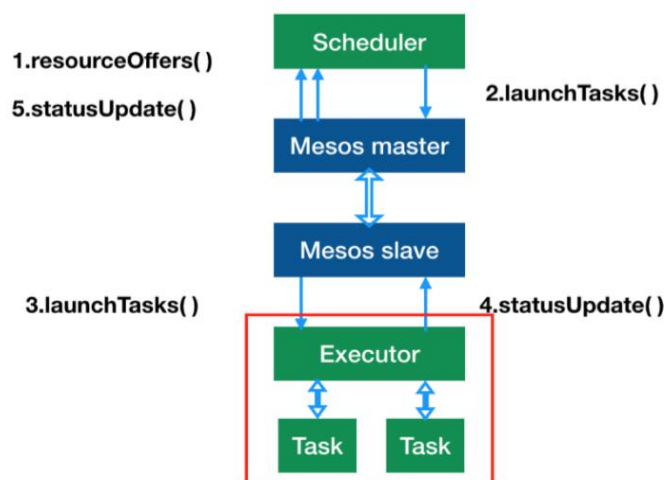


图 2-7: Mesos 主从服务器资源调度顺序

根据 Mesos 这些优良的特性，主要是可以将接入 Mesos 集群中的各种计算框架，根据实际需求，分配资源。使得各任务交错运行，达到混布的目的。本文研究也因此依托 Mesos 对资源进行管理和调度。

2.3.1 Scheduler 调度器

Mesos Scheduler 是 Mesos 框架中的调度器，Framework 都需要通过 Mesos Scheduler 注册到主节点，具体来说，是通过 Mesos Scheduler Driver 接入到 Mesos 中去[28]。并且，调度器会将作业分解为任务，同时一边申请相应的资源，一边监控任务运行的状态，向上下级传达。

Mesos 框架中采用的调度机制被叫做“Resource Offer”，这个调度机制是基于 CPU 和内存这些资源量的。在本文第二章对图 2-4 的分析中，我们举了使用 Mesos 管理 Hadoop、Spark、Storm 任务的例子，在此，依然假设右边 Mesos 集群中（见图 2-8）安装了 Hadoop、Spark、Storm 三个框架，并使用 Mesos 对它们进行资源调度管理。因为九台服务器都被 Mesos 管理，所以这里有一个数据稳定性问题。如果装的 Hadoop，Mesos 会进行调度，这期间的计算资源和存储都不共享，同时假如还在跑 Spark 框架进行网络数据迁移，显然会对速度影响很

大。于是就产生了“Resource Offer”这种资源分配机制，即对于在窗口的可调度资源可以由框架自行选择。

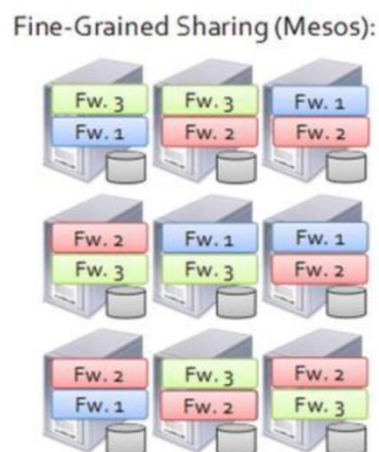


图 2-8: Mesos 的细粒度资源分配

Mesos slave 直接将资源量向上汇报给 Mesos Master, Master 获得这些信息后, 再把现有的资源量分配给 Framework, 这种分配方式参考了 DRF 机制 (Dominant Resource Fairness)。每个 Mesos Slave 都需要管理自己节点上的所有任务, 在 Framework 向 Mesos Master 成功申请到资源之后, 收到消息的 Slave 会立即启动对应 Framework 的执行器。

由前文已经提出, Mesos 其实采用的是双层调度的结构。类似这种二级调度的框架, 一方面, 在设计时需要考虑的问题肯定会有: “在不知道每个 Framework 对资源的需求时, 如何满足它们所有的需求?”, 对于 Mesos 来讲, 这个问题即为: “Framework 中有哪些数据、都存放在了哪几个节点, 这些都对 Mesos 来讲是不可知的, 在这种情况下 Mesos 需要如何做到数据本地化呢? ”。针对以上问题, Mesos 设计了一种名为 “reject offer” 的机制, 顾名思义, 就是 Mesos 允许 framework 有权暂时拒绝那些不满足它对资源的需求的 Slave, 这个机制也可以理解为与 Hadoop 框架中的 “delay scheduling” 调度机制对应。另一方面, 这种二级调度框架在设计时也需要考虑的问题还有关于决策问题。即两层不同的调度器分别要适应哪些调度机制的原理去执行, 是把大部分资源调度都交给第一层的调度器, 还是将第一层的调度器仅作为简单资源分配的支撑?

在整个 Mesos 集群环境中，作业资源调度从整体上看是一段分布式的过程。对于偶然出现失败的情况，这种机制必须要表现出鲁棒性，同时也不失高效性。因此，Mesos 平台采用了如下几种机制：

- (1) 过滤器 (filters) 机制：每一次调度的时候，Mesos Master 和 Framework scheduler 进行通信。如果某几个 Framework 总是拒绝 slave 提供的资源，那么这些通信开销将被视为是额外的开销，会影响调度性能。因此，为了避免与减少不必要的通信，产生了 filters 机制。允许 Framework 只接受节点列表中的 slave，或接收剩余资源量远大于 Framework 要求资源量的 slave。即每个 Framework 可以通过过滤机制告诉 Mesos Master 它对资源的偏好，比如只接受某些节点，或者是希望被分配来的 offer 中的节点上有多少空闲资源。这种机制某些程度上可以加速资源分配的收敛。
- (2) 回收 (rescinds) 机制：每次调度过程中，如果某个 Framework 在一段时间内没有返回为其分配的资源相对应的任务，Mesos 就会将这些资源量全部回收，并且按需分配给其他的 Framework。这种情况下，Mesos Master 就拥有动态调整长期、短期任务分布的功能了。

按照以上内容的分析，由于 Mesos 的调度机制是 “Resource Offer” 机制，面临着较大的资源碎片问题，每个节点上的资源不大可能被全部分配完毕，而同时，仅剩的那一点资源也很可能不足以提供给某个任务运行。如此便顺理成章地产生了资源的碎片问题。针对资源碎片化的优化，目前也已经有很多相关方面的研究了。本文的研究工作中，暂时不考虑这个问题。另外，Mesos 的资源分配设计上也有一个轻微的隐藏缺点。由于 Mesos 采用了无中心化的分配方式，所以很可能并不会带来全局最优解。但是当在一个计算中心的资源贡献率不超过 50%，即绝大部分的计算中心都处于空闲状态时，这个针对数据资源的缺点也就不严重，对本文的研究来讲也是无影响的。

简单概括来讲，Mesos Scheduler 负责的是按照作业输入的数据量，将其分解成若干子任务，并为每个子任务都申请资源，同时监控它们的运行状态，一旦发现某个任务运行失败，则再次为它申请新的资源，整个结构都如图 2-9 所示。

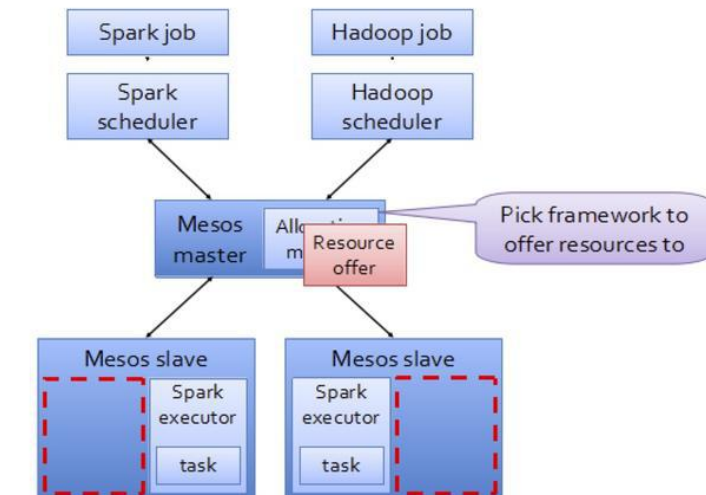


图 2-9: Mesos 的资源分配图

2.3.2 Executor 执行器

Mesos Executor 是 Mesos 框架中的执行器，安装在每个 slave 节点上，用于启动和负责执行每个 Framework 的任务，它才是真正的执行任务的逻辑。如图 2-10 所示，Mesos Executor 是一个 Container 进程，运行在这个 Container 进程中的每一个单元就是一个 task。

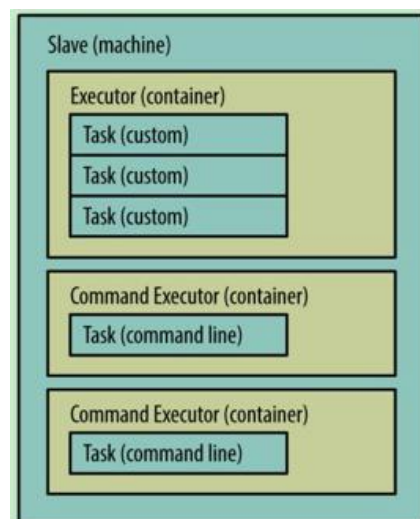


图 2-10: slaves(machines), executors(containers)和 tasks 的关系

在 Mesos 集群里不太区分需要执行的是哪些任务，这就给用户提供了一定的灵活性，可以按需编写不同的 Mesos Executor。例如较常用的 Mesos 运行容器的部分，如上图中所示，就是一个 Mesos Executor。当然，类似的还用 Map Reduce 这种大型任务，不管是 Map 还是 Reduce，Mesos Executor 都能够执行。它的表现形式是一串二进制的字符，Mesos 在运行时，Mesos slave 会把 Mesos Executor 从远程的一个 URL 上拉取下来，然后开始启动执行 Mesos Executor。

例如简单处理图片、文件这种小任务的 Mesos Executor，自定义一个 Executor 来实现这个接口，最终输出成一串二进制字符，放在一个 URL 里。然后 slave 就可以拉取该 Executor，然后执行任务。Mesos Executor 和 Mesos Slave 之间是通过进程间通信方式（网络端口）进行交互的。Mesos 也内置里两种 Executor，分别是 Command Executor（直接调用 shell）和 Docker Executor 这两种。

如果 Mesos Scheduler 和 Mesos Executor 是对应的，那么 Mesos Executor 就是执行的这一部分。在图 4-6 中的 container 这个 Executor 是 Mesos 框架自带的，用户不需要重新写一个 Mesos Executor 就可以运行一个 Docker 任务。但如果还有其他个性化的需求，就需要自己定制 Mesos Executor 了。在本文的工作中，直接使用 Mesos 支持的 Docker Executor。Mesos 可以结合 Chronos 和 Mesosphere 的 Marathon 框架运行和管理 Docker 容器，而 Executor 是 Mesos 和底层 Docker 任务之间的连接器。

2.4 本章小结

本章首先介绍了消息传递编程接口 MPI，以及 MPI 的一些特性；接着将容器技术和虚拟机技术多方面进行比较，凸显容器技术轻量与高效的特性，并选取本文用到的 Docker 容器技术为例，简单介绍了 Docker 内部的构造；最后在介绍统一资源管理调度平台时，着重介绍了 Mesos 框架，分别介绍了 Mesos Scheduler 和 Mesos Executor，更进一步阐明了 Mesos 资源管理的机制。并阐明它的优势。

第三章 基于 Mesos 的 MPI 容器化运行研究

3.1 方案整体描述

本文的第一个目标就是要为用户提供一个随取随用的 MPI 编程运行环境，且适用于不同的操作系统，例如 macOS、Windows、Linux 等。要实现这个目标，容器技术是一种非常合适的手段。假如使用容器技术，用户就不需要在集群中的每台主机上都配置相关环境，只需下载容器镜像，即可一键部署与安装。所以本文将采用 Docker 封装 MPI 运行环境，选取 mpich-3.2.1 作为 MPI 的具体实现。这样，任何操作系统的用户，都能通过下载该 Docker 镜像来启动一个可运行 MPI 任务的环境。

第二个目标是要合理利用集群资源，在运行 MPI 任务的同时也要考虑任务混布的需求，因此本文采取 Mesos 进行统一调度管理。Mesos 作为资源管理平台，它可以使任何接入集群的框架都能按需窗口获得资源，对集群资源进行细粒度的分配，以减少粗粒度分配资源时的资源冗余率。但是，使用 Mesos 平台做资源管理调度时，要使 MPI 任务在 Mesos 平台上顺畅运行，需要通过自定义 Mesos Scheduler 来实现这个目标。

整个解决方案的环境图如图 3-1 所示。

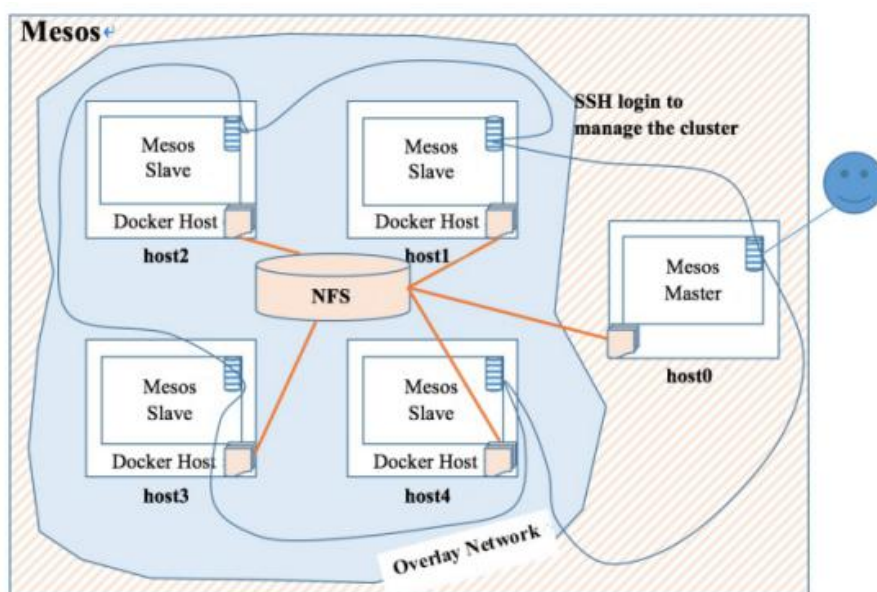


图 3-1: 本文研究方案的整体环境

3.2 MPI 的容器化封装

如前文所说，本文工作的第一步是 MPI 的容器化封装。为达成这个目标，首先需要制作包含 MPI 运行环境的镜像，其次考虑到后续要投入 Mesos 平台使用，所以在不同主机间的不同容器中还要做到计算、存储资源的隔离。所以，本章将分为两个部分来介绍：制作 Docker 镜像和用 Overlay 网络在计算存储资源相互隔离的情况下实现容器跨主机通信。

3.2.1 使用 Docker 制作镜像

Docker 项目提供了一组构建于 Linux 内核功能之上，协同运行的高级工具。它的目标是帮助运维人员以及开发人员更便捷地跨主机跨系统进行应用程序交付和获得相关依赖。通过 Docker 容器——一个安全的、轻量级的环境，可以达成这个目标。Docker 使用 Linux Container 进行自动化部署，是最为突出的软件集装箱化平台之一。并且，Docker 可以运行在很多操作系统上，诸如多数现代 Linux 发行版、macOS、Windows 和各种云服务提供商上。使用 Docker 可以非常便捷地为任何一个应用创造出轻量的、可移植的、安全性高且拥有良好生态环境的容器。这些容器能够被批量地在生产环境中发布和部署。容器内置了 Linux 内核功能，有 CGroups、Namespaces、UnionFS 等等，并且完全按照沙盒机制的理念，它们相互之间不会有任何接口，互相封闭，性能开销极低。

Docker 实现虚拟化的方式是，开发者打包他们开发的应用及依赖包，然后封装到一个可移植的容器中，最后上传发布。便捷已成为 Docker 最显著的优势，在过去某个任务可能需要经过数天甚至数周才能完成，但是现在依托 Docker 容器，只需要数秒的时间就能完成这些操作。伴随着云计算的兴盛，开发者不需要配置高额的硬件才能追求效果，使用 Docker 依然可以达到这个目的，因此这一举措也一定程度上改变了高性能必然导致高价的一种思维定势。Docker 与云两相结合，云空间因此可以得到更充分的利用，这不仅可以解决硬件管理问题，更是提出了新型的虚拟化方式。

从图 3-2 中我们可以看出，Docker 包含了三个最为基本的概念，即为镜像（Image）、容器（Container）和仓库（Repository）。Docker 运行容器的前提是镜像，镜像存放在仓库中，由此可见镜像 Docker 技术的核心。

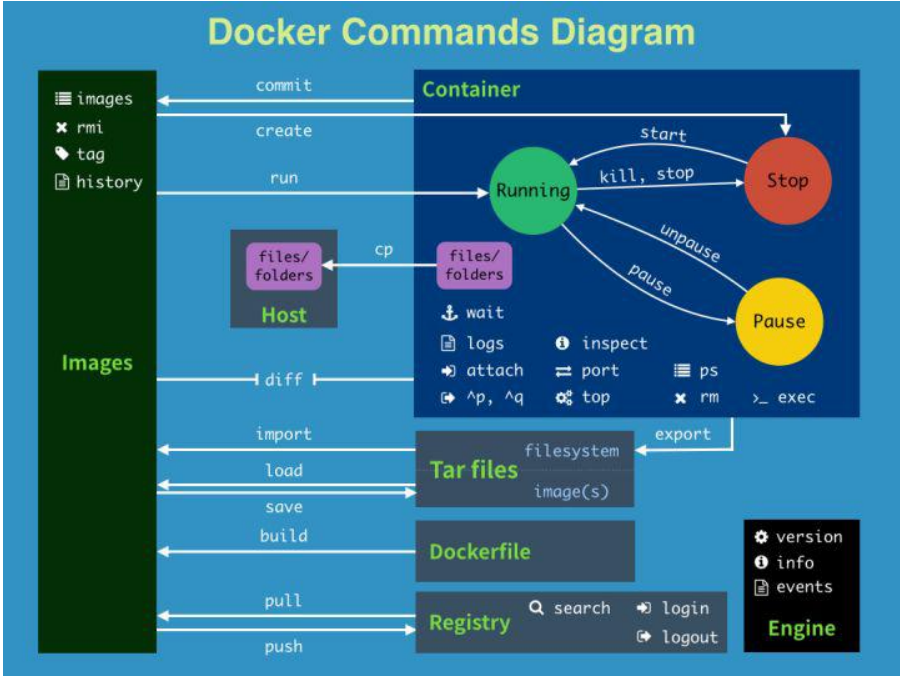


图 3-2: Mesos 的高资源利用率

可以将 Docker 镜像视为一个特殊的文件系统，它包含操作系统的基础文件及软件运行的依赖和环境。Docker 镜像在提供容器运行时所需的程序、库、配置文件外，还包含了例如环境变量、匿名卷、用户等为运行时准备的配置参数。Docker 镜像中没有动态数据，并且它的内容即使在构建后也依然不会被改变。容器（container）和镜像（image）的联系及区别如图 3-3 所示。

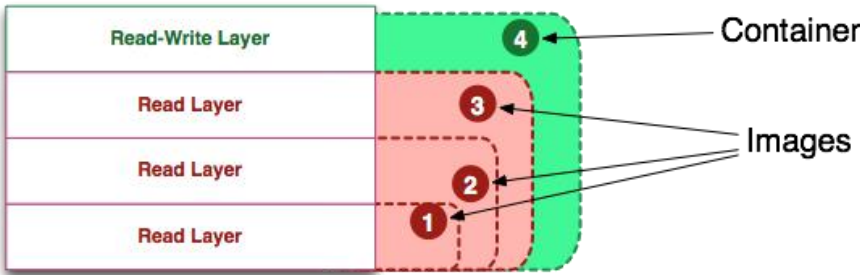


图 3-3: 容器与镜像

如图 3-4 中的结构所示，镜像（Image）实质上是一套只读层（read-only layer）的统一视角。

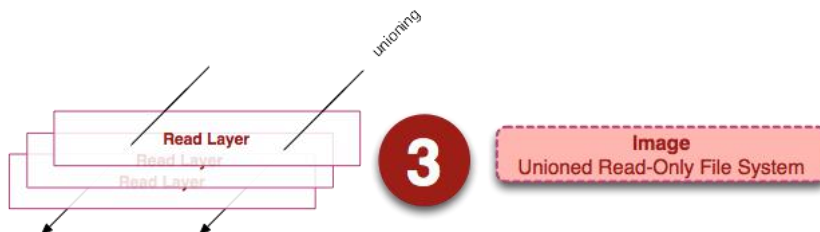


图 3-4：镜像的定义

从图 3-4 的左边我们可以看到重叠在一起的多个只读层。除最底层之外，其他每一层都会有一个指针指向下一层。Docker 内部的实现细节就是由这些层组合起来的，并且它们能够在主机的文件系统上被访问到。统一文件系统（union file system）技术可以做到把不同的层整合成为一个新的文件系统，并为这些层提供统一的视角，这样就实现了对多层存在的隐藏。从用户的角度来看，如图 3-4 的右边，只存在一个文件系统。作为 Docker 最优秀的创新之一，镜像的产生变革了软件交付标准。如果想理解整个 Docker 的生命周期，必须先理解镜像的含义。

由于一个 Docker 镜像中包含着应用的运行环境和依赖配置，所以使用 Docker 可以对多种应用实例工作的部署进行简化。像 Web 应用、后台应用、数据库应用、大数据应用等应用都可以打包成一个镜像，然后再进行一键部署。

当发出运行容器的指令时，如果将要投入使用的镜像本地没有，Docker 就会自动去镜像仓库中寻找并且下载，默认的镜像仓库是 Docker Hub 公共镜像源。但是正因为如此，大部分 Docker 镜像都相当于开箱即用的，有时候却并不一定能够满足我们的某些个性化需求。举个例子，我们直接从 Docker Hub 拉取了一个 tomcat 镜像，它使用的 Java VM 是 Open JDK，但我们需要的却是 Oracle JDK。由于版权之类的种种问题，Docker Hub 并不能提供基于 Oracle 的 JDK 镜像，这时候就需要我们自己动手，深度定制满足我们目标的新镜像了。

我们可以使用定义 dockerfile 或者通过 docker commit 命令来扩展和重建新的镜像。在本文的研究中，也因为需要自定义镜像，所以采取了 Alpine 作为基础

操作系统，它是一个大小只有 5MB 的轻型操作系统，通过编写 Docker，在这之上添加了 mpich-3.2.1 的环境以及一些必要的工具例如 gcc、make，总大小也只有 150MB。这与我们轻量级的目标十分相符。

3.2.2 容器间跨主机通信

针对本文中的跨主机容器应用场景，由于集群中存在较多个运行着 Docker 容器的主机，因此我们需要建立一个跨多节点的覆盖网络，并且可以让运行在不同主机上的 Docker 容器可以通过此覆盖网络感知到彼此。所以在网络架构时，会考虑以下几个原则：

- 容器与 IP 一一对应；
- 容器与主机、主机与主机之间互联；
- 网络隔离；
- 高性价比的性能损耗；

一般来讲有两种方法，一种是创建两个不同的 Overlay 网络，使用 Bridge 网络制式将它们连接；另一种则是直接在同一个 Overlay 网络中实现[29]，本文采用后者。不同主机间的多个容器可以通过网络资源管理器运行在同一个 Overlay 网络中，以此大幅度减少网络逻辑的复杂性，同时也可以保证二者的计算和存储资源互相隔离，满足设计需要。

简而言之，Docker 网络是原生的容器 SDN 解决方案。Docker 技术一共提供了包括 Bridge（桥模式）、Host（主机模式）、Container（容器模式）、None（无网络模式）这四种网络模式[30][31]。在默认的 Docker 网络环境下，单个主机上的 Docker 容器可以凭借 docker0 网桥与宿主机直接进行通信。但是不同主机上的 Docker 容器之间只能各自在宿主机上进行端口映射，这样才可以互相通信。显然这种通过做端口映射来进行通信的方案对很多集群来说是极不方便使用的。如果 Docker 容器之间可以直接使用自己的 IP 地址，以此来进行通信，这样很多问题就将被一应而解。因此，针对分布式集群环境，按照实现原理的不同，Docker 提供了几种方案，比如直接路由方式、桥接方式、Overlay 隧道方式。在本文的研究工作中将会采取其中的 Overlay 网络模式。

Overlay 是一种新的数据格式，它被封装在 IP 报文之上，实现的前提是不改变已有的网络基础设施，通过某种既定的通信协议，在 IP 报文之上再封装一个二层报文。因此这种新型数据格式可以通过已经十分成熟的 IP 路由协议在网络集群中进行分发。Overlay 采用扩展的隔离标识位数，它能够突破 VLAN 对于数量的限制，支持更多的用户，并在必要时把广播流量转成组播流量，这种方式有效地避免广播数据泛滥。因此，Overlay 网络是目前较为主流的容器跨节点数据传输方案，对于本文针对网络架构要求的重合度也较为饱和。

Docker 容器在不同主机之间互相通信时，将会按照 Overlay Network 这个网络模式来进行通信活动。如图 3-5 所示，本文基于 Docker Overlay Network 技术，在 host1~4 之间建立了 Overlay 网络，并且配置了网络内部的容器，使各容器之间可以相互通信。

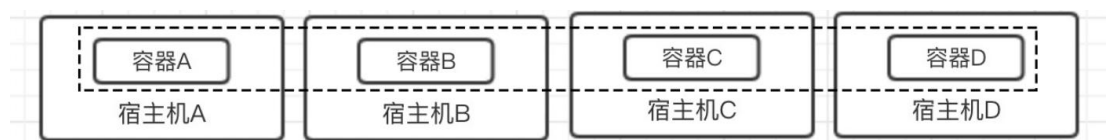


图 3-5: Overlay 网络

在具备 MPI 运行环境以及启动了若干 Docker 容器的前提下，本文在配置 Overlay 网络的过程中，使用命令如下。

```
$ docker network create
    --driver overlay      \
    --subnet 10.0.9.0/24  \
    --opt encrypted       \
    mpi-network            \
```

现在，我们就可以使整个 MPI 集群中的所有节点都连接到这个 Overlay 网络中去了。整个 Overlay 网络的环境图如图 3-6 所示。

在 host1~4 这四个服务器上，每个主机都作为工作节点，并包含一个可以运行 MPI 任务的容器。整体环境都搭载在 Mesos 平台上。

由于每个容器都是基于同一个 Docker 镜像的，他们都有互相认证的 SSH 公钥和私钥，这样他们相互之间就能进行通信，如图 3-7 所示。

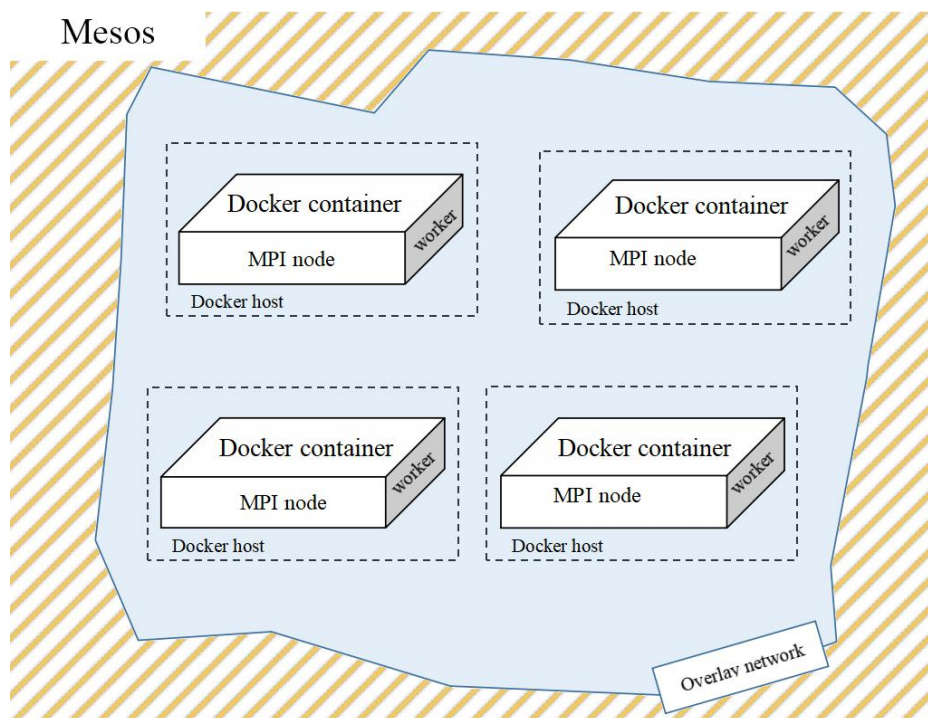


图 3-6: 本文研究中的 overlay 网络环境

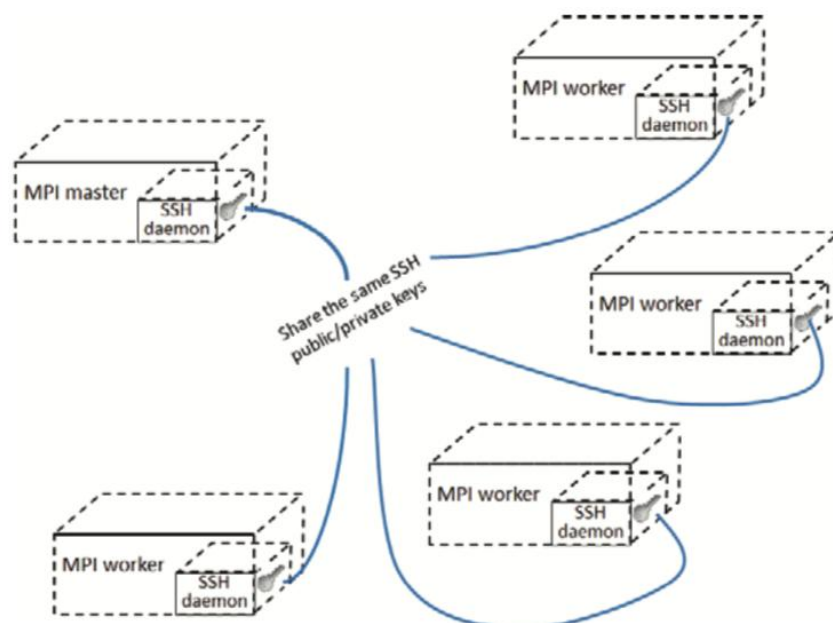


图 3-7: SSH 充当 MPI 节点间的通信端口

3.3 使用 Mesos 进行资源管理

根据 2.3 节中已经介绍过，Mesos 的一大优势就是可以灵活调度资源，使得各种接入集群的计算框架都能够按需在窗口获得资源，从而使任务交错运行，大幅提高集群的资源利用率。

在本文的研究工作中，3.2 节已经介绍了如何将 MPI 环境打包在 Docker 容器里。接下来，本节将解决如何在 Mesos 集群中部署该打包了 MPI 环境的 Docker 容器，以及如何为运行在 Mesos 集群中的 MPI 任务调度、分配资源这两个问题。首先，由于 MPI 任务是运行在集群中不同主机的不同容器上的，所以这些具有相互依赖关系而又不同的容器需要一个共享目录，用于存放运行时必要的文件和运行结果的临时输出。所以，本文将在集群中创建 NFS 网络文件共享系统，然后将共享目录卷挂载到容器镜像中，即可达成数据一致性的目标。其次，如何使 MPI 任务运行在 Mesos 集群中，这就需要我们自定义一个 Mesos Scheduler 调度集群资源来适应 MPI 本身的特性。因此，本节将分为两个部分来介绍使用 Mesos 做资源管理的具体工作，即使用 NFS 网络文件共享系统解决存储一致性问题 and 自定义 Mesos Scheduler 实现资源分配策略。

3.3.1 NFS 网络文件共享系统

针对这此类容器跨主机的集群分布式应用，目的是要所有容器实例的存储都保证高度一致性。所以，怎样才能把存储资源集中起来，即便用户的某应用需要启动多个容器，并且这些不同的容器都跑在集群中不同的主机上的实际应用场景下，依然可以达成存储数据保持一致的目标，也是本文需要研究探讨的一个问题。

本文采用 NFS 网络文件系统来整合共享存储资源。NFS，全称是 Network File System。它是一种基于 Unix 操作系统的网络文件系统，可以将之理解成是一种网络文件传输协议[32]。NFS 可以定义文件数据在网络中通过何种方式去传输，以及通过怎样的协议去访问网络端的文件。NFS 网络文件系统允许某个主机在网络上通过 TCP/IP 网络协议与其他主机共享目录及包含的所有文件。举个例子，假设现在有三台机器，分别叫 A、B、C。它们都需要访问同一个内容都是图片的目录，比较传统的方法是将这些图片分别存入 A、B、C 三台主机中。但如果

替换为采用 NFS 网络文件系统，我们仅需要将准备共享的那些图片先存到主机 A 的某一个目录中去，然后让主机 A 把该目录共享给 B 和 C 即可。主机 B 和 C 进行访问的时候，是通过网络的方式去访问 A 上的共享目录的。

可以使用 NFS 进行网络文件共享的对象包括任何整个或部分目录树，以及文件分层结构，也包括单个文件。计算机在 NFS 网络共享文件系统中承担的角色有两种名分别是客户机和服务器，如图 3-8 所示。通过网络向其他主机共享其文件系统的计算机就负责扮演服务器的角色，那么那些访问共享文件系统的计算机就相应地被称为客户机。在使用 NFS 服务时，网络中的任何一台主机都可以访问到其他任何计算机的共享文件系统。同时，NFS 服务也支持对其自身文件系统进行访问。网络中的计算机可以在任何特定情境下仅扮演客户机或服务器其中一种角色，也可以同时拥有客户机和服务器这两个双重的角色。

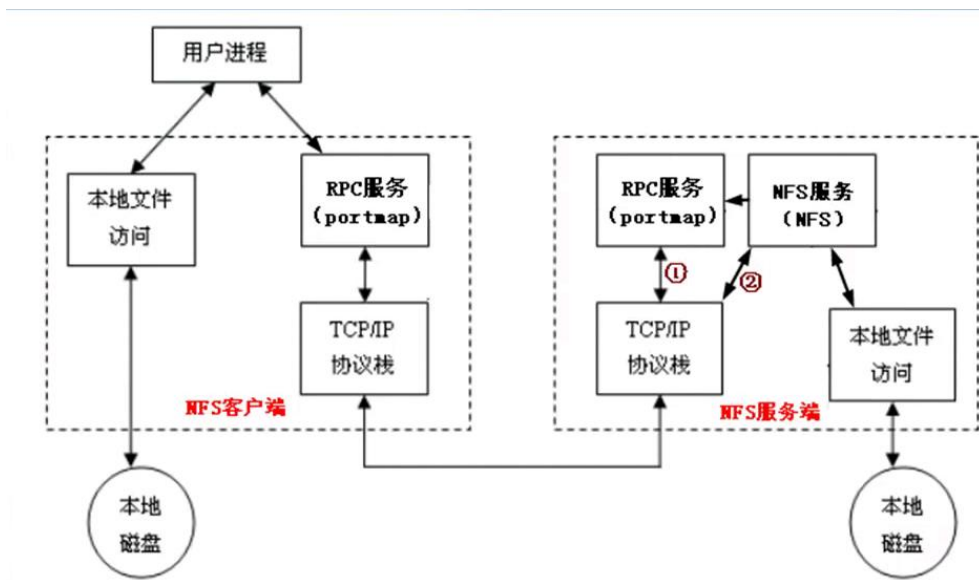


图 3-8：客户端和服务端的通信过程

客户机如果需要访问服务器上的文件，是通过挂载服务器的共享文件系统这种方式来达成目标的。客户机在挂载远程的共享文件系统时，并不会复制这个文件系统。由于挂载文件系统这个进程必然会涉及到一系列远程的过程调用，客户机可以通过这些调用对服务器磁盘上的文件系统进行透明访问。这种挂载方式和本地挂载相类似，用户键入命令时这些文件系统就像是本地的文件系统一样。

使用 NFS 进行网络文件共享有十分显著的优点：一是可以尽可能地节省本地存储空间。如果把经常用到的数据都统一存放在一台 NFS 服务器上，并且其

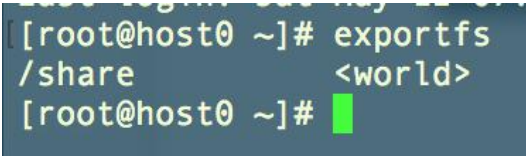
他服务器可以通过网络访问到这些常用数据，那么本地终端就可以减少自身对存储空间的使用要求；二是用户不需要在网络中的每台主机上都创建一个 Home 目录，Home 目录都被存放在 NFS 系统中的服务器上，并且可以在通过网络被访问和使用；三是服务器上的存储设备也都可以通过网络被别的机器使用，这就可以适当减少整个网络中可移动介质设备的数量。

在本文的实验环境中，共有五台服务器，分别如表 3-1 所示。

主机名	IP 地址	操作系统	安装服务
host0	172.16.0.157	CentOS7	SSH、mpich、NFS、mesos
host1	172.16.0.161	CentOS7	SSH、mpich、NFS、mesos
host2	172.16.0.170	CentOS7	SSH、mpich、NFS、mesos
host3	172.16.0.169	CentOS7	SSH、mpich、NFS、mesos
host4	172.16.0.167	CentOS7	SSH、mpich、NFS、mesos

表 3-1：集群中每个主机的基本信息

其中，NFS 配置于整个集群每台服务器上的 /share 目录下，如图 3-9 所示。



```
[root@host0 ~]# exportfs
/share <world>
[root@host0 ~]#
```

图 3-9：NFS 共享文件夹路径

首先，本文研究所用的五台机器均配置好 SSH 并且互相之间可以免密登录。在服务端，关闭防火墙和安装好所需的软件包之后，修改 NFS 配置文件，定义 NFS 共享，并创建 /share 目录。在客户端，将服务器上的共享目录 /share 挂载到本机的 /share 目录下。其中，host0 充当 NFS 服务器角色，集群中另有 host1~4 作为 NFS 客户机，集群中五台机器通过 /share 目录进行 NFS 文件共享,整体结构如图 3-10 所示。

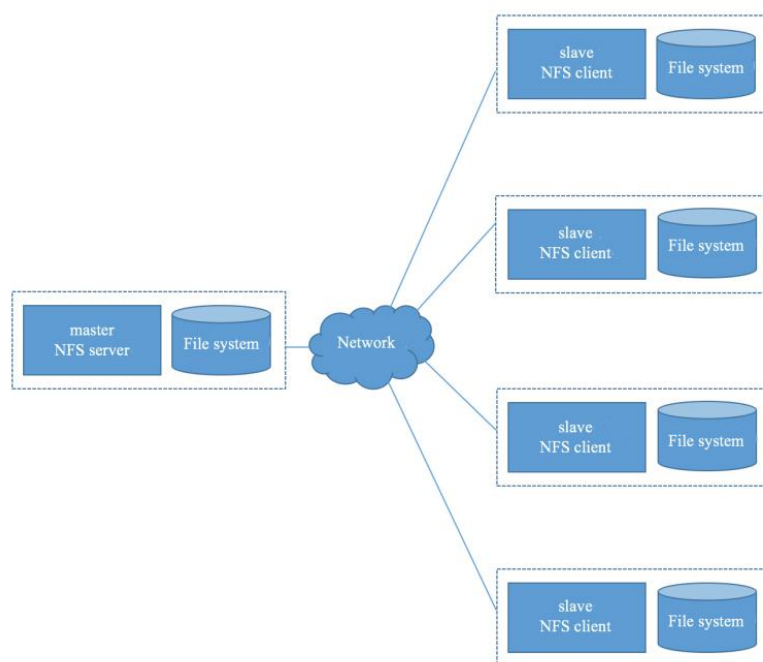


图 3-10: 本文研究中的 NFS 文件系统结构

3.3.2 自定义 Mesos Scheduler

前文中的概念指出，有时候用户需要根据个性化的需求自定义 Framework。Mesos 为开发者提供了一个 Framework 的基础库，只要实现 Mesos Scheduler 和 Mesos Executor 这两个接口即可，主要是 Scheduler Driver 和 Executor Driver 的实现。同时，该基础库支持例如 C++、Java、Python、Go 等语言。

由于 Mesos 框架中的 Framework 都是独立于整个系统的，其具体的部署方式以及高可用性等都需要 Framework 的开发者自己解决。所以需要实现一个完备高可用、稳定性好的 Framework，还是具有相当高的复杂度的。另外，Mesos Framework 的机制比较适合有任务分发调度需求的分布式系统。比如 Hadoop 和本文中采用的 MPI 等。

在本文的研究工作中，为了使 MPI 任务可以在 Mesos 平台上顺畅运行，首先需要使 Mesos 平台的调度策略适配 MPI 的资源调度。因此，需要自定义 Mesos Scheduler 来达成这个目标。

本文工作将按照如下方式自定义 Mesos Scheduler:

第一，实现资源筛选和节点挑选策略。本文通过实现 Mesos Scheduler 提供的一个接口 `resourceOffer` 方法，在这个方法中，根据申请任务对资源的要求筛选资源，然后从中挑选出适合运行本次任务的节点；

第二，任务拿到资源后启动相应的容器。这个步骤可以通过实现 Mesos Scheduler 中的 `statusUpdate` 方法，将任务和将要执行任务的对应节点合并成一个键值对，然后向 Master 反馈这个任务和节点的键值关系，Master 会在相应的 Slave 节点上启动容器，并分配属于它的任务；

第三，在每个成功启动的容器中，执行各自的 MPI 任务。这就需要考虑到如何启动执行 MPI 任务的进程，以及这些进程通过何种方式知道如何执行 MPI 任务。所以，本文在被分配到任务的节点上都启动 `mpich` 自身的 `hydra` 进程管理器中名叫 `hydra_pmi_proxy` 的进程代理，MPI 任务在传统的 MPI 集群中都是运行在 `hydra_pmi_proxy` 之上的。然后，本文在客户端实现一个 `startMPIExec` 方法，该方法会以子进程的方式启动 `mpiexec.hydra`。这个 `mpiexec.hydra` 用于向 `hydra_pmi_proxy` 发送任务将如何运行的命令。

第四，`hydra_pmi_proxy` 执行完 MPI 任务后，将结果反馈给 `mpiexec.hydra`，最终在客户端显示，所有任务结束后销毁容器。

以上就是本文自定义 Mesos Scheduler 时的整体思路与流程。下面将分别介绍每一步的具体实现。

在挑选可执行任务节点时的策略时，需要将集群中所有空闲的资源进行过滤。资源都是有多个维度的，比如 CPU、内存与磁盘，可以将之定义为一个向量。每个任务申请资源时，就都相当于在申请该资源向量。而调度器的作用，就在于怎么从任务申请中选出合适的任务来执行。所以我们需要先筛选出符合执行任务要求的资源，实现逻辑如图 3-11 所示，首先根据任务要求的 CPU 或者 Memory 这个向量的维度来判断资源是否符合要求，如果不符合就拒绝这个接受资源 Offer，如果符合，就将这个资源筛选出来。下一步就是要向对应提供资源的节点发布任务。

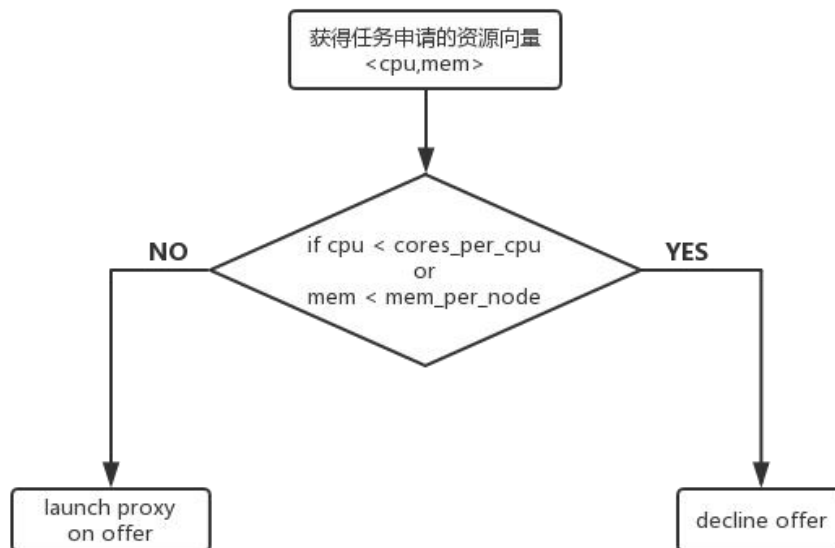


图 3-11: 筛选资源流程

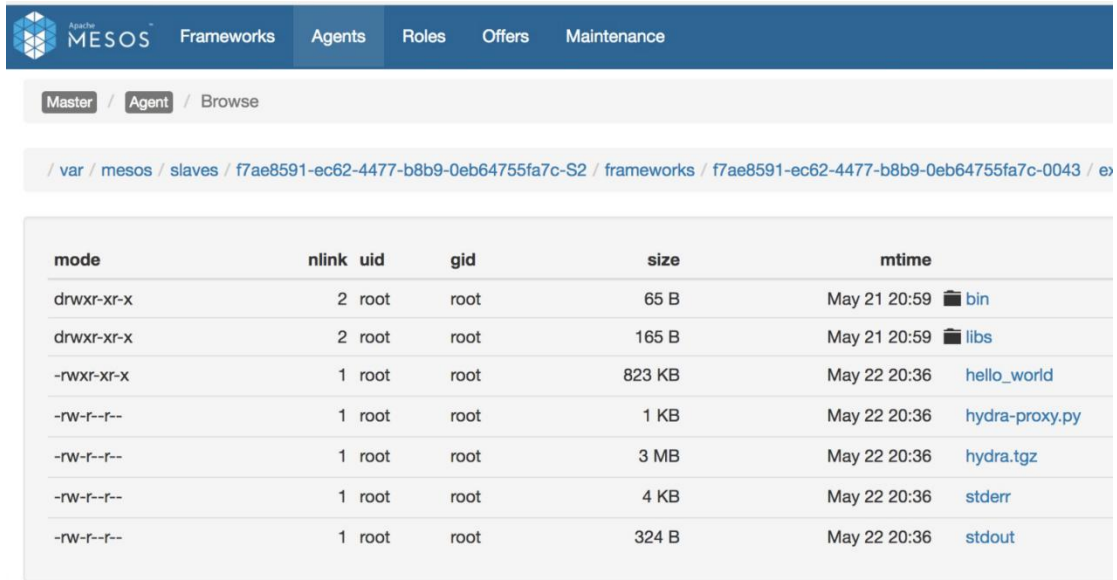
筛选出符合任务要求的资源后，将对它们进行选择，其中，涉及到了贪心算法中的装箱问题。可以用 **first fit**，即选取首先满足任务申请资源数的节点；也可以用 **best fit**，即这些满足需求资源中最适合本任务运行的节点，这种方法也就涉及到了前文提及的资源碎片最小化的问题。同样，本文暂不考虑资源碎片的影响，选择 **first fit** 算法，对已满足任务申请资源数的节点分发任务。

在挑选出合适的任务执行节点后，就需要按照 2.3 节介绍的 Mesos 工作方式执行调度流程。Mesos Master 会通过心跳来获得 Mesos slave 的情况，然后 Mesos scheduler 通过 client 接受用户申请，获得申请后根据图 3-11 筛选出节点来运行任务。最后，把任务和节点组成的键值对关系反馈给 Mesos Master。整个调度流程在图 2-7 中可见。

Mesos Master 从 Mesos Slave 处接收到任务消息后，启动 Mesos Slave 上的 Executor。然后运行程序会启动容器，MPI 任务包含在容器中，并对容器作出资源限制。另外，Executor 上还会暴露出一个包含消息的端口，反馈运行情况。当任务完成后，任务运行结果会写进 NFS 中。最后，销毁容器。

前文已经说明，本文采用了 Overlay 网络使得不同主机间的容器之间可以相互通信。容器被分配到了 Overlay 网络中固定的 IP，会为每个任务创建一个沙盒

空间，运行任务产生的临时输出就会保存在沙盒中。一旦程序的生命周期结束，这个沙盒就会销毁。如图 3-12 展示了本文研究中的沙盒内容，这种沙盒机制，就可以将计算、网络、存储进行隔离。



mode	nlink	uid	gid	size	mtime
drwxr-xr-x	2	root	root	65 B	May 21 20:59 bin
drwxr-xr-x	2	root	root	165 B	May 21 20:59 libs
-rwxr-xr-x	1	root	root	823 KB	May 22 20:36 hello_world
-rw-r--r--	1	root	root	1 KB	May 22 20:36 hydra-proxy.py
-rw-r--r--	1	root	root	3 MB	May 22 20:36 hydra.tgz
-rw-r--r--	1	root	root	4 KB	May 22 20:36 stderr
-rw-r--r--	1	root	root	324 B	May 22 20:36 stdout

图 3-12: 沙盒内容

在本文的实现方式中，首先，指定 NFS 共享目录的路径；然后执行 make 命令，最后通过如下命令分配资源、执行任务：

```
$ ./mrun -N <#Nodes> -n <#MPI processes> -c <#Cores per MPI process>
<leading-master> <mpi-program>
```

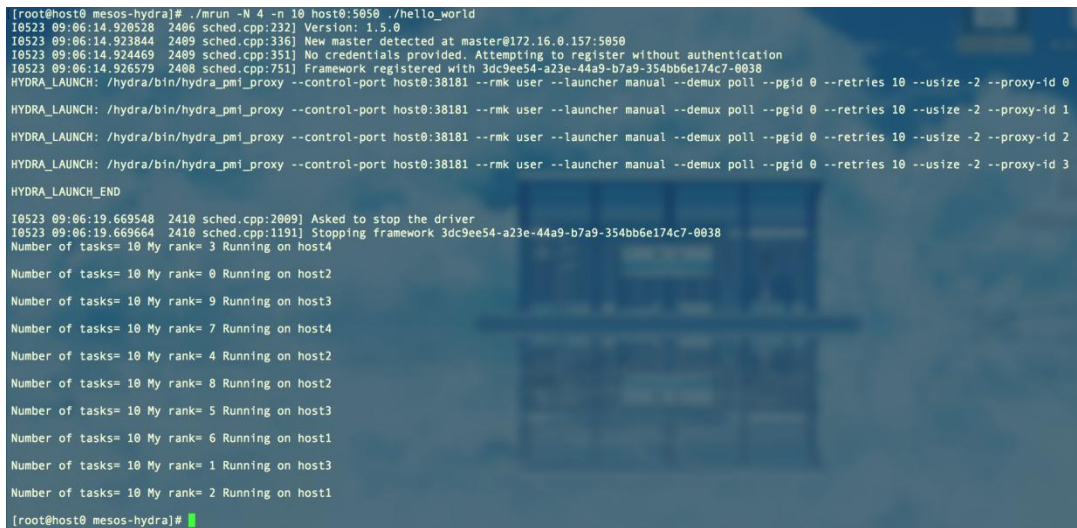
其中，对于一个任务的资源申请，-N 指定节点数量，-n 指定任务数，-c 指定每个任务需要的 cpu cores。在图 2-7 的 resourceOffer 方法中，分配资源，主要是 CPU 和内存，同时需要占用一个端口资源。然后启动 Executor，在每个 Executor 中执行 hydra_pmi_proxy 进程，该进程的作用类似于 MPI 中的 mpd 守护进程，同时，hydra_pmi_proxy 也会监听上文提及被占用的端口。

同时，在 statusUpdate 这个方法中，有一个事件监听。当 Executor 启动以后，会告诉 Mesos Scheduler 该任务状态变为 TASK_RUNNING，也就意味着 Executor 上的 hydra_pmi_proxy 启动成功了。之后，就可以在这之上分配任务。这时会调

用 startMPIExec 方法，该方法会在 client 上以子进程的方式启动 mpiexec.hydra，然后该文件会生成如下命令来告知 hydra_pmi_proxy 如何去执行任务：

```
HYDRA_LAUNCH:/tmp/tmpUS93PF/./export/bin/hydra_pmi_proxy
--control-port host0:41204 --rmk user --launcher manual --demux poll --pgid 0
--retries 10 --usize -2 --proxy-id"
```

最后，Scheduler 会直接和 hydra_pmi_proxy 通信，将此命令通过 TCP 发出去，然后 hydra_pmi_proxy 会执行这个任务命令，同时通过 TCP 返回输出结果，由 mpiexec.hydra 得到结果，再由子进程的管道返回给 Scheduler。最后在 client 上得到输出结果。本文中的样例如下图 3-13 所示。



```
[root@host0 mesos-hydra]# ./mrunc -N 4 -n 10 host0:5050 ./hello_world
I0523 09:06:14.920528 2406 sched.cpp:232] Version: 1.5.0
I0523 09:06:14.923844 2409 sched.cpp:336] New master detected at master@172.16.0.157:5050
I0523 09:06:14.924469 2409 sched.cpp:351] No credentials provided. Attempting to register without authentication
I0523 09:06:14.926579 2408 sched.cpp:751] Framework registered with 3dc9ee54-a23e-44a9-b7a9-354bb6e174c7-0038
HYDRA_LAUNCH: /hydra/bin/hydra_pmi_proxy --control-port host0:38181 --rmk user --launcher manual --demux poll --pgid 0 --retries 10 --usize -2 --proxy-id 0
HYDRA_LAUNCH: /hydra/bin/hydra_pmi_proxy --control-port host0:38181 --rmk user --launcher manual --demux poll --pgid 0 --retries 10 --usize -2 --proxy-id 1
HYDRA_LAUNCH: /hydra/bin/hydra_pmi_proxy --control-port host0:38181 --rmk user --launcher manual --demux poll --pgid 0 --retries 10 --usize -2 --proxy-id 2
HYDRA_LAUNCH: /hydra/bin/hydra_pmi_proxy --control-port host0:38181 --rmk user --launcher manual --demux poll --pgid 0 --retries 10 --usize -2 --proxy-id 3
HYDRA_LAUNCH_END
I0523 09:06:19.669548 2410 sched.cpp:2009] Asked to stop the driver
I0523 09:06:19.669664 2410 sched.cpp:1191] Stopping framework 3dc9ee54-a23e-44a9-b7a9-354bb6e174c7-0038
Number of tasks= 10 My rank= 3 Running on host4
Number of tasks= 10 My rank= 0 Running on host2
Number of tasks= 10 My rank= 9 Running on host3
Number of tasks= 10 My rank= 7 Running on host4
Number of tasks= 10 My rank= 4 Running on host2
Number of tasks= 10 My rank= 8 Running on host2
Number of tasks= 10 My rank= 5 Running on host3
Number of tasks= 10 My rank= 6 Running on host1
Number of tasks= 10 My rank= 1 Running on host3
Number of tasks= 10 My rank= 2 Running on host1
[root@host0 mesos-hydra]#
```

图 3-13：输出样例

本文实现的 Scheduler 到此为止已经满足了文章开头提出的两大目标，即用户只需要输入一行命令，就可以完成在集群中一键部署 MPI 运行环境，并可以直接运行 MPI 任务，最后只在客户端汇集显示计算结果。同时，在集群做 MPI 任务计算的同时，也是可以交叉运行其他计算框架的。因此，集群中的每台主机，都可以最大程度地得到利用，避免粗粒度的资源分配造成的浪费。

本文的代码文件结构如图 3-14 所示。

在本文的 MPI 任务运行时，同时运行 marathon 等计算框架，如图 3-15 和 3-16 所示，可以看出 Mesos 完全可以使各种计算任务交错运行，同时存在。

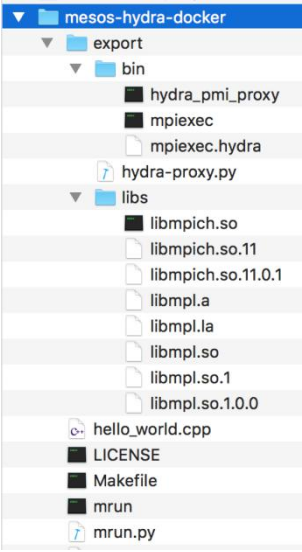


图 3-14：代码文件结构

Mesos

Frameworks

Agents

Roles

Offers

Maintenance

njlucs_mesos_cluster

Master

Frameworks

Active Frameworks

Find...

ID ▼	Host	User	Name	Roles	Principal	Active Tasks	CPU's	GPU's	Mem	Disk	Max Share	Registered	Re-Registered
...0eb64755fa7c-0163	n167.njlucs.cn	root	MPICH2 Hydra : integration	*		4	28	0	4.0 GB	0 B	35%	just now	-
...0eb64755fa7c-0162	n167.njlucs.cn	marathon	marathon	*		3	3	0	384 MB	0 B	3.75%	2 hours ago	3 minutes ago
...0eb64755fa7c-0000	n167.njlucs.cn	mesos	chronos	*		1	0.1	1	128 MB	256 MB	12.5%	yesterday	-

Inactive Frameworks

Find...

ID ▼	Host	User	Name	Roles	Principal	Active Tasks	CPU's	GPU's	Mem	Disk	Max Share	Registered	Re-Registered
------	------	------	------	-------	-----------	--------------	-------	-------	-----	------	-----------	------------	---------------

Completed Frameworks

Find...

ID ▼	Host	User	Name	Roles	Principal	Registered	Unregistered
...0eb64755fa7c-0161	n167.njlucs.cn	root	MPICH2 Hydra : integration	*		19 hours ago	19 hours ago
...0eb64755fa7c-0160	n167.njlucs.cn	root	MPICH2 Hydra : integration	*		19 hours ago	19 hours ago
...0eb64755fa7c-0159	n167.njlucs.cn	root	MPICH2 Hydra : integration	*		19 hours ago	19 hours ago
...0eb64755fa7c-0158	n167.njlucs.cn	root	MPICH2 Hydra : hello_world	*		19 hours ago	19 hours ago

图 3-15：Mesos 的任务混布特性体现（一）

MESOS

FrameworksAgentsRolesOffersMaintenance

njlucs_mesos_cluster

Master

17ae8591-ec62-4477-bbb9-0eb64755fa7c

Cluster: njlucs_mesos_cluster

Leader: n167:5050

Version: 1.6.0

Built: a week ago by centos

Started: 2 days ago

Elected: 2 days ago

Master Log: DownloadView

Agents

Activated4

Deactivated0

Unreachable0

Tasks

Staging0

Starting0

Running8

Unreachable0

Active Tasks

Find...

Framework ID	Task ID	Task Name	Role	State	Health	Started	Host	
...0eb64755fa7c-0164	2	task 2	*	RUNNING	-	just now	n169	Sandbox
...0eb64755fa7c-0164	0	task 0	*	RUNNING	-	just now	n167	Sandbox
...0eb64755fa7c-0164	1	task 1	*	RUNNING	-	just now	n168	Sandbox
...0eb64755fa7c-0164	3	task 3	*	RUNNING	-	just now	n170	Sandbox
...0eb64755fa7c-0162	nginx.83d19bc1-5f29-11e8-8cac-62c7e749a080	nginx	*	RUNNING	-	just now	n167	Sandbox
...0eb64755fa7c-0162	nginx.80d3c50e-5f29-11e8-8cac-62c7e749a080	nginx	*	RUNNING	-	just now	n167	Sandbox
...0eb64755fa7c-0162	nginx.35367835-5f29-11e8-8cac-62c7e749a080	nginx	*	RUNNING	-	3 minutes ago	n167	Sandbox
...0eb64755fa7c-0000	ct:1527148864538:0:test	ChronosTasktest	*	RUNNING	-	7 minutes ago	n170	Sandbox

Unreachable Tasks

Framework ID	Task ID	Task Name	Role	Started	Agent ID
No unreachable tasks.					

图 3-16：Mesos 的任务混布特性体现（二）

从以上两张图中可以看出，集群不仅运行了 MPI 的分布式任务，同时也有节点在运行 marathon、nginx 等框架等计算任务。这种交错运行的方式，可以使集群资源得到充分利用，不会出现机器长时间空闲等待的状况。

3.4 本章小结

本章主要介绍了本次研究工作的具体方案及解决方法。分为两个方面，一个是如何利用 Docker 容器封装 MPI 运行环境，另一个是如何使用 Mesos 平台对封装在容器中的 MPI 任务进行资源调度。

首先介绍了 Docker 制作镜的方法，然后针对容器间跨主机的应用场景，提出了一种使用共享 Overlay 网络、隔离用户工作空间的方案。

其次，介绍了 NFS 网络文件系统，本文采用其数据存储和共享的功能；最后，通过描述自定义 Mesos Scheduler，研究了对于有个性化需求的 Framework 应该如何编写、配置 Mesos 集群。

第四章 对比验证与实验

4.1 不使用 Docker 部署 MPI 任务

在本文工作实际进行前，为了熟悉 MPI 的使用过程，也做了许多相关的实验。其中，第一步做的就是在一个普通的集群中，不使用 Docker 镜像，直接部署运行 MPI 任务。

在不使用 Docker 镜像部署，通过直接安装 `mpich` 或者 `openmpi` 进行 MPI 任务的部署运行时，由于集群中机器数量较多，必须保证每个主机上安装的 MPI 实现厂商、版本以及安装目录都要完全相同，否则就会遇到一些因为无法适配的问题，导致任务无法正常运行。

在传统的高性能计算中，MPI 集群模式的工作方式是：每个 `worker` 机器上都需要开一个 `mpd` 守护进程，这个守护进程固定监听一个端口。`client` 通过 `mpirun/mpiexec` 提交任务时，会根据用户对节点数量和资源的要求，分配固定数量的节点给这个任务，然后发消息给这些节点上的 `mpd`，消息的内容是所要运行的任务的命令。这些节点上 `mpd` 会开子进程，进行计算，并返回计算结果到 `client` 进行汇集。

4.1.1 集群环境

集群一共使用三台主机，分别为 `apple@u171`、`apple@u194`、`apple@u36`，使用的操作系统都是 CentOS7。配置各个主机之间可以通过 SSH 互相访问，并实现了互相免密登陆。统一下载、编译、安装了 `mpich-3.2.1` 版本，并解压在相同的目录 `/home/apple` 下。

4.1.2 对照实验

在主机 `apple@u171` 的 `/home/apple` 目录下，新建 `test.c` 文件，按照 MPI 的 API 编写的代码如图 4-1 所示，这是一个在三台机器上运行 10 个 `hello world` 进程的小程序。另外，在每个主机的 `/home/apple` 目录下还要新建一个用以保存每个机器 IP 地址的文件，命名为 `host`，其中第一行是各自的本机 IP。


```

#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
    int rank;
    int size;
    int namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(0, 0);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Get_processor_name(processor_name, &namelen);
    printf("Hello World form process %d of %d is on %s\n",
           rank, size, processor_name);
    MPI_Finalize();
    return 0;
}

```

图 4-1: test.c 文件内容

再将 test.c 文件进行编译，生成一份可执行文件 test。生成可执行文件 test.c 之后，复制到集群中其他两台机器的相同目录下，然后执行 mpirun 命令，得到结果如下图 4-2 所示。

```

apple@u171:~$ mpirun -n 10 -f /home/apple/hosts ./test
Hello World form process 8 of 10 is on u36
Hello World form process 1 of 10 is on u194
Hello World form process 0 of 10 is on u171
Hello World form process 3 of 10 is on u171
Hello World form process 7 of 10 is on u194
Hello World form process 6 of 10 is on u171
Hello World form process 2 of 10 is on u36
Hello World form process 4 of 10 is on u194
Hello World form process 5 of 10 is on u36
Hello World form process 9 of 10 is on u171

```

图 4-2: 执行 MPI 任务之后的结果

在这个实验环境下，对 MPI 安装版本及程序目录都有苛刻要求。相比之下，假如替换前文研究的使用 Docker 封装的 MPI 运行环境，会方便很多，可以做到一键化部署运行 MPI 任务。

4.2 实验结论

Mesos 框架作为一个集群资源管理平台，它将整个数据中心的所有资源都整合成一个单一的池，以此来简化对于资源的配置。以本文为例，Mesos 就可以通过自定义框架对容器资源作出限制与筛选。另外，提供了一系列非常实用的容错基础设施，应用程序和基础设施可以弹性扩张。

Mesos 的独到之处还在于它可以单独地对各种不同的工作负载进行管理，不管是传统的应用程序，比如无状态的 Docker 微服务、批处理作业，还是实时分析有状态的分布式数据服务。它都可以通过自身的二级架构来对这些应用进行感知，从而做到调度。在本文的研究中，也是利用了 Mesos 的可自定义调度来灵活实现调度策略的。

在提供底层基础设施的同时，Mesos Master 还不忘保持隔离。每个工作负载都配有自己对应的调度器，深刻满足了对部署、缩放、升级的具体需求。应用程序调度器也可以被独立地开发管理，这就给 Mesos 框架不断地注入生命力，使得它始终保持高度可扩展性，随着时间的推移会支持更多更新的工作负载。

4.3 本章小结

本章首先做了不使用 Docker 封装 MPI 运行环境的对比实验，突出需要依靠 Docker 的可移植特性，来解决本文一开始提出的 MPI 版本在集群间不适配的问题。其次，本章介绍了 Mesos 作为集群资源管理的优势，在于可自定义调度策略与优秀的资源筛选限制策略。

第五章 总结与展望

5.1 工作总结

本文基于 Mesos 统一资源管理调度平台，以 Docker 容器为基本资源分割管理单位，部署运行 MPI 任务。根据 MPI 的特性与在实际 HPC 中应用的不足，结合 Docker 容器技术和 Mesos 资源框架的轻量级可扩展的优势，设计研究了这套系统的解决方案。

本文完成的主要贡献如下：

- 针对使用容器部署分布式并行计算任务的需求，选用 Docker 容器技术，以镜像封装任务运行时所需的环境。
- 针对在集群中运行 MPI 任务的需求，采用 Mesos 作为分布式资源管理框架，自定义 Mesos Scheduler 进行资源调度。
- 在以上两点研究的基础上，结合前期的参照实验，将本文提出的解决方案与现有概念进行对比，从不同角度验证该解决方案的可行性。

5.2 研究与展望

本文把 Docker 技术作为资源分配的基本载体单元，采用 Mesos 进行统一管理，实现了在 HPC 中一键部署应用 MPI 任务的目标。但是由于本人经验不足，精力和时间也有限，本文设计的方案仍有一定的不足之处需要改进。包括以下几点：

- 容器调度策略单一。
- 资源存储持久化欠缺。
- 资源碎片整理、优化不足。

在容器调度策略方面，可以深度挖掘调度策略，而不仅仅局限于最简单的贪心算法。其次，自定义 Mesos Framework 需要考虑的另一大问题就是如何保证资源存储持久化，是通过本地化数据亦或是别的方式，都是可以深入探讨的。最后，也是本文前后都不作考虑的内容，即如何优化资源碎片，将资源利用率提高至可用范围内的最高点，以期收益最大化。

参考文献

- [1] Forum M P. MPI: A Message-Passing Interface Standard[M]. University of Tennessee, 1994.
- [2] Fink J. Docker: a Software as a Service, Operating System-Level Virtualization Framework[J]. Code4lib Journal, 2014(25).
- [3] Merkel D. Docker: lightweight Linux containers for consistent development and deployment[J]. 2014, 2014(239).
- [4] Joy A M. Performance comparison between Linux containers and virtual machines[C]. Computer Engineering and Applications. IEEE, 2015:342-346.
- [5] 袁忠良. 容器云计算平台关键技术研究[D]. 南京大学, 2017.
- [6] Felter W, Ferreira A, Rajamony R, et al. An updated performance comparison of virtual machines and Linux containers[C]. IEEE International Symposium on PERFORMANCE Analysis of Systems and Software. IEEE, 2007:171-172.
- [7] Naik N. Building a virtual system of systems using docker swarm in multiple clouds[C]. IEEE International Symposium on Systems Engineering. IEEE, 2016:1-3.
- [8] Bernstein D. Containers and Cloud: From LXC to Docker to Kubernetes[J]. IEEE Cloud Computing, 2015, 1(3):81-84.
- [9] Saha P, Govindaraju M, Marru S, et al. Integrating Apache Airavata with Docker, Marathon, and Mesos[J]. Concurrency & Computation Practice & Experience, 2016, 28(7):1952-1959.
- [10] Hindman B, Konwinski A, Zaharia M, et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center[C]. Proceedings of the 8th USENIX conference on Networked systems design and implementation. USENIX Association, 2011:429-483.
- [11] Wilkes J, Wilkes J, Wilkes J, et al. Borg, Omega, and Kubernetes[J]. Communications of the Acm, 2016, 59(5):50-57.
- [12] Schwarzkopf M, Konwinski A, Abd-El-Malek M, et al. Omega: flexible, scalable schedulers for large compute clusters[M]. ACM, 2013.

- [13]C. Kniep, Containerization of High Performance Compute Workloads using Docker, doc.qnib.org, 2014.
- [14]Nguyen N, Bein D. Distributed MPI cluster with Docker Swarm mode[C]. Computing and Communication Workshop and Conference. IEEE, 2017:1-7.
- [15]刘国乐, 余彦峰. 浅析 Docker 容器技术[J]. 保密科学技术, 2017(10).
- [16]王绍伟, 于跃, 盛小波,等. 工具室信息化管理系统的开发研究[J]. 无线互联科技, 2017(3):50-53.
- [17]郑显义, 史岗, 孟丹. 系统安全隔离技术研究综述[J]. 计算机学报, 2017, 40(5):1057-1079.
- [18]张翰博, 倪明, 卢胜林. 基于 RBD 的 Docker 虚拟化技术性能优化研究[J]. 信息技术, 2016(8):125-129.
- [19]Sun J, Cai X, Wang X, et al. Docker Remote API Unauthorized Access Vulnerability Exploitation Tool[J]. Computer Systems & Applications, 2017.
- [20]孙建, 蔡翔, 王潇,等. Docker Remote API 未授权访问漏洞利用工具[J]. 计算机系统应用, 2017, 26(8):247-251.
- [21]席岩, 王磊, 张乃光,等. 面向媒体服务的云计算调度算法研究[J]. 广播电视信息, 2017(4):65-69.
- [22]Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters[M]. ACM, 2008.
- [23]Zaharia M, Chowdhury M, Franklin M J, et al. Spark: cluster computing with working sets[C]. Usenix Conference on Hot Topics in Cloud Computing. USENIX Association, 2010:10-10.
- [24]Neumeyer L, Robbins B, Nair A, et al. S4: Distributed Stream Computing Platform[C]. IEEE International Conference on Data Mining Workshops. IEEE, 2011:170-177.
- [25]李永峰, 周敏奇, 胡华梁. 集群资源统一管理和调度技术综述[J]. 华东师范大学学报(自然科学版), 2014, 2014(5):17-30.
- [26]Shepler S, Callaghan B, Robinson D, et al. NFS version 4 Protocol[J]. Rfc, 2000.
- [27]Vellaipandiyan S. Big Data Framework - Cluster Resource Management with Apache Mesos[J]. 2014, 3(4):228-294.

- [28] Abu-Dbai A, Breitgand D, Gershinsky G, et al. Enterprise Resource Management in Mesos Clusters[C]. ACM International on Systems and Storage Conference. ACM, 2016:17.
- [29] Bernstein D. Containers and Cloud: From LXC to Docker to Kubernetes[J]. IEEE Cloud Computing, 2015, 1(3):81-84.
- [30] Jannotti J, Gifford D K, Johnson K L, et al. Overcast:reliable multicasting with on overlay network[C]. Conference on Symposium on Operating System Design & Implementation. USENIX Association, 2000:14-14.
- [31] 冯明振. 基于 macvlan 的 Docker 容器网络系统的设计与实现[D]. 浙江大学, 2016.
- [32] 赵星月. 基于 Docker 容器技术的 Host 网络通信模式研究[J]. 科学与财富, 2017(6).

致谢

随着这篇论文的完稿，我的大学生涯也即将走到尽头。这一路走来，虽说有很多事情原本可以做的更好，但也是站在终点回头看的时候才知道，所以这样的青春也应是无悔了。记忆里入学时那个夏天的温度依旧灼热，四年来老师们的谆谆教导在上下课铃声的交替中纷飞。感谢系里对学生专业素质和品德的培养，感谢遇到的所有老师与同学，与我一起经历了这四年。站在尾声的我，不仅比当年收获了更多知识，也经过岁月的打磨，收获了心理上的成长。

此次研究工作能够顺利展开，与老师、同学、前辈以及家人的指导和鼓励是不可分割的。在此，我要特别感谢我的导师曹春老师。从一开始的论文选题、文献的阅读、系统的设计、结构、布局到最终的论文的撰写、修改、查重，每一个阶段曹春老师都认真负责地督促着我，尽心尽力。整个流程走下来我跟着曹春老师学到了很多知识，也明白了遇到困难及时与人沟通的重要性。再者还要特别感谢陈洁学长，我多次向他讨教问题，他也不厌其烦地一次次给我解答，为我的毕业设计以及毕业论文扫清诸多障碍，与师兄的交流使我获益匪浅。

最后，由于本人经验尚且不足，知识储备也比较浅薄，这篇论文存在着许多需要改进的地方。希望阅读本篇论文的老师 and 同学们可以多加指正，多提宝贵的意见，我将感激不尽！