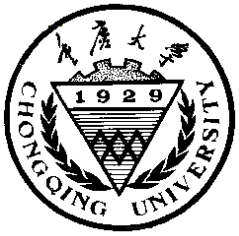
基于 Ceph 的
分布式异构存储系统数据处理优化研究



重庆大学硕士学位论文

（学术学位）

学生姓名：梁宇彤

指导教师：冯　亮　研究员

专　　业：计算机科学与技术

学科门类：工　学

重庆大学计算机学院

二 O 一八年四月

Optimizing Data Placement of MapReduce on Ceph-Based Framework under Load-Balancing Constraint

A Thesis Submitted to Chongqing University

in Partial Fulfillment of the Requirement for the

Master's Degree of Engineering

By

Yutong Liang

Supervised by Dr. Liang Feng

Specialty：Computer Science and Technology

College of Computer Science

Chongqing University, Chongqing, China

April, 2018

摘　要

近年来，互联网相关行业产生了大量由用户通过如社交网络、博客以及多媒体共享服务等应用生成的数据。研究界和工业界面临的共同挑战是如何设计一个低成本的存储系统来应对数据爆炸的问题。分布式对象存储系统作为一种常用的解决方案经常被用来在实际生产中存储大量的数据。而Ceph凭借自身的高可用性、可靠性和可扩展性，其作为一种分布式对象存储系统被广泛地应用在各个不同的行业领域。

数据的大规模增长导致的另一个问题是，生产环境中的多个集群之间可能在部署时存在先后顺序或者存在不同的功能需求，进一步造成集群间的异构问题，例如计算能力异构，存储异构，网络异构等情况。在异构环境下，如何对数据进行放置成为了一个热点问题，根据数据放置策略的不同，可以极大地影响系统的性能和集群之间的负载均衡。

目前 Ceph 中所采用的 CRUSH 算法更多关注的是数据的负载均衡问题，虽然通过 CRUSH Map，Ceph 允许用户可以自定义对象存储设备的权重，但这里的权重并没有真正体现出异构环境中不同设备之间的差异，而是仅仅反映了存储介质容量大小的区别，忽略了计算能力和网络等异构情况。在默认情况下，Ceph 中的数据将以一种近似平均的方式，被分配到各个对象存储设备上。由于没有考虑到集群之间可能存在的异构性所带来性能下降的问题，最终会导致整体运行时间变长。

针对上述存在的集群异构性的问题，本文提出了一种改进的 Ceph 架构。这个新的架构综合考虑了集群的负载均衡，以及包括节点计算能力和网络带宽在内的异构性。然后本文在改进的 Ceph 架构下，对分布式计算框架 MapReduce 进行研究。在将 MapReduce 迁移至改进的 Ceph 架构之后，如何在异构集群环境下确定初始数据的分配，找到最优的数据放置策略，从而最小化整个程序的完成时间是十分关键的。

因此，本文在保证负载均衡前提下，基于改进的 Ceph 架构和 MapReduce 的性质，首先提出了混合整数线性规划算法用来求得在异构环境下 Ceph 的最优数据放置策略。但是，MILP 算法的计算复杂度比较高，时间开销大，无法适用于实际的生产环境。所以本文设计了一种基于遗传算法改进的数据放置算法可以在很短的时间内求得问题的近似最优解。实验结果表明该算法得到的数据放置策略与传统 Ceph 中的策略相比，在性能方面可以提升高达 25.6%。

关键词：分布式系统，Ceph，异构性，数据放置，负载均衡。

ABSTRACT

In recent years, a large number of data has been generated by users through applications such as social networks, blogs, and multimedia sharing services. The challenge for the research and industrial community is how to design a low-cost storage system to deal with the explosion of data. As a common solution, the distributed object storage system is often used to store large amounts of data in the actual production.

Ceph, as a distributed object storage system, has been widely used as the basic storage for distributed systems due to its high availability, reliability and scalability. Strategies of data placements in Ceph composed of heterogeneous clusters can greatly affect the system performance and load balancing.

At present, the algorithm which is used in Ceph only focuses on the load balancing of the system. Through the CRUSH Map, Ceph allows users to customize the weight of object storage devices, but the weight doesn't reflect the difference between devices in a heterogeneous environment. It only indicates the different size of the storage, without considering the computing capability and the network heterogeneity. By default, the data will be assigned to each object storage device in an approximate average way. Because there is no consideration of heterogeneities among clusters, it will eventually cause the performance degradation and lead to a longer running time.

Based on the heterogeneous cluster environment, this paper presented an improved Ceph framework. This new framework takes the load balancing and the cluster heterogeneity, including the computing power and the network bandwidth into account. In addition, for a given application, how to determine the initial data allocation and find the best data placement strategy to minimize the completion time of the whole application in heterogeneous cluster environment is very critical.

This paper focuses on how to allocate the data after migrating to the proposed Ceph-based framework. This paper chooses MapReduce, a distributed computing framework, as a use case because MapReduce is still widely used and also the presented framework is suitable for the applications based on the principle of moving computation rather than data across clusters, such as MapReduce.

According to the proposed Ceph-based framework and the properties of MapReduce, we formulate the mixed integer linear programming (MILP) to obtain the optimal data placement. However, because of the large computational complexity of MILP, we devise

an efficient algorithm to obtain the near-optimal solutions. The experimental results show that the proposed algorithm can achieve up to 25.6% improvement on system performance, compared with the original strategy implemented in Ceph.

Keywords: distributed system, Ceph, heterogeneity, data placement, load balancing.

目 录

1 绪 论

1.1 背景介绍

近年来，随着全球数据量指数级别的增长，人们对于计算性能以及存储性能的要求开始不断提高。传统的集中式系统，即将计算、存储集中在一台主机的系统，受到 CPU 和存储介质发展速度的限制，已经无法适应数据增长的速度。因此，人们转而将计算和存储分散到多台主机，使得每台主机都可以负责部分的数据计算和存储任务，继而实现分布式计算和分布式存储。

2003 年，Google 在国际顶级会议 SOSP 上发表重量级论文《The Google File System》[1]，随后相继在 2004 年的 OSDI 和 2006 年的 OSDI 上发表《MapReduce: Simplified Data Processing on Large Clusters》[2]以及《Bigtable: A Distributed Storage System for Structured Data》[3]两篇重量级论文，合称大数据时代的“三驾马车”，标志着大数据时代的到来。

也因此，工业界和开源社区诞生了许多非常优秀的项目，最著名的莫过于 Apache 开源项目 Hadoop[4]。Hadoop 主要由两部分组成：分布式计算框架 MapReduce[2]和分布式文件系统 HDFS[5]。Hadoop 作为 Apache 顶级的开源项目，2004 年由 Doug Cutting 开始实施，至今已经有 14 年的历史。在此期间，Hadoop 取得了很多令人惊叹的成就并多次刷新大数据排序记录。但是，数据量的暴增以及随着业务的要求越来越高，使得 Hadoop 的底层存储 HDFS 暴露出来很多架构设计之初就存在的问题，如单点故障，不适用于小文件存储，仅支持 Append 操作，存储空间的利用率低以及扩展性差等问题。

为了解决 HDFS 中遇到的各种痛点，出现了许多关于替代 HDFS 相关的研究和项目，例如：①Cassandra[6]。Cassandra 是一个开源的 NoSQL 键值存储系统，通过与 Hadoop 进行整合，使得网络应用可以达到快速访问数据和处理的功能。但是 Cassandra 稳定性较差，在 Facebook 内部测试时，常常会出现一些严重的问题；②Lustre[7]。它同样是一个开源的文件系统，常被用在大型的集群中或者超算领域，相比于 HDFS 在速度方面的提升，Lustre 在使用成本上面也更为低廉，但是由于 Lustre 采用的是集中式的架构，所以与 HDFS 一样，存在单点故障的问题；③GPFS[8]。它由 IBM 进行开发并且维护，并通过与 Hadoop 整合，在应用程序执行速度方面远超 HDFS，因为其相比 HDFS 而言，属于内核级别的文件系统，不需要经由操作系统提供的 EXT4 文件系统。而缺点是同样为集中式的架构并且代码没有开源，使用成本较高；④MapR File System[9]。MapR 是 Hadoop 最为出名的商业发行版之一。其底层的文件系统相比 HDFS 有更优秀的性能，并且支持快照和镜像功能，

在近年来，收到大量用户的追捧。缺点方面与 GPFS 相同，并且同样属于闭源项目，需要花费高昂的价格购买使用。

而本文将对另一种替代方案，分布式对象存储系统 Ceph 进行具体的研究。相比于上述几种替代方案，Ceph 有如下几种优点：

①去中心化。Ceph 不需要依赖主（Master）节点来保证系统整体的运行，也因此不存在单点故障的问题。

②空间利用率。Ceph 支持通过纠删码的方式对文件进行备份，相比于采用传统多副本的方式，可以有效地提高空间利用率。

③动态扩容。Ceph 通过两层映射机制，使得添加节点以及删除节点的操作对于客户端而言是透明的。

④镜像和多级别快照。Ceph 同样支持镜像和快照的功能，并且 Ceph 提供两种级别的快照，一种是池（Pool）级别，另一种是 RBD（RADOS Block Devices）级别的快照。

⑤多种存储接口。Ceph 作为一个先进的分布式存储解决方案，支持对象存储，文件存储，和块存储三种接口[10]，具体如下：1）对象存储主要指通过利用 C，C++，Java，Python，PHP 等语言编写程序，直接利用 Ceph 的库 librados 来进行数据访问，也可以通过 RESTful 的接口进行访问；2）文件存储是将 Ceph 以文件系统的方式挂载并兼容 POSIX 的接口，以文件系统的方式进行访问；3）块存储则是将 Ceph 作为块设备直接挂载，提供类似读磁盘的方式进行访问。

⑥开源。Ceph 是一个开源项目，也就意味着使用和学习成本较低，并且 Ceph 拥有较为完善的使用和开发文档以及良好的社区支持。

之所以选择 Ceph 作为研究对象的另一个原因是因为 Ceph 在近两年的发展十分迅速，也使得对象存储的概念重新活跃了起来，逐渐出现在各大云存储厂商的产品列表中。除此之外，Ceph 也被广泛应用在很多企业的生产系统中，例如 Cisco，Bloomberg，CERN，还有国内的华为，360 等都在利用 Ceph 作为底层的存储支撑上层应用对外提供快速稳定的无故障服务。

1.2 国内外发展现状

本章节重点介绍分布式对象存储 Ceph 和分布式计算框架 MapReduce 的一些发展现状，以及相关热点问题的发展现状。

1.2.1 分布式对象存储 Ceph 发展现状

据预测，至 2025 年，世界上总的数据量将会攀升至 163ZB，其中绝大部分为非结构化的数据，如视频，音频，图片等[11]。传统分布式文件系统在存储非结构化的数据时，往往受限于元数据的扩展问题。所以，分布式对象存储凭借自身扁

平化的元数据组织方式，在扩展性方面更加灵活，因而更加符合海量数据对于存储的需求。

Ceph 作为分布式对象存储的典型代表，自然也拥有了许多对象存储的优点。如：①扩展性。考虑到纵向扩展面临的问题，Ceph 利用商用硬件进行水平扩展，放弃使用传统文件存储所采用的目录结构，即分层的组织方式，转而使用扁平的组织结构方式，令其元数据有很好的扩展性，可以支撑大规模的数据存储；②小文件存储。许多分布式文件系统在存储时，都采用分片的方式对文件进行处理，而比如 HDFS 这类的分布式文件系统专门被设计用于海量大文件的存储，所以在存储过程中首先会对文件进行分片，通常会以 128M 或 256M 进行切分，这就导致在处理小文件时，会造成空间浪费。相比而言，对象存储通常可以很好的处理小文件的存储；③易维护。不需要维护多余元数据的层次结构，仅仅关注数据自身相关的元数据。

Ceph 来源于 Sage Weil 的博士研究课题，于 2004 年开始开发，2010 年被加入 Linux 内核中（v2.6.34）。整个项目托管在 Github 上面，主要由 RedHat 进行维护。目前已经累计超过了 84000 次代码提交，先后经历了 250 个版本，超过 650 人参与开发。

Ceph 通过其开放的社区和插件化的代码架构吸引了越来越多的底层厂商，包括 Mellanox，希捷以及 Intel 等都在积极的参与其中，利用自身的优势来持续对 Ceph 软件体系进行优化[12]。此外，Ceph 在存储后端方面，还以插件的方式提供多种存储引擎的支持，如 KVStore，BlueStore 等。

另外，近年来关于 Ceph 也有许多相关的研究。2015 年，Michael A. Sevilla 等人针对 Ceph 系统中对数据进行迁移时，数量，位置，方式等难以控制的问题，实现了可编程元数据存储系统，让用户可以按照自己的需求逻辑实现负载均衡[13]。2016 年，Ke Zhan 等人通过以命令行或者 librados 的方式，实现了多线程的文件读写算法，针对不同大小的文件使用不同的线程模型对读写进行优化[14]。Jiayuan Zhang 等人针对 Ceph 中的一致性模型无法很好满足用户需要的低延迟写操作问题，提出在备份节点使用异步的方式对数据进行同步[15]。

1.2.2 分布式计算框架 MapReduce 发展现状

分布式计算的出现是为了解决随着数据量的爆炸增长，在处理大量数据时，无法把所有的数据一次性装载进内存，进而造成计算成本高的问题。而现在通常服务器的内存在 256G 左右，所以当需要处理的数据远远超过这个数据量之后，如果只是以单纯地对内存进行扩充或者多次通过在单机上置换内外存的方式来解决，那么会存在一些问题。前者扩充内存的成本相对较高，而后者在运算时间上会有

较大的代价。这就使得需要通过某种运算机制，来将计算任务切分并下发到多个节点，让每台存储任务数据的机器都可以参与计算。

MapReduce 作为分布式计算框架的典型代表，就是利用“分而治之”的思想，将任务进行切分，然后分发到各个数据所在的存储节点上进行处理，然后将处理的结果进行汇总，从而可以很好的让大量的数据处理任务并发执行，减少任务的执行时间，并且降低了对机器性能的要求。

尽管 MapReduce 已经发展了十余年的时间，也陆续有一些新的分布式计算模型被提出，如 Spark[16]，Flink[17]，Storm[18]，Beam[19]以及 Google 最新的分布式计算模型 Cloud Dataflow[20]。但是之所以本文选用 MapReduce 作为研究对象，是因为 MapReduce 依然被很多所属不同领域的公司使用，并在学术界，持续不断地有许多学者对它进行研究和优化，虽然整体呈现下滑的趋势，但平均每年仍有两三百篇与之相关的论文发表。

相关论文的研究主要大致可以分为几个方面：①性能方面。2014 年，Zhuo Tang 等人优化 MapReduce 调度的算法，该算法分为任务分类和任务分配两个阶段，首先将任务分为计算密集型和 IO 密集型两类，然后根据本地性原则对任务进行分配，从而提升在异构环境下任务分配调度的性能[21]。2015 年，Inigo Goiri 等人提出了将近似计算和 MapReduce 的框架进行融合，旨在减少任务时间开销和能耗问题[22]。2016 年，Wenhong Tian 等人利用经典的 Johnson 模型，设计了一种调度器，将 MapReduce 中综合了 Johnson 算法的特性，从而有效地缩短了多任务的最晚完成时间[23]。2017 年，Dazhao Cheng 等人实现了一个跨平台资源调度的中间件，用以提升在使用 YARN 的情况下，MapReduce 的资源利用率和性能[24]。同年，Dazhao Cheng 等人还提出了一个自适应的任务调优方法，会自动为运行在不同节点的每个任务的配置文件选择最优的配置方案，从而提高任务执行效率[25]；②安全性方面。2018 年，Yao Dong 等人针对云环境下上传数据时可能存在安全隐患，对 MapReduce 进行改进，使得其可以直接处理加密的数据[26]。而 Kerim Yasin Oktay 针对混合云环境下，可能将敏感数据暴露给公有机器所导致的安全问题。提出了一种方法，将放置在公有机器上的生成的结果根据是否为敏感数据分别 Shuffle 到公有和私有机器上，保证公有机器不会有敏感数据[27]；③可靠性方面。2015 年，Yandong Wang 等人提出了一种新的容错框架，主要通过对关键日志进行分析和快速推测迁移错误任务，可以有效的降低由于故障导致的多余开销，并且能够在不同的情境下对失败的任务进行快速处理[28]。2017 年，Huansong Fu 等人针对 MapReduce 任务执行推断机制的故障问题，提出了一种错误感知的推断模式和任务分配策略，用于减少任务执行过程中由于故障造成的时间开销[29]。Mohammad Tanvir Rahman 等人对不同故障场景下，使用不同参数时，两种典型的 MapReduce 应用 WordCount 和

Stack Exchange 的性能作出评测，并提出了多种选项在 MapReduce 中注入错误，以模拟真实的故障问题[30]；③灵活性方面。2015 年，Emilio Coppa 等人针对数据倾斜和负载不均衡可能导致执行时间延长的问题，设计了一个指示器，用于在云环境下对数据处理进行预测，减少用户按需付费的成本[31]。2016 年，Seyed Morteza Nabavinejad 等人提出了一种根据实际情况选择最适合的虚拟机配置的方法，用于提高性能并且减少成本[32]。

1.3 论文研究意义

以往主流的分布式存储通常是文件存储，文件存储主要是以层级的方式对文件进行存储，通过维护一个目录树，使用户根据特定路径来访问具体位置的文件。但文件存储所带来的问题就是基于层级的处理方式的扩展性不够好，另外的一个问题就是元数据的爆炸增长，导致操作文件性能的降低。而基于块的阵列方式，如 RAID，由于携带大量的保护信息，随着数据量的不断增多，也同样不利于扩展。所以分布式对象存储凭借易扩展性，加上元数据是以扁平化的方式进行管理，有效地解决了层级方式管理的弊端。部分分布式对象存储系统还支持纠删码的方式对数据进行备份，一方面保证了数据的安全，另一方面降低了存储的空间，提高了利用率。

而分布式对象存储中最具有代表性的就是 Ceph，它是一个去中心化的分布式对象存储系统，旨在实现高可用性，可扩展性和可靠性。它可以在 Hadoop 生态环境中替代 HDFS 作为底层文件系统，从而使得如 MapReduce 这样的分布式计算框架运行在其上，并可以表现出很好的性能。传统的 Ceph 使用 CRUSH 算法[33]对数据进行分配，它可以有效地解决系统内部集群之间负载均衡的问题。然而，CRUSH 算法并没有充分利用存储设备和网络特性的不同，忽略这些不同的特性可能会使得整个系统存在过多闲置资源或者部分任务无法获取所需资源，最终将导致应用程序的执行时间变长。

由于存储设备计算能力之间的差异，不同的数据放置策略将严重影响系统整体性能[34]。此外，在通过改变数据放置策略，提高系统性能的同时，还需要考虑网络的异构性，特别是在跨地域设备间传输数据的网络延迟，避免造成网络拥塞。并且，如果单纯追求性能而采用极端的数据放置策略，势必将导致整个系统的负载不均衡。因此，在保证系统负载均衡的前提下，如何在异构集群中优化数据放置策略已经成为分布式系统领域的一个重点问题。

现有的一些研究工作，在优化 MapReduce 数据分配时，并没有考虑到负载均衡的问题[35-39]。另外，目前在 Ceph 架构下，还没有相关优化 MapReduce 的数据放

置问题的文章。本文旨在寻找一种数据放置策略，可以使得在 Ceph 架构下的 MapReduce 应用在满足负载均衡的同时，可以最小化执行时间。

1.4 论文主要研究内容

在本文中，针对实际生产中可能遇到的问题，提出了一种基于 Ceph 改进的分布式架构。在这个前提下，使用 MapReduce 作为研究背景，对如何进行数据分配进行了研究。在这个架构中，存储设备按照地理位置划分为不同的集群。由于不同地理位置之间的网络条件可能不同，进一步数据传输速率也就不同，所以网络被认为是异构的。这个架构在分配数据时，使用了一种新的机制，通过一张概率表将数据分配给存储设备，本文称为“面向性能的分区”机制。在原有的 Ceph 基础上增加了一层哈希，用于被存储的数据确定其对应的某个具体集群。概率表中的每一个分区都与一个特定的集群绑定。如果计算结果落在概率表中的某一分区，则会将数据分配给与这一分区所绑定的集群。因此，通过合理调整概率表，从而改变分布式对象存储系统 Ceph 中的数据放置策略，以达到优化 MapReduce 执行效率的目的。

为了求出最优的数据放置策略，本文首先定义了所要研究的问题模型，接着根据模型，利用混合整数线性规划（Mixed Interger Linear Programming，MILP）算法对问题求解。然而，MILP 算法的计算复杂性随着问题的规模增大而呈现指数级别地增长，例如，在求解将 9 个 Map 任务分配到 18 个存储设备上的最优数据放置策略时，利用 MILP 算法无法在 10 个小时之内求出最优解。所以为使研究的内容更符合实际生产的情景，本文基于遗传算法提出了一个改进的快速算法，用来求得问题的近似解决方案。由实验结果可以看出，本文提出的算法在较大数据量的情况下，能够在几分钟内快速对问题求解。最后通过 Hadoop 基准测试，将基于遗传算法改进的数据放置（Data Placement Based on Genetic Algorithm，DPGA）算法得到的放置策略与 Ceph 默认使用的平均分配（Average）算法还有另一种基于性能优先的贪心（Greedy）算法对比，整个任务的完成时间分别平均减少了 24.9%和 11.4%。本文的主要工作总结为以下几点：①在异构集群环境中，提出了一种基于 Ceph 改进的架构，专门针对如 MapReduce 这样的属于计算迁移类型的应用；②在相对负载均衡的前提条件下，对数据放置问题建模并给出问题定义；③提出了 MILP 算法，并在满足负载均衡约束下，求得数据放置的最优策略；④当问题的规模变大时，提出了一个基于遗传算法改进的算法来获得待求解问题的近似最优解。

本文的部分研究成果已经发表在 ICPADS 2016（CCF C 类）国际会议上。

1.5 论文组织结构

本文的主要工作是通过对现有分布式解决方案 Ceph 进行研究并总结，发现其中的问题。然后提出一种基于 Ceph 改进的架构，并针对 MapReduce 在改进的架构下的数据分配问题提出优化算法，对原有的数据放置策略进行优化。其余的论文组织如下。

第一章为绪论部分。主要介绍了问题的背景环境，并针对本文具体使用到的 Ceph 以及 MapReduce 的研究现状进行了总结，最后说明了本文具体的研究意义，并给出重点的研究内容。

第二章为背景技术分析。介绍了相关的一些技术背景，比如 Ceph 的架构，读写流程，CRUSH 算法，以及 MapReduce 计算框架等。

第三章为基于 Ceph 改进架构的设计。针对存储节点存在的异构性，网络的异构性，以及考虑到实际生产中可能面临的灾备问题，提出了一种基于 Ceph 的改进架构，接着根据改进架构的特性，提出了面向性能优化的分区机制用以对数据进行分配。

第四章为优化数据分配问题的定义与算法。首先，给出数据分配问题的定义并利用改进架构的特点和 MapReduce 的性质对问题进行建模。然后针对问题定义给出的模型，提出了一种基于 MILP 的算法，给出了详细的算法公式，并能通过该算法求得问题最优解。接着考虑到 MILP 算法的局限性，提出了另一种基于遗传算法的近似算法。

第五章为实验方案结果及分析。首先给出了针对改进架构下 MapReduce 数据分配的实验方案。然后对所提出的两种优化算法的求解效率进行对比，接着选择了效率较高的基于遗传算法的近似算法和传统 Ceph 中使用的 Average 算法以及另外一种基于性能优先的 Greedy 算法进行性能对比，并对结果进行分析。

第六章为总结与展望。主要对本文的研究内容做出总结，并针对研究过程之中的不足之处和未来可扩展的研究方向给出简要说明。

2 背景技术分析

本章大致可以分两部分内容，首先是关于 Ceph 相关概念的介绍，其次是 MapReduce 相关概念的介绍。

2.1 Ceph 架构

Ceph 试图实现高效的负载均衡策略以防止热点问题，即热点节点比其他节点承受更高的负荷。实质上，Ceph 会通过一种伪随机算法将数据的存储分散在集群中的所有主机之上。此外，Ceph 强调高可用性，允许少量节点发生故障，而不会影响数据完整性和可访问性。

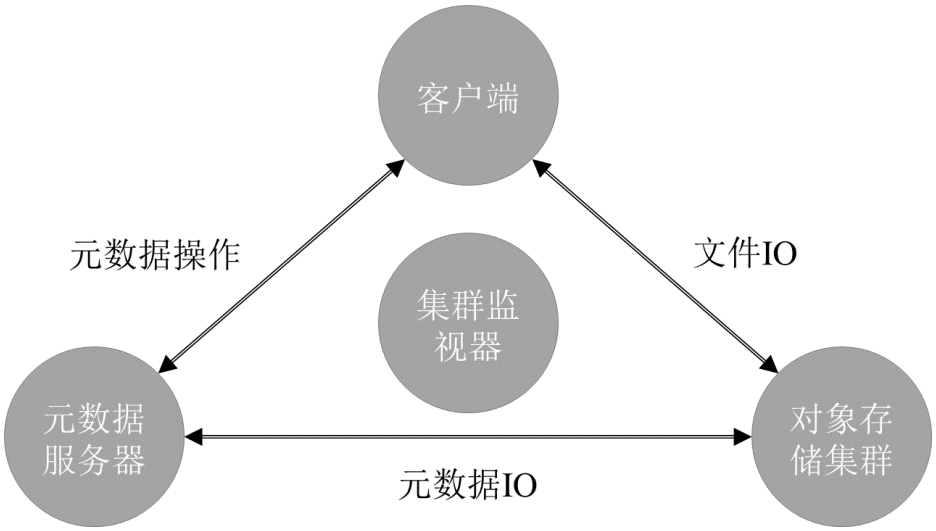


图 2.1 Ceph 的组件

Figure 2.1 The components of Ceph

如图 2.1，Ceph 的组件主要包含如下几部分：

①客户端。在实际生产过程中，绝大多数情况下，用户不会直接对对象进行读写。所以 Ceph 提供了三种接口方便用户对集群进行操作，如图 2.2。

块设备接口。由于块设备接口的发展日趋成熟，Ceph 可以利用虚拟块设备与海量数据存储系统实现交互。Ceph 的块设备接口提供了具有可扩展性的高性能内核模块，或者是针对 KVM（如 QEMU）以及基于云计算系统的集成，如依靠 libvirt，或者是 QEMU 的 OpenStack 以及 CloudStack。

文件系统接口。也称作 CephFS，是一个利用 RADOS 作为底层存储的文件系统，并且它可以兼容 POSIX 接口[40]。Linux 为 CephFS 提供了原生的驱动支持，所

以，CephFS 可以在任何类 Linux 的系统下被使用。CephFS 将数据和元数据分开存储，也因此使得运行在其之上的应用程序的性能和可靠性得到了提升。

而在 Ceph 集群内部，CephFS 需要利用一个链接库 libcephfs 来访问 RADOS，而且为了使用 CephFS 访问底层存储集群，必须要在集群中至少设置一个节点，用以部署元数据服务进程。

对象网关接口。它提供了与 OpenStack Swift 和 Amazon S3 相兼容的接口，并且由于 S3 和 Swift API 共享一个公共命名空间，因此可以用一个 API 写入数据，并用另一个 API 检索它[41]。

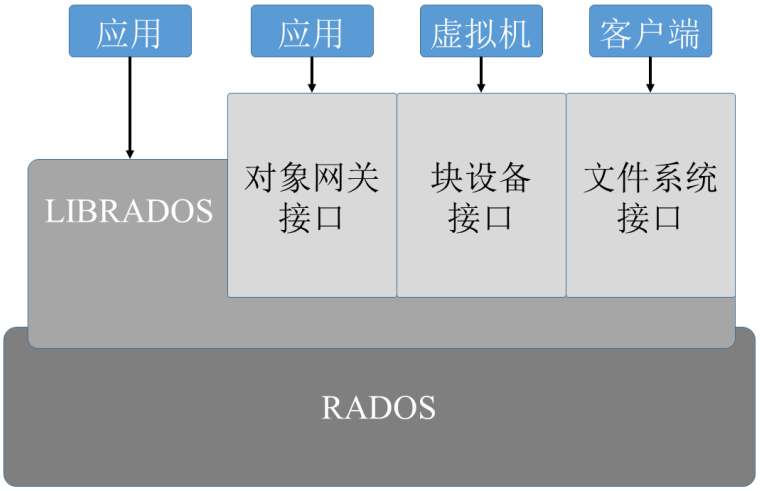


图 2.2 Ceph 提供的访问方式

Figure 2.2 The service interfaces of Ceph

②元数据服务器。当使用 CephFS 对底层存储进行访问时，它主要用来维护 Ceph 文件系统的命名空间。元数据服务器中的元数据和相应具体的数据都被存储在对象存储集群，因此，具有很好的可扩展性。为了避免出现热点问题，元数据服务器通过内部的动态子树算法使得系统的命名空间可以动态调整。

③集群监视器。它是一组实例，基于 Paxos 算法实现，用以维护一份集群的元信息 Cluster Map。通过与集群监视器进行通信，获取 Cluster Map，客户端可以获得各个集群组件的位置。并且客户端可以将 Cluster Map 作为输入，利用 CRUSH 算法进行运算，得到任何对象所对应的节点的位置。

④对象存储集群。是 Ceph 中实际存储数据的地方，由多个对象存储设备（OSD）构成，负责存储对象，并通过网络提供对它们的访问。既可以利用对象存储设备的 CPU 和内存进行数据复制，同时还支持负载均衡、故障恢复、监控和报告等功能。

2.2 Ceph 读写流程

Ceph 的设计采用了可以管理对象分布、复制和迁移的分布式对象存储服务，RADOS（可靠的自主分布式对象存储）[42]。Ceph 使用智能的对象存储设备取代了传统的存储（例如，磁盘或者 RAID）。图 2.3 是传统 Ceph 对数据进行分配的基本过程。一个 Ceph 集群通常拥有大量的 OSD 且每个对象都可能被分配到任意一个 OSD 上。

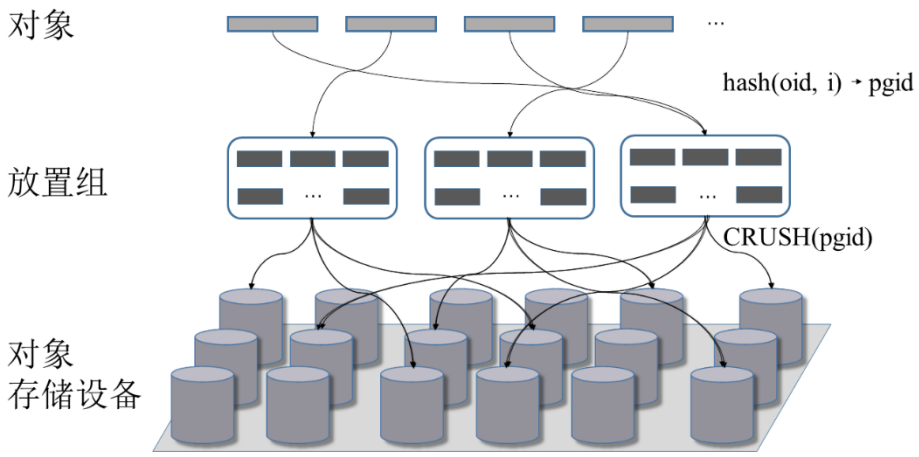


图 2.3 传统 Ceph 读写流程

Figure 2.3 The read and write of traditional Ceph

对于读写操作而言，客户端首先与 Ceph 监视器进行通信，获取集群的拓扑图，集群的拓扑图可以帮助客户端了解整个集群的配置和状态。然后将所需要存储的数据分割成许多对象进行处理，每个对象都有一个池 ID 和对象 ID，池是 Ceph 存储数据时用到的逻辑分区，每个池有自己的数据放置规则，比如可以通过定义一个池，专门用作将数据存放在快速存储设备上，如 SSD。也可以针对数据访问频率的特性，建立对应的池，将冷热数据分开存放在相应的存储设备上。接着根据对象 ID（OID），Ceph 将这些对象通过 Hash 函数映射到逻辑的分组，称为放置组（PG）。PG 通过在对象之上抽象出来一层，可以有效降低数据管理的难度，还可以在对象数量特别大的情况下，防止恢复时需要遍历每一个 OSD 上的对象，从而降低故障处理的难度。这些 PG 根据放置组 ID（PGID）唯一确定。对于每个放置组，通过利用 CRUSH 算法来决定存放数据的主节点的位置，在得到 OSD ID 之后，客户端通过直接和这个 OSD 通信，并对数据进行存储。当数据被写入到主节点后，再对副本进行写入[33]。

Ceph 的客户端将拥有诸多不同对象的存储集群看作是一个整体的对象存储系统或者是命名空间。因为没有 Master 节点，即完全去中心化，客户端可以直接对 OSD 发出读写请求。因此，这种设计在处理大量并发的请求时，不会对集群整体的性能造成影响。

2.3 CRUSH 算法

一般意义上的存储，大致包含两个部分，分别是数据存储及其元数据存储。元数据是关于数据的信息，如数据存放的位置，数据的大小以及数据相关的时间信息等，它描述了数据本身的属性。在传统的分布式系统中，当将新数据被添加到存储系统时，它的元数据首先会被更新，相应的实际存储数据的物理位置会在 Master 节点写入，然后再对具体数据进行存储。这个过程在数据量只有 GB 或者 TB 级别的时候被实践证明是完全可行的。然而，在存储 PB 或 EB 级别的数据时，上述方法就不再适用。此外，这种存储方式还面临着可能存在的单点故障问题，一旦 Master 节点出现故障，那么就会造成元数据的丢失，进而丢失全部数据。因此，无论是通过维护单个 Master 节点上的多个副本，还是复制整个数据和元数据，以确保更高程度的容错性，从而使元数据免受灾难影响都是至关重要的。往往会因为没有高效合理地对元数据进行管理，而导致存储系统产生瓶颈，使得系统在可用性或者性能上大幅度降低。

Ceph 在数据的存储和管理方面使用了具有革命意义的 CRUSH（Controlled Replication Under Scalable Hashing）算法，CRUSH 算法是一种智能的数据分布机制。它是 Ceph 的整个数据存储机制的核心。传统的存储系统通常需要依赖于一个中心节点，节点上存放有用于存储系统管理所需要的元数据或者索引表。不同于传统系统，Ceph 利用 CRUSH 算法计算，确定数据应该被写入到哪里或者从哪里读取。区别于传统的存储元数据的方式，Ceph 通过计算来确定数据的位置，从而消除了以传统方式存储元数据所遇到的所有限制。CRUSH 只有在需要时才会执行，而现在的商用机器已经可以快速和高效的使用 CRUSH 来对数据进行查找。

CRUSH 算法本质上是一个伪随机算法，对于给定的输入而言，如$pgid$，$cluster\ map$（集群的拓扑图），$placement\ policy$（用户自定义的放置规则）等，通过多参数哈希算法，得到具有可靠映射关系的输出结果，即一组 OSD 列表。由于算法本身的特性，对象和存储设备之间的映射关系不是显式相关的。

用户可以自定义权重来决定对象的分布情况，除此以外，用户还可以制定数据的放置策略，比如副本个数，以及副本放置的要求（如副本不能放在同一机架）等。集群拓扑图除了对集群的属性进行描述以外（如机柜数量，机架数量，磁盘数量，以及三者对应的关系），还包含了集群的资源描述，包括磁盘空间等等。

可以用一个简单的式子来描述 CRUSH 算法，如下所示：

$$\mathrm{CRUSH}(pgid,\ cluster\ \ map,\ placement\ \ policy) = (osd_0,\ osd_1, \ldots,\ osd_n)$$

其中n表示副本个数，$cluster\ \ map$由 device 和 bucket 构成，device 对应于具体的 OSD 设备，而 bucket 可以包含任意的 bucket 或者是 device，bucket 有四种不同的类型，如 uniform bucket，list bucket，tree bucket 和 straw bucket[33]。每种类型使用的数据结构不同，主要是针对性能和效率方面做的不同折中方案并且对应于不同的选择算法。每个 device 以及 bucket 都有自己的 ID 以及权重，目前 Ceph 中 device 的权重大小仅由存储设备的容量大小决定，而 bucket 的权重则是由所包含的所有 bucket 和 device 的权重的和决定[33]。

2.4 MapReduce 计算框架

MapReduce 是一个大规模并行处理的编程模型，使用大量的计算节点对数据进行处理，而这些节点根据拓补结构可分为集群和网格。集群指的是节点使用相似的硬件配置并共享同一个局域网，网格则指的是节点硬件配置多为异构，并且分布在不同的地理位置。MapReduce 背后的核心思想是首先将结构化或者非结构化的数据转化为键值对，然后对于所有键值对中相同的键进行合并的操作。除此之外，它可以很好的利用数据本地化的优点，在距离数据最近的节点上进行处理，以便减少通信带来的开销。

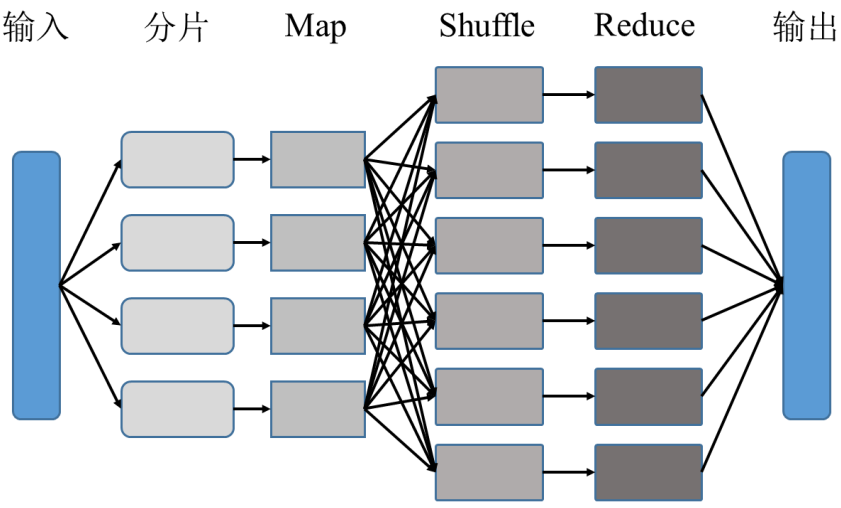


图 2.4 MapReduce 计算框架

Figure 2.4 The framework of MapReduce

具体任务流程如图 2.4 所示，MapReduce 提交任务时，首先需要计算作业的分片，也就是对任务根据配置文件制定的块大小进行切分，然后将分片和具体任务的 jar 包拷贝到分布式存储集群中。主要包含三个阶段，分别是 Map 阶段，Shuffle 阶段以及 Reduce 阶段，分别对应的是 Map 函数，Partition 函数以及 Reduce 函数。具体而言，在 Map 阶段，Map 函数获取一个键值对作为输入，并对输入进行处理，输出任意多数量的键值对。Map 阶段不做任何聚集操作，而是交由 Reduce 阶段。Map 阶段的主要目的是，提取用户感兴趣的数据并将数据建模成键值对集合。

在 Shuffle 阶段，通过自定义 Partition 或者使用系统默认实现的 Partition 函数，实现将 Map 经过处理得到的键值对集合分发到 Reduce 节点上[2]。简单来说就是键值对集合里面含有相同键的键值对被分成同一组并被发送到同一台机器上执行 Reduce 操作。

Reduce 阶段首先获取一系列的键值对，接着根据用户自定义的规则，将这些含有相同键的键值对进行合并。然后将合并后的结果写入到分布式存储系统中，整个任务就算执行完成。

MapReduce 适用于包括排序，构建倒排索引，URL 访问率统计，科学计算，聚类等大规模离线批处理任务，并不适合于实时性较高的计算问题。

2.5 本章小结

本章主要对后文可能涉及到的一些概念以及技术背景做简要介绍。首先介绍了 Ceph 的架构，并对读写流程进行介绍，然后将读写流程中最为重要的 CRUSH 算法做了说明。最后，说明了什么是 MapReduce 和 MapReduce 是怎样工作的。

3 基于 Ceph 改进架构的设计

本章拟针对 MapReduce 的特性，提出基于 Ceph 改进的分布式对象存储架构。传统并行计算主要将多个计算节点与外部存储相互连接，比如通过利用 SAN 来提供大规模的存储，但是 MapReduce 为了减少大规模并行计算所带来的网络上的 IO 开销，采用了将计算进行迁移，而不是数据迁移的方式。在将计算分发到存储节点后，会首先尽可能的计算本地的数据，然后才会利用就近原则计算其他存储节点上的数据。

本文所提出的架构旨在充分利用数据本地化的特性，来对 Ceph 进行优化。具体主要在两方面与传统的 Ceph 不同。首先，假设整个存储系统是由多个异构的集群组成。每个集群都由不同计算能力的存储设备组成，然后使用不同带宽的网络与其他集群互连。其次，本文提出了一种新的面向性能优化的分区机制，将 Ceph 的对象分配到相应的 OSD，从而可以获得更好的性能。

3.1 灾备问题

随着数据量的不断增长，技术框架的不断改进，用户在选择存储系统方面的标准也随之越来越高。传统意义上的数据备份，在面对如断电，网络大规模故障或者一些无法预计的自然灾害时，已经无法满足用户对于系统高可用性的要求。为了能更好地解决可能会发生的备份丢失的问题，现在大部分企业都采用了异地容灾，或者异地多活的方式来应对这一问题，提高在各种不可抗因素发生的情况下系统的可用性。

传统的 Ceph 将所有对象存储设备看作是一个整体的存储系统对外进行服务，每个 PG 会根据 CRUSH Map 中的 bucket 类型进行划分，通常分为主机，机架，以及机柜三个级别，然后将每个 PG 的多个副本分布在不同的故障域。这里的故障域对应的就是上文所说的传统数据备份时所采用的方法。但事实上，在实际生产中，只是凭借这一机制，无法完全解决灾备问题。因为上述方法只能保障单个集群内部故障引发的数据丢失，而当整个集群发生故障时，系统将不能保证可用性。所以现在大多数公司在使用分布式系统时都会采用异地多活的技术，相比于异地容灾的方法，异地多活要更加复杂，对数据的同步要求更高，因为不再是将数据以冷备份的方式，备份在地理隔绝的集群上。而是通过一系列分布式一致性算法，来保证当系统一部分集群宕机后，系统可以快速切换，并继续对外提供服务，比如 A 地发生了自然灾害，B 地还可以对外提供正常服务，而不需要多余的备份恢复工作。异地多活可以很好的解决容灾问题，另外异地多活还具备降低集群的部

署成本，实现流量均衡等等的优点。而采用异地多活所面临的问题主要是应该部署一个系统，还是部署多个相同的系统。如果部署一个系统，就需要一个有效的数据分配和资源调度机制以及一致性问题。而如果采用多个镜像系统的部署，就会导致存储空间的浪费，并且同样会涉及到的多个系统同步的问题。

就本文而言，针对原本的 Ceph 架构，将底层的存储集群通过跨机房的部署方式或者叫多机房部署来对原本的集群进行地理上的故障域划分，通过维护一个由多个不同地理位置的集群组成的系统，从而解决 Ceph 原本无法解决的异地容灾问题。

3.2 异构系统

现在的分布式系统相比于过去那些由相同节点组成的普通集群或超算集群而言，任意两个节点的故障率、带宽、网络延迟、数据处理速度、安全性和其他方面都可能会有很大的差异。这种异构性可能是由许多因素导致，但根本上是因为数据量爆炸性的增长驱动的。现代多数互联网应用都具有非常大的规模，因此即使是在单一的数据中心内也可能有设备之间存在差异的问题。

由于本文在 Ceph 中添加了地理位置这一级别的故障域，将原本 Ceph 构成的底层存储根据地理位置进行了划分，分散到了不同的地方。所以面临的问题是，因为集群位于不同的地理位置，就会导致在集群之间传输数据时的网络延迟不同，从而产生网络的异构问题。相比低延迟网络，在高延迟网络上的通信会导致传输速率变慢，而网络传输速率的差异最终会对整个系统性能造成影响，因为当任务间的数据存在依赖关系时，这就要求下游任务的执行必须要等到上游任务的结果完成传输后才能开始，这个时候网络因素就成为了瓶颈。

除去网络因素以外，OSD 的计算能力是另一个影响整个系统性能的因素。相比于同构集群而言，允许位于不同地理位置的集群之间计算性能存在差异，比如可能不是同一时间采购上线，或者采购的标准有所差别。因此，在执行相同的单位任务时，由于每个 OSD 的计算能力可能不同，会导致这些 OSD 执行相同任务时产生的时间开销不同。考虑到上层的计算框架使用的是 MapReduce，根据数据本地化的特性，Map 任务会优先对本地的数据进行处理[43]。所以如果计算性能强的 OSD 上有更多的数据，那么在同一个时间段内，具有更强计算能力的 OSD 可以完成更多的任务。这就是为什么想要把更多的数据放到性能比较好的设备上。

然而，当数据量远远超过所有那些所谓性能强的机器容量总和时，就必须要考虑集群的负载均衡，负载均衡的目的是为了避免某一台主机因为数据或计算任务过多，导致访问节点时发生阻塞。集群通过负载均衡可以提高可靠性和可用性，可以优化资源分配，平衡流量，减少网络拥塞，有利于提高系统的整体性能以及

减少故障率。如果把所有的数据全部都放在某些性能很好的设备上，那么可能会出现部分性能好的设备上任务队列过长造成等待，而其他设备任务队列为空所造成计算资源的浪费，也更容易导致节点资源不足造成宕机。所以，将所有的数据分配到部分高性能的设备有时并不是最好的选择，需要对各个因素综合考虑。

本文重点考虑网络延迟，节点计算性能以及存储容量的异构，暂时忽略其他可能存在的异构情况。为了充分利用负载均衡的优点，本文通过引入负载均衡的约束条件这一概念，保证异构集群间的相对负载均衡。

3.3 面向性能优化的分区机制

为了迎合大数据时代对于数据处理的性能需求，数据的放置策略面临诸多挑战：①如何合理放置数据，使得存在依赖关系的数据尽可能减少网络传输；②如何在异构环境下，通过改进数据放置策略，充分发挥不同存储设备的处理能力；③如何在兼顾系统负载均衡的同时提升系统总体的吞吐量。

针对以上的三个问题，本文提出基于 Ceph 架构的优化方法，并提出一种基于分区机制的数据分配优化策略，以达到提升系统整体性能的目的。

Ceph 中所有的数据最后都会以对象的方式存储到 RADOS 中。在默认情况下，Ceph 会通过 CRUSH Map 中的获取的用户自定义的放置规则以及集群拓扑，通过 CRUSH 算法，将数据较为均匀分配在所有的 OSD，因为 CRUSH 算法实质上是一个带参数的哈希算法。但是 CRUSH Map 获取的用户自定义放置规则通常只是根据磁盘的容量大小来决定每个 OSD 的权重，而没有考虑到每个 OSD 实际的计算能力的差异，以及集群内部网络延时的差异。因为忽略了上述的这些可能存在的异构因素，当运行 MapReduce 这样的应用程序去解决实际问题，如日志分析和数据预测时，由于只考虑容量这一因素，会使得容量大的机器上数据量多。但是，假设这样的机器计算性能很差，那么就会导致任务堆积或者造成大量的网络传输流量。所以仅仅以这样的方式对数据进行放置，应用程序的执行时间往往会超出预期的时间。而本文主要针对改进的 Ceph 架构进行优化。所以为了简化研究的问题，需要添加一个假设的前提条件。本文假定不同集群内部的计算性能，存储容量，以及网络延迟都是同构的，即集群内所有的 OSD 都采用同样的配置，这在实际采购，部署，上线过程中也是比较合理的，所以由此可以忽略每个 OSD 之间计算性能的细微差异。而在改进的架构下，内部的网络延时，相比与外部的网络延时几乎可以忽略不计，因此，也可以假定集群内部各个节点之间的网络延时也是完全相同的。

在这些假设条件下，本文提出面向性能优化的分区机制来分配数据，在满足负载均衡的约束条件的同时，使得整个应用程序执行时间最小化。

面向性能优化的分区机制的主要思想是充分利用分布式存储系统的异构性，特别是设备的计算能力与网络延迟，在负载均衡限制条件和性能方面做一个折中。在这种情况下，本文在 Ceph 的原有架构中增加了一个数据分配概率表，这个概率表的思想是借鉴了轮盘赌选择算法。

概率表用于将数据分配给放置组，这些放置组和相应固定的集群绑定。为了实现这一点，需要在概率表，放置组和集群之间建立相互的映射关系。概率表中的分布比率是根据在负载均衡的前提下，利用优化算法求出的最优策略得到的。每一个分区都代表着数据被分配到某个集群上的概率。

具体来说，首先对数据的做一个采样，然后按照优化算法，将采样数据分配到不同集群的 OSD 上，得到最优的数据分配方式，相应可以得到不同集群所包含对象的数量，求出每个集群所分配的对象占总样本的比例，并作归一化的处理。最后将每个集群所占的比例大小整合到一条直线上。即为所求的概率表。

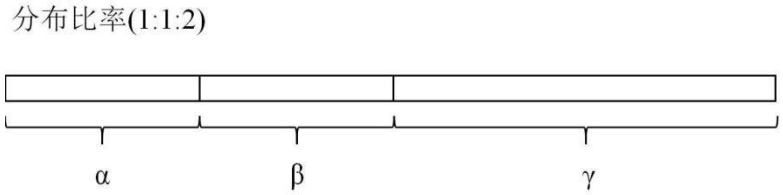


图 3.1 面向性能优化的分区机制示例

Figure 3.1 An example of performance-oriented partitioning

接下来，通过一个例子，具体阐述概率表的工作原理。假设已经通过优化算法得到了一个概率表如图 3.1 所示，各个分区比例之和等于 1。如果现在有四个对象需要分配到三个集群：α、β和γ，并且这三个集群的负载情况相同。不同于传统的 Ceph 那样，对象被平均分配到 OSD。利用概率表将对象分配给集群，则对于任意一个对象，将会被按照 0.25，0.25，0.5 的概率分配给上述三个集群，然后针对每个集群，CRUSH 算法用于将对象映射到 OSDs。

3.4 数据分配机制

这一部分具体描述了在本文提出的架构中，是如何对数据进行处理的，如图 3.2 所示。首先，需要将被处理的数据切分成许多的对象，对象的概念和定义与传统 Ceph 中的概念相同，不作修改，但由于在本文中，上层应用框架使用的是 MapReduce，所以每个对象大小修改为 MapReduce 所定义的数据块大小，即每个对象对应一个数据块，大小为 64M。

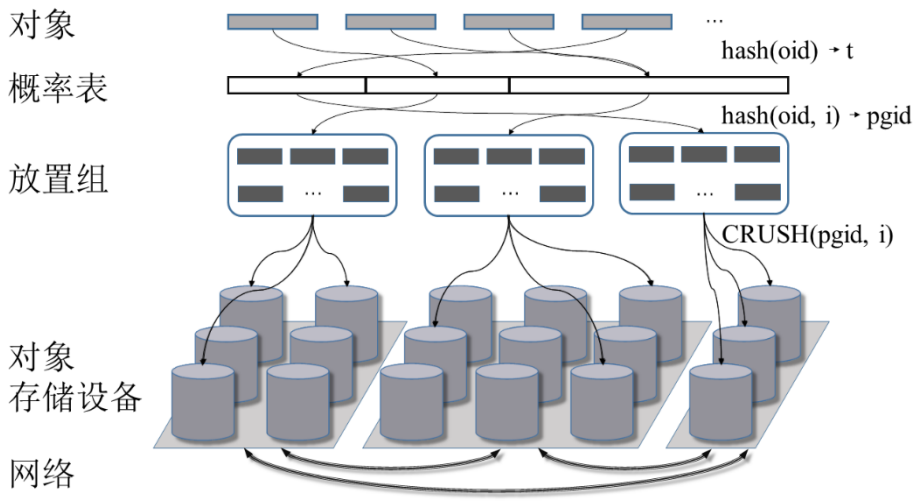


图 3.2 改进 Ceph 的数据分配机制

Figure 3.2 The mechanism of data placement in proposed Ceph

然后将对象作为参数调用哈希函数。区别于传统 Ceph 中的一次哈希，这里通过调用两次哈希函数，分别获取对象对应的集群 ID，以及每个集群内部 CRUSH 算法的输入所需要的 PGID。

第一次调用的哈希函数是为了获取对象所对应的集群 ID。哈希函数的输入是对象 ID，然后根据哈希函数执行的结果，做归一化处理，接着将结果与概率表作比较。由于上面讲过，概率表中每一个分区对应的都是一个特定集群。所以通过与概率表进行比较，根据该对象的分区 ID 可以获得对应的集群 ID。

在得到集群 ID 之后，对该对象进行第二次哈希，第二次哈希的输入是对象 ID 以及第一次哈希函数得到的集群 ID，哈希的输出为 PGID。区别于传统 Ceph 中的哈希只有对象 ID 一个输入，在进行第二次哈希时，添加一个集群 ID 作为输入，因为每个集群都会对应到一些固定放置组（每个放置组预先已经和特定集群绑定），所以通过集群 ID 来保证所得到的 PGID 是与集群 ID 相关的。

最后调用 CRUSH 算法，同样，在算法中增加一个集群 ID 作为输入参数，来确保 PGID 可以分发到与之绑定的特定的集群中，从而获得与对象相对应的 OSD 位置。

上述是如何将对象分配到对应节点的具体过程。因为本文是通过替换 HDFS 而作为 MapReduce 计算框架的底层存储，所以任务分配过程与传统 MapReduce 的过程相似。当数据分配好以后，MapReduce 会按照数据存放的位置，将待运行的程序拷贝到对应数据所在的节点，然后启动相应的进程。

3.5 数据备份流程

数据备份是为了保证数据在意外发生的情况下，不会造成数据永久性丢失，对于整个系统优劣的评估占有很大的比重。Ceph 本身支持两种方法对数据进行备份。一个是使用纠删码，另一个是使用传统多副本的形式。纠删码的好处在于，副本之间是通过计算的方式对数据进行恢复的，而多副本则是通过对数据冗余保证数据的可靠性。纠删码相比于多副本机制的优点在于可以节省存储空间，但是不足之处是，需要消耗多余的计算资源用于恢复数据时的计算工作。由于本文重点关注系统的性能提升，所以在这里以多副本为例，做一个简单介绍。

通过上述方法，可以得到对象的 primary OSD，primary OSD 是主要对外服务的节点，但是仅有 primary OSD 还不够，为了能够拥有更高的可用性，就会涉及数据的备份问题，因此需要获得副本的对应存储位置。与传统 Ceph 相同，依然可以在每个集群内部支持不同故障域级别的隔离，如主机，机架，机柜等。除此之外，还可以提供更为安全的跨地域级别的故障域隔离。用户可以根据自己的实际需求来决定选择哪一种或者多种故障域级别隔离方式。

针对跨地域级别的故障域隔离，仅需要依次将概率表中对象所在的集群分区删除，对剩余的分区作归一化处理，这样就可以保证对象的副本位于不同地理位置的集群。接下来，通过一个例子来进行说明。假设有一个对象，副本的数量为 3，底层的存储按照地理位置被划分为 5 个区域，也就是整个系统由五个集群组成，分别为 A，B，C，D，E，由优化算法并经过归一化处理得到分区概率表，假设分别为 0.1，0.1，0.2，0.3，0.3。再假设计算 primary OSD 时通过哈希得到的数值为 0.6，则对应集群 D，换句话说，就是该对象对外提供服务的集群是 D。然后需要对该对象的所有副本进行放置，首先将 primary OSD 所在的集群从概率表中排除，就是上述的 D，然后计算第二个副本应当存放的位置。对除 D 以外的剩余集群做归一化处理，可以得到新的分区概率表。在这个例子中，得到的新的分区概率表对应于集群 A，B，C，E 分别为 0.14，0.14，0.29，0.43，接着同样进行哈希求出第二个备份所在的集群。第三个副本同理。

虽然这种方法与传统方法相比，需要多次调用哈希，但是提供了更完善的故障域隔离级别，提高了系统的可用性。

3.6 本章小结

本章首先分析了传统 Ceph 在容灾备份方面的不足之处，进而引出了一种新的架构，将底层存储根据地理位置进行拆分。由于集群部署的地理位置不同，会导致集群间网络延迟存在异构，又考虑到计算性能同样可能存在异构问题，综合两者，提出了面向性能优化的分区机制。介绍了如何利用概率表对数据进行分配，以及简单介绍了改进框架之后，如何处理数据备份的问题。

4 优化数据分配问题的定义与算法

本章，主要对所研究的问题建立模型并对问题进行了定义，另外，提出了两种用以解决问题的优化算法。

4.1 模型和问题定义

在这一节中，根据改进的 Ceph 架构并结合 MapReduce 计算框架，对所研究的问题进行了定义。这个问题主要是在最小化执行时间和负载均衡之间进行权衡，以便在实际生产中更佳有效。本节主要分成四个部分，分别从不同角度对问题模型进行定义，并针对给出的模型，对需要求解的问题进行定义。

4.1.1 系统模型

根据前一章提出的架构，这部分主要对集群底层的存储模型进行定义。一个存储系统通常由许多集群组成，集群的集合用CL表示。每个集群内部都由许多对象存储设备组成并通过内部高速网络互联，对象存储设备的集合这里用O表示。对于每个集群内部来说，存储设备的容量可能不同，设备容量用c_n表示，已用空间用u_n表示，u_n/c_n反应的就是这个集群n的负载情况，定义为l_n。影响集群整体负载的另一个因素是集群内部的 OSD 的数量，具体的计算集群负载情况的方法会在后文给出。除此之外，在本文的系统模型中，集群之间是通过异构网络进行相互连接，所以在传输数据时的开销会随着网络情况的变化而变化，这里定义$cet_{n,m}$，用来表示从节点o_n传输一单位的数据块到节点o_m所用的时间。并且为了更加清楚显示 OSD 和集群两者之间的对应关系，定义$C_{x,y}$来表示 OSD 和集群的映射关系。

4.1.2 应用模型

这一部分主要阐述了具体应用模型的定义，因为本文在上层计算框架方面选用的是 MapReduce，所以专门针对 MapReduce 的流程进行建模，由第二章背景技术分析可知，MapReduce 工作流程包括三个阶段：Map 阶段，Shuffle 阶段和 Reduce 阶段[2]。另外，MapReduce 为了优化系统性能，提高资源的利用率，一旦某个 Map 任务完成并且集群中的节点存在闲置资源的时候，Shuffle 阶段就可以开始，由于在系统模型中已经引出了网络传输所花费的开销$cet_{n,m}$，而且 Shuffle 阶段最主要的时间开销就是网络传输，所以本文没有另外对 Shuffle 阶段建模，直接采用系统模型中定义的网络传输的开销。然后通过在空闲节点上启动 Reduce 任务，调用 Copy 函数，可以获取各个节点上 Map 任务的中间输出。因此，如果系统存在剩余空闲资源，那么 Map 和 Reduce 阶段实际可以在同一时刻执行。

考虑到在大多数情况下，根据数据本地化的原则，MapReduce 的数据放置问题主要是指在 Map 阶段的初始数据放置，为了简化问题，暂时忽略了 Reduce 阶段的数据放置问题。为了描述任务（数据块）与节点的映射关系，本文定义了两种任务放置函数A: $M \to O$和A: $R \to O$，A是一个 0-1 变量，用来表示任务（数据块）是否存在于节点上。例如，对于任意的一个 Map 任务i和 OSD n， A_{m_i,o_n}表示 Map 任务i是否分配给 OSD n，如果任务i分配给了 OSD n，则$A_{m_i,o_n} = 1$，反之，如果任务i没有分配给 OSD n，则$A_{m_i,o_n} = 0$。同理，A_{r_j,o_n}则代表是否将 Reduce 任务j分配给 OSD n。

此外，因为本文主要针对的是异构环境下的任务分配问题，所以同一任务在 OSD 上的执行时间由于 OSD 的不同，计算能力是不同的。因此，定义mst_{m_i}和rst_{r_j}用来表示 Map 任务i和 Reduce 任务j分别的开始时间，同时定义met_n和ret_n分别表示分配给 OSD n的 Map 和 Reduce 任务相应的执行时间。这里下标使用n而没有使用m_i或者r_j，是因为在 MapReduce 计算框架中，所有 Map 任务的内容都是相似的，所以任务的执行时间仅与节点的性能有关。有了上述定义，就可以通过$mst_{m_i} + met_n$来得到 Map 任务i的完成时间，同理也可以得到 Reduce 任务j的相应完成时间。

4.1.3 负载均衡约束的阈值

负载均衡作为本文重点考虑的限制条件在这部分给出定义。由上文对于 MapReduce 特性的说明，这里可以使用数据块来代表每个对象。因为本质上在 Map 任务开始前，会对数据进行切片，而 Ceph 在存储数据时同样会将数据切块变成对象，所以在这个特殊的情况下，本文规定每个对象大小等同于数据块的大小。a_n表示分配给 OSD n的数据块数量。aa_n则代表在 OSD n上放置a_n个数据块以后，OSD n现有的数据块的数目。另外这里给出负载均衡的定义：给定一个负载均衡约束阈值的T，T是一个可以调整的阈值，并且由系统的所有者根据网络条件，硬件状况等因素决定。如果在所有集群中，拥有最大负载的集群的l_n与拥有最小负载的集群的l_n之间的偏差小于这个阈值，就可以认为这个存储系统是相对负载均衡的。这里之所以使用偏差作为计算，而不是使用方差进行计算，是因为目前大多数学术研究中都使用偏差作为衡量负载是否均衡的方法。另外这里的负载均衡是一个相对的概念，也就是说，并不是完全的负载均衡，只是建立在允许偏差范围之内的负载均衡。

4.1.4 问题的定义

有了上述关于模型的定义，接下来给出本文所要研究问题的定义。

问题输入是，给定一个对象的集合（如上所述，一个对象也等价于一个数据块），然后给定一组集群，不同集群上每个 OSD 执行相同任务的时间，集群之间的网络带宽延时，还有系统所有者定义的负载均衡的约束阈值。

并且所定义的问题是针对 MapReduce 这样具体的计算框架，待解决的问题是如何在上述给定的输入条件下，确定数据放置的策略（由于考虑到 Map 任务的本地化特性，所以数据放置策略与 Map 任务分配等价）使得系统在满足负载均衡的约束前提下，最小化 MapReduce 程序的执行时间。

因此，问题的输出就应该包括对象在集群间的分布状况以及任务最终的执行时间。

4.2 混合整数线性规划算法

任务调度是分布式异构系统中非常重要的问题，针对这个问题，近几年已经有不少的研究成果并提出了各种各样的算法[44-51]。在这一节中，本文利用 MILP 算法来解决上文提到的异构集群环境下的数据放置问题并给出最优解。

MILP 算法是指线性规划模型中的约束变量既有整数，也有连续值。而在研究中遇到的很多实际问题都可以使用它来进行求解，尤其是针对某些特殊的问题，比如，任务调度问题和网络流问题等。很多的优化问题都可以将一个大问题转化为多个小的子问题，从而通过 MILP 算法进行求解。

这里首先使用在上一节中定义的符号，构成待求解问题的约束条件，具体如表 4.1 所示。另外根据上文所述，可以假设 Hadoop 的 MapReduce 的数据块的大小等于 Ceph 的对象的大小。

表 4.1 MILP 算法的符号定义

Table 4.1 The notations of the MILP algorithm

标记	描述
O	输入对象的集合
M	Map 任务的集合
R	Reduce 任务的集合
CL	位于不同地理位置集群的集合
T	负载均衡约束的阈值
$C_{n,x}$	OSD n和集群x的关系
bn	一个趋近于无穷大的数
mst_i	Map 任务i的开始时间
rst_i	Reduce 任务j的开始时间
$cst_{i,j}$	将 Map 任务i输出的中间结果拷贝到 Reduce 任务j的开始时间
met_n	在 OSD n上单个 Map 任务的执行时间
ret_n	在 OSD n上单个 Reduce 任务的执行时间

标记	描述
$cet_{n,m}$	从 OSD n将 Map 输出的中间数据拷贝到 OSD m的时间
c_n	OSD n的总容量
u_n	OSD n的已用空间
a_n	分配给 OSD n的数据块数量
aa_n	分配后 OSD n的已用空间
l_x	分配后的集群x的负载情况
A_{m_i,o_n}	如果将 Map 任务i分配到 OSD n上，则为 1，否则为 0
A_{r_j,o_n}	如果将 Reduce 任务j分配到 OSD n上，则为 1，否则为 0

本文提出的 MILP 算法相比于其他现有的研究，对整个 MapReduce 的过程都添加了约束，包括 Map 任务开始时间，执行时间，以及中间结果的拷贝时间，还有 Reduce 任务何时开始，以及相应的执行时间。除此之外，在将对象（数据块）分配到对象存储设备的同时，将整个系统的负载情况作为约束条件，提供一个阈值作为负载均衡的标准。因此，这里将 MILP 算法涉及的约束分成两个部分。

4.2.1 任务约束

这一节主要对分布式计算框架 MapReduce 的流程做梳理，列出该框架相应于 MILP 算法的限制条件。

首先对上文定义的两个 0-1 变量做出限制。每个 Map 任务必须且仅能够被分配到一个 OSD 上，即对于任意一个 Map 任务i，A_{m_i,o_n}进行求和，结果为 1，

$$\sum_{n\in O} A_{m_i,o_n} = 1, \forall i \in M \tag{4.1}$$

同理，每个 Reduce 任务必须且仅能够被分配到一个 OSD 上，

$$\sum_{n\in O} A_{r_j,o_n} = 1, \forall j \in R \tag{4.2}$$

在 Shuffle 阶段时，每个 Map 任务的中间结果被复制到 Reduce 任务节点的开始时间必须在相应的 Map 任务完成时间之后，$\forall i \in M$，$\forall j \in R$，$\forall n \in O$，

$$cst_{m_i,r_j} \geq mst_{m_i} + met_{o_n} + \left(A_{m_i,o_n} - 1\right) \cdot bn \tag{4.3}$$

之所以在后面添加一个$(A_{m_i,o_n}-1)\cdot bn$的目的是通过一个 0-1 变量与大数的乘积，可以用一个不等式来约束m_i在或者不在节点n两种情况。m_i如果在节点n上，那么后半部分为 0，不等式自然成立，如果不在n上，那么后半部分的值为负无穷，该不等式依然成立。

与上述方法相似，接着给出另一个约束，Reduce 任务只能在 Shuffle 阶段所有中间结果的复制完成之后，才能开始，$\forall i \in M$，$\forall j \in R$，$\forall m \in O$，

$$rst_{r_j} \geq cst_{m_i,r_j} + cet_{o_n,o_m} + (A_{m_i,o_n} + A_{r_j,o_m} - 2)\cdot bn \tag{4.4}$$

下面的两个约束需要有一个先行的假定条件，为了简化问题，本文假设每个计算节点上仅有一个槽（Slot），也就是说同一时间单个节点只能执行一个任务，暂时忽略节点内部的并行。

在同一个 OSD 上先后执行的两个 Map 任务必须不能有重叠，$\forall i \neq j \in M$，$n \in O$，

$$\left(A_{m_i,o_n} + A_{m_j,o_n} - 2\right)\cdot bn + (mst_{m_j} - mst_{m_i})(mst_{m_i} + met_{o_n} - mst_{m_j}) \leq 0 \tag{4.5}$$

在同一个 OSD 上先后执行的两个 Reduce 任务必须不能有重叠，$\forall i \neq j \in R$，$n \in O$，

$$\left(A_{r_i,o_n} + A_{r_j,o_n} - 2\right)\cdot bn + (rst_{r_j} - rst_{r_i})(rst_{r_i} + ret_{o_n} - rst_{r_j}) \leq 0 \tag{4.6}$$

考虑到这两个公式都是非线性的形式，无法适用于混合整数线性规划模型，为了使这些公式变成线性形式，可以通过夹逼的方式，来对约束进行转换，所以在这里定义了四个 0-1 变量：x，y，p，q用来辅助，从而实现非线性模型向线性模型的转化。以 Map 任务为例，因为两个 Map 任务不能同时在同一个 OSD 中开始执行，所以有两个情况满足这个不等式。一个是假设 Map 任务j的开始时间大于 Map 任务i的开始时间，那么 Map 任务i的开始时间加上执行时间必须小于 Map 任务j的开始时间，即$mst_{m_i} - mst_{m_j}$和$mst_{m_i} + met_{o_n} - mst_{m_j}$具有相同的符号。另一个是在上述情况下，Map 任务i的开始时间加上执行时间如果等于 Map 任务j的开始时间，那么同样成立，即$mst_{m_i} + met_{o_n} - mst_{m_j}$等于零。进一步可以得到下面三个式子，而如果两个 Map 任务之间没有重叠，换句话说，下面的三个式子将同时成立，且x和y的总和将等于 1，即$\forall i \neq j \in M$，$n \in O$，

$$x_{m_i,m_j} \geq \frac{mst_{m_i} + met_{o_n} - mst_{m_j}}{bn} + \left(A_{m_i,o_n} + A_{m_j,o_n} - 2\right) \cdot bn \tag{4.7}$$

$$y_{m_i,m_j} \geq \frac{mst_{m_j} + met_{o_n} - mst_{m_i}}{bn} + \left(A_{m_i,o_n} + A_{m_j,o_n} - 2\right) \cdot bn \tag{4.8}$$

并且如果没有重叠，那么$\forall i \neq j \in M$，

$$x_{m_i,m_j} + y_{m_i,m_j} = 1 \tag{4.9}$$

Reduce 任务的转换采用的方法和 Map 任务的转换类似，$\forall i \neq j \in R$，$n \in O$，

$$p_{r_i,r_j} \geq \frac{rst_{r_i} + ret_{o_n} - rst_{r_j}}{bn} + \left(A_{r_i,o_n} + A_{r_j,o_n} - 2\right) \cdot bn \tag{4.10}$$

$$q_{r_i,r_j} \geq \frac{rst_{r_j} + ret_{o_n} - rst_{r_i}}{bn} + \left(A_{r_i,o_n} + A_{r_j,o_n} - 2\right) \cdot bn \tag{4.11}$$

如果没有重叠，那么$\forall i \neq j \in R$，

$$p_{r_i,r_j} + q_{r_i,r_j} = 1 \tag{4.12}$$

4.2.2 负载均衡约束

上一个小节已经给出了任务部分的约束公式。由于是特定的计算框架，所以 Map 任务的数量就可以代表对象的数量。这里用a_n表示每个 OSD 上分配 Map 任务的数量，通过对一个 OSD 上所有的 Map 任务进行求和得到。

$$a_n = \sum_{i \in M} A_{m_i,o_n}, \forall n \in O \tag{4.13}$$

接着用下面两个公式计算某个集群的负载情况，可以通过计算将数据分配到这个集群之后，整个集群的容量和已经使用的空间的比率，得到这个集群的负载情况。特别地，因为在集群内部，数据的放置是通过 CRUSH 算法完成的并且内部是同构的，所以这里假定所有的 OSD 上分配到的数据量是相同的，仅需要通过计算集群中所有节点在分配后的数据量之和与所有节点容量的总和，就可以通过相除的方式得到集群的负载情况。

$$aa_n = u_n + a_n, \forall n \in O \tag{4.14}$$

$$l_x = \frac{\sum_{n \in O} c_{n,x} \cdot aa_n}{\sum_{n \in O} c_{n,x} \cdot c_n}, \forall x \in CL \tag{4.15}$$

这里用T来表示负载均衡约束的阈值，即拥有最大负载和最小负载集群的偏差，并且每个集群的负载情况必须满足这个阈值[52]。

$$l_{max} \geq l_x, \forall x \in CL \tag{4.16}$$

$$l_{min} \leq l_x, \forall x \in CL \tag{4.17}$$

$$l_{max} - l_{min} \leq T \tag{4.18}$$

4.2.3 优化目标

有了上述的两类约束条件，现在需要给出整个模型的优化目标。而由于在MapReduce 框架中，Reduce 任务需要等到 Map 任务全部完成之后才能开始执行，所以优化的目标从最小化整个任务的执行时间转化成最小化最慢的一个 Reduce 任务的完成时间，即最小化

$$\max\{rst_{r_i} + ret_{o_n}\}, \forall i \in R, n \in O \tag{4.19}$$

在得到优化目标之后，将上述约束不等式作为输入通过 lingo 或者 gurobi 等优化软件进行求解，即可得到对应的任务分配，以及最后的任务执行时间。

4.3 基于遗传算法改进的数据放置算法

当对象存储设备，集群数量或者任务数量比较多时，利用 MILP 算法的时间复杂度太高，无法在合理的时间内得到最优解，因而需要寻找一种快速的算法进行求解。而且真正在大规模集群中时，数据的分配受限于很多因素，无法做到完全精准。鉴于这个情况，可以使用一种近似算法来求得问题的近似最优解，代替最优解，从而在时间上可以更符合实际。

遗传算法是一种启发式的近似算法，在最近几年的相关研究中，已经有不少人通过对遗传算法进行改进，从而求解任务调度的问题[53-54]。它的大致思想是借鉴了生物在进化过程中可能出现的染色体变异以及交叉等概念，通过将优化问题的候选解群（由染色体组成的集合）进行演化迭代，从而逐渐进化为更好的解。每个候选解中有一组属性（基因）可以发生突变或者交换以重组当前的解，通常

的做法是对每一个候选解以二进制 0 和 1 的方式进行编码得到相应的字串，当然也可以使用其他的编码方式。

遗传算法在开始的时候，先由许多随机生成的个体组成一个种群，成为第一代，同时，评估每个个体自身的适应度，适应度是由优化问题的目标所决定的。接着根据给定的选择算法从当前种群中选择出那些更好，更适合的个体，对它们的基因组进行修改（重组或者随机突变）从而形成第二代种群，然后进行下一次迭代。通常，当迭代次数到达事先规定的最大迭代次数时或者种群的适应度达到预先规定的水平，该算法就停止了。

基于遗传算法理论，本文提出了一种高效的改进算法（DPGA 算法）来解决之前定义的问题模型。本文所提出的算法可以充分模拟任务的分配情况，包括 Map 和 Reduce 两种任务，另外还特别考虑到了集群间的网络延迟以及负载均衡的约束条件。

首先，给出算法伪码中将会使用到的一些符号。定义G作为更新种群的迭代次数，C作为整个种群，是若干染色体的集合，用$C.size$表示染色体的数量，p_c和p_m用来表示是否调用交叉算子或者变异算子的预定概率，然后定义N和O用来表示总的任务量（染色体中的基因数目）和 OSD 的数目，CP表示交叉点。

本文所提出的算法的基本思想是首先利用随机的方式初始化由不同染色体组成的种群。每个染色体代表一种任务分配的解，因为本文使用的是 MapReduce 计算框架，所以这部分包含 Map 任务的分配和 Reduce 任务的分配。由于本文假设 Map 任务是在存放数据的 OSD 上执行的，所以可以使用 Map 任务的分配情况来表示数据放置，染色体的长度会随着数据量的增加而变长。区别于传统那些遗传算法，将染色体编码成二进制 0 与 1 的字串，本文直接使用存储设备的编号对染色体进行编码，即假设有一个染色体的 Map 任务部分的分配情况为 1，2，5，则代表数据块分别被分配到了编号为 1，2 和 5 的存储设备上。此外，当染色体初始化时，会首先判断数据放置方案是否符合负载均衡的约束条件。假如不符合，将会重新初始化，直到符合约束条件。在突变和交叉阶段，同样会有这样的检测，从而在整个算法的计算过程中，始终可以保证结果满足负载均衡条件的限制。

因为遗传算法本身属于近似算法，所以它只能得到近似的最优解，但是和传统算法相比，该算法可以快速地得到更好的解。具体过程如算法 4.1 所示：

算法 4.1 基于遗传算法改进的数据放置算法

```
输入：G, C, CP, pm, pc
输出：最优的任务分配策略
1: for chrome in C do
2:         Initialization()
3: end for
4: g<-0, c<-0, sum<-0, vec<-[]
5: while g<G do
6:         while c<C.size do
7:                 fit<-fitness()
8:                 vec.push(fit)
9:                 sum<-sum+fit, c<-c+1
10:                if random<pc:
11:                        Crossover_operator(N, C, CP, vec, sum)
12:                if random<pm:
13:                        Mutation_operator(N, C, vec, sum)
14:                g<-g+1
15:        end while
16: end while
```

调用算法时会先通过一个循环对所有的染色体进行初始化，它将通过随机函数生成一系列由 Map 和 Reduce 任务分配的组合构成的染色体。然后根据 Map 部分的分配情况和第一小节定义的负载均衡计算方式，计算每个集群的负载情况，以及得出整个系统是否满足负载均衡的约束。如上所述，这里将 Map 任务分配到不同的节点等同于将数据放置到不同的节点。Reduce 采用随机分配的方式，并且分配的结果不计入负载。

接下来将迭代次数置 0 并开始进入种群更新迭代过程。

每次更新开始时，先获取种群中每个染色体的适应度，并通过遍历的方式得到所有染色体适应度的和，以及由每个染色体适应度组成的数组。数组中任意一个值除以适应度之和，就是每个染色体发生变异或者交叉时，可能被选择的概率。

然后根据随机生成的数与p_c比较，决定是否调用交叉算子，若调用，将染色体集合，以及适应度数组和适应度之和作为参数传入，根据之前的评估适应度所得到的概率获取两条染色体，交换两条染色体以继承良好的特性，产生新的染色体。

同理，根据随机生成的数与p_m比较，决定是否调用变异算子，若调用，传入同样的参数，并以同样的方式获取一条染色体进行变异操作。

最后迭代次数加一，进入下一次循环。

算法 4.2 遗传算子

```
输入：N, O, C, CP, vec, sum
输出：新的染色体
1: procedure Crossover_operator(N, C, CP, vec, sum)
2:         Ci<-max{vec[i]/sum}
3:         vec.remove(i)
4:         Cj<-max{vec[j]/sum}
5:         for n<-1 to CP do
6:             repeat
7:                 k<-random(N)
8:                 swap(Ci[1..k], Cj[1..k])
9:             until Ci, Cj 满足负载均衡约束的条件
10:        end for
11:        return Ci, Cj
12: end procedure
13:
14: procedure Mutation_operator(N, C, O, vec, sum)
15:        Cm<-max{vec[m]/sum}
16:        repeat
17:            k<-random(N)
18:            v<-random(O)
19:            Cm[k]<-v
20:        until Cm 满足负载均衡约束的条件
21:        return Cm
22: end procedure
```

关于交叉算子，通常有单点交叉，两点交叉，多点交叉，以及均匀交叉和算术交叉[55]。这里使用两个交叉点来交换选定的染色体。算法 4.2 说明了交叉操作是如何进行的，首先根据染色体适应度大小按照一定的方法选择出要执行交叉算子的染色体，接着将其排除后，以同样的方法选择出另一条染色体，得到的两条染

色体就是将要执行交叉算子的染色体。其次交叉算子会随机从染色体中选出一位，作为交叉点，然后将两条染色体从开始部分一直到交叉点进行交换，重复 n 次这样的过程。假设这里选择两点交叉，那么 n 等于 2。最后对两条染色体进行验证，判断其是否达到了负载均衡约束的要求。如果没有达到，需要重新对两条染色体执行上述过程，反之，返回两条经过交叉后产生的新的染色体。

为了避免局部最优解，传统的遗传算法会根据变异概率p_m，决定是否调用变异算子，如果调用，该算子会把经过选择得到的染色体中的某一位从 0 变为 1 或者从 1 变为 0。通过这种方式更新种群，变异算子可以扩大问题的解搜索空间，从而得到更好的结果。改进后的变异过程同交叉算子一样，首先通过计算适应度，根据各自所占概率比的大小，选取一条染色体执行变异算子。然后从染色体中随机选择出一位作为变异点，并从所有存储设备编号中随机选出一个，作为变异后的值，且每次执行变异算子，都需要判断由变异生成的染色体是否满足负载均衡的约束，直到满足条件，返回新生成的解。

在算法的每一次迭代中，遗传算法通过利用适应度函数，对当前种群中包含的所有解决方案的质量进行评价，并根据预定概率对被选择的染色体执行交叉运算或者变异运算。适应度函数用来评估一个种群中单个解决方案的好坏，在遗传算法中起着非常重要的作用。

针对种群每次具体的更新，使用算法 4.3 中的给出的算法获取所有染色体各自的适应度。这里有一点需要注意，每个染色体都代表数据放置的一种方案。所以根据这些不同数据放置方案，可以将这些 MapReduce 的数据流图转换为不同的有向无环图（DAG），利用最晚完成时间来表示适应度。具体转化过程如下：

首先，规定 DAG 中所有的点与任务具体的执行时间无关，而由边来表示任务的执行时间。并在 DAG 中引入一个虚拟辅助节点，用来表示在某个节点上分配的 Reduce 任务的完成。

因为可能存在同一个机器上被分配多个 Map 任务或者 Reduce 任务，所以会出现 Map 节点相互连接或者 Reduce 节点相互连接的情况。而对于两个相互连接的 Map 节点而言，其相连接的边赋值为有向边起点的 Map 任务的执行时间，同理，两个 Reduce 节点相互连接的边赋值为有向边起点的 Reduce 任务的执行时间。另外，还存在两种情况，一个是 Map 与 Reduce 节点相连的边，这里赋值为 Map 任务的执行时间，加上执行 Map 任务的节点向 Reduce 任务的节点拷贝中间临时结果时所用去的时间。另一种情况是 Reduce 节点与虚拟节点连接的边，这里赋值为 Reduce 任务的执行时间。

算法 4.3 适应度函数

```
输入：DAG=<V, E>
输出：最长路径长度
1: for v, e in DAG:
2:        Initialization()
3: end for
4: topological_sort(DAG)
5: for v in S do
6:        r[v]<-0
7: end for
8: for v in V\S do
9:        r[v]<-max{r[u]+e(u, v)}
10: end for
11: for v<-1 to V.size do
12:        makespan<-max{r[v]}
13: end for
```

算法 4.3 给出了详尽的过程，为了理解，首先给出算法中相关变量的定义，$r[v]$表示以节点v结尾的最长路径，$V.size$表示所有的节点数量之和。特别地，定义S来表示 DAG 中入度为 0 的点组成的集合。

算法首先通过对 DAG 中所有的点和边进行初始化，得到一个用边来表示权重的 DAG，然后对其执行拓扑排序。拓扑排序用来形象地表示任务间执行的先后顺序，一般用节点表示任务，用节点之间的有向边来表示任务之间的关系。后续的任务必须在其所属的有向边起点的任务执行完成后，才可以执行[56]。之所以这里需要进行拓扑排序是因为在同一节点的多个 Map 任务和多个 Reduce 任务，以及 Map 和 Reduce 任务之间存在严格的顺序关系，通过将 DAG 进行拓扑排序，对于给定的边$e(u, v)$，可以确定任务u发生在任务v之前。

因为集合S中包含的节点的入度都为 0，代表这些点要么是孤立点，要么是起始节点，由于是 DAG，所以只可能是起始节点。然后将这些起点的最长路径$r[v]$赋值为 0，而对于除去S以外的剩余节点按照拓扑排序的先后结果依次求出最长路径。通过最长路径算法，求出各点的最大路径长度，并求出所有点中的最大值。最大值表示的是，在某种任务分配下的最终执行时间，也就是所需要的染色体适应度。

算法 4.3 给出的算法在求解最长路径时利用了动态规划的思想，将问题分解成规模更小的子问题进行求解。在获得全部节点的最大路径之后，调用一个循环求出最大值。

一旦求出所有染色体适应度的总和，其中某一条染色体的适应度所占据的比重，就反映了当交叉算子或变异算子被调用时，这个染色体获选的可能性，适应度越大的染色体越容易被选中。

4.4 本章小结

在本章的第一节，首先给出了待研究问题的定义和理论模型，以及可能会涉及的相关符号的定义。然后根据问题的定义，在第二节利用混合整数线性规划的方法对问题进行建模并求解，接着对该方法进行分析，发现数据量较大时，算法时间复杂度较高。因此，在本章第三节提出了 DPGA 算法。

5 实验方案结果及分析

为了证明算法的有效性，本文通过以下实验，从算法的执行效率和性能两个方面来对其分析评估。

5.1 实验环境及实验方案

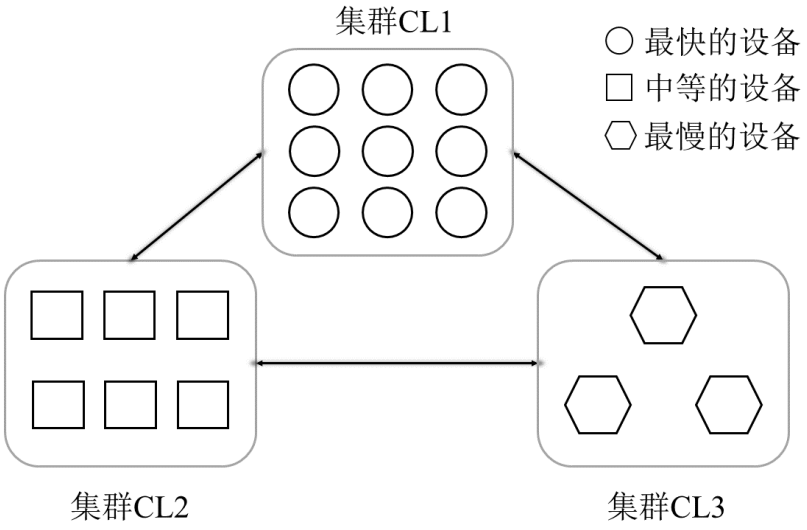


图 5.1 实验集群配置

Figure 5.1 The experimental cluster settings

为了符合本文的架构设计，在实验过程中，通过对 CPU 进行调频来表示计算性能的异构，整个实验运行在由三个异构集群组成的系统上。这些集群之间经由不同延时的网络互相连接，每一个集群都拥有不同数目的对象存储设备。如图 5.1 所示，集群 CL1 中的设备性能最好，运算能力最强，反之，集群 CL3 中的设备性能最差。

表 5.1 是相关环境的配置。通过利用 Ceph 官网提供的一个插件 cephfs-hadoop[57] 来完成对 Hadoop 中的分布式文件系统 HDFS 的替换，然后进行一个单机的测试，对应于不同的基准测试程序，分别得到测试结果，并从中提取出 Map 和 Reduce 阶段各自的运行时间。在进行实验时，MapReduce 使用默认参数配置（比如，块大小是 64MB）[57]。网络的带宽和延时是通过使用 iperf3[58]测试在不同地理位置的两个设备之间，传输数据时的传输速率得到的，并且同一个集群中的设备之间的相互传输被视为本地传输，忽略内部的网络延时。另外本文在实验过程中暂时忽略副本的问题，所有算法都只是适用于单副本的情况。算法的参数如表 5.2 所示。

其中交叉概率和变异概率为经验值。为了衡量在不同的数据放置策略下，系统性能的差异，实验时，通过设计了一个模拟器，来获取 MapReduce 最后的完成时间。上述所有的实验都运行在英特尔的 Core i3-3220 处理器（3.30GHz）上。

表 5.1 异构系统配置信息

Table 5.1 The configurations of the heterogeneous system

集群	CL1	CL2	CL3
节点数	9	6	3
已使用空间（块）	90	60	30
总容量（块）	900	600	300
CPU 频率（GHz）	3.2	2.5	1.8
与集群 CL1 之间的带宽（Mb/s）	30000	20	125
与集群 CL2 之间的带宽（Mb/s）	20	30000	60
与集群 CL3 之间的带宽（Mb/s）	125	60	30000

表 5.2 算法具体参数

Table 5.2 The parameters of the algorithm

实验参数	数值
迭代次数	任务数量大小的 10 倍
种群大小	50
交叉概率	0.7
变异概率	0.35

本文评估所提出的算法的性能主要是通过 Hadoop 的基准测试包括 WordCount，TeraSort 和 Sort[59]三种。

5.2 实验结果及分析

本节，主要从两方面来对实验给出的结果进行分析，首先对比了 MILP 算法和 DPGA 算法在求解时间上产生的开销。其次，在选择了效率更高的后者之后，通过三种常见的基准测试，将其与 Ceph 默认的 Average 算法以及性能优先的 Greedy 算法进行对比。

5.2.1 算法执行效率

表 5.3 展示了 MILP 算法和 DPGA 算法的对比实验结果。该表列举了在处理不同数量的 Map 任务（等同于不同数量对象）时，两种算法的对比情况。 “解空间

大小”指的是算法在计算过程中获得最优解时整个搜索空间的大小。通过表 5.3 观察得到，伴随着任务量的线性增长，解空间也呈现出线性的增长趋势。“最晚完成时间”代表每种算法可以得到的最优解是什么，对应于 MapReduce 而言，就是整个 MapReduce 程序的执行时间。“运行时间”则表示利用算法寻找最优解所花费的时间。“倍数”表示 DPGA 算法与 MILP 算法相比，前者在求解时间上提升的倍数。表 5.3 中的符号“×”代表使用 MILP 算法经过十个小时的计算，仍然无法找到最优解，这种情况由于已经远远超出了可接受的时间，所以这个结果可以认为是无解。

表 5.3 MILP 算法与 DPGA 算法时间开销的比较

Table 5.3 Comparisons of time cost between the MILP algorithm and the DPGA algorithm

Map 个数		4	5	6	7	8	9
解空间大小		1.17×10^9	2.33×10^9	4.54×10^9	8.60×10^9	1.59×10^{10}	2.88×10^{10}
MILP 算法	最晚完成时间	21150	24669	25658	33183	42044	×
	运行时间	5	26	57	619	36417	×
DPGA 算法	最晚完成时间	21150	24669	25658	33183	42044	43033
	运行时间	0.1	0.12	0.15	0.18	0.21	0.25
倍速		50	217	380	3438	173414	×

通过比较两者，虽然 MILP 算法理论上可以得到问题的最优解，但是可以看出随着任务数量地不断增长，求解过程的执行时间也会呈指数级增长。例如，当 Map 数量等于 4 时，MILP 算法求出问题的最优解需要花费 5 秒，但当 Map 数量等于 8 时，则需要花费 36417 秒。相比而言，本文提出的 DPGA 算法在任务数量为 4 的情况下只需要 0.1 秒，而当 Map 数量等于 8 时，也仅要 0.25 秒就可以求出任务的分配结果。显然，当问题的规模变大时，MILP 算法并不适用，即便它能得到问题的最优解，但是在实际生产过程中是无法容忍这么高的时间代价的。并且，可以发现 DPGA 算法几乎可以适用于任何规模的问题，因为可以从实验结果看出，它所花费的时间开销几乎是线性增长的。除此之外，当问题规模在很小的时候，DPGA 算法能够得到和 MILP 算法一样的最优解，并且只需要更少的时间开销。

5.2.2 基准测试对比

图5.2至图5.7是在由三个集群组成的异构环境下，进行基准测试的实验结果。实验内容是将本文提出的算法和其余两种算法比较，一种是基于性能优先的Greedy算法，另一种是Ceph默认的Average算法。之所以选择上述两种算法，是因为本文的提出的架构在性能方面关注的影响因素较多，据不完全调研，尚未有类似问题的相关算法。

基于性能优先的 Greedy 算法的基本思想是，在满足给定的负载均衡的约束条件情况下，尽可能多地将数据放在计算能力较强的节点上，是一种典型的贪心策略。而 Ceph 默认的 Average 算法主要是以数据的负载情况为依据，将数据分发到各个集群，使得集群间的负载差异尽可能的小。而负载仅根据磁盘的容量和使用空间决定，所以 Ceph 默认的 Average 算法没有考虑到计算性能和网络的异构情况。

实验结果通过绘制直方图来展现，这些直方图的横坐标列出的是从 0.075 到 0.125 之间，不同的负载均衡约束的阈值情况，纵坐标表示的是应用程序的执行时间，单位是毫秒。针对每个阈值，给出了三种算法的时间对比，从左到右依次是 Ceph 默认的 Average 算法，基于性能优先的 Greedy 算法，以及本文提出的优化算法。

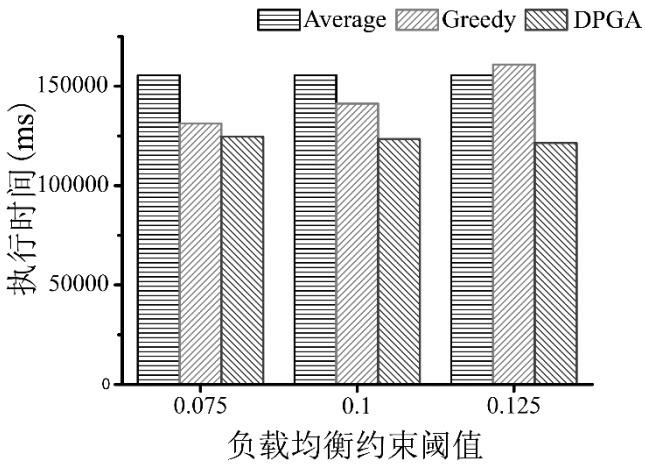


图 5.2 Map 任务数为 162 时，WordCount 基准测试对比图

Figure 5.2 The WordCount benchmark with 162 Maps

图 5.2 是 Map 任务数为 162，Reduce 任务数量为 18 时，WordCount 程序的运行结果，从图上可以看出 Average 算法和本文提出的算法随着约束阈值的增大，在任务执行时间上变化不明显。因为 Average 算法本质上就是优先使得系统拥有良好的负载均衡，所以不用考虑负载均衡约束的限制。而 Greedy 算法求出的数据放置策略会导致任务的执行时间变得越来越慢。比如，当给定一个负载均衡约束的阈

值 T = 0.075 时，利用 Average 算法得到的放置策略，程序运行时间为 155570 毫秒，利用 Greedy 算法得到的结果，程序运行时间为 131334 毫秒，本文提出的算法则仅需要 124600 毫秒，相比 Average 算法和 Greedy 算法分别提升了 19.9%和 5.1%。当阈值 T = 0.1 时，通过三种算法进行数据放置后，程序的执行时间分别为 155570 毫秒，141196 毫秒和 123462 毫秒，由于 Average 算法仅与数据量大小相关，所以这里还是相同的时间。相比之下，程序的运行效率分别提升了 20.6%和 12.6%。特别地，当 T 为 0.125 时，可以发现 Greedy 算法可能比 Average 算法的结果花费更长的运行时间，Average 算法为 155570 毫秒，而 Greedy 算法需要花费 160920 毫秒。这是因为 Greedy 算法试图把尽可能多的任务分配给由性能最好的 OSD 组成的集群 CL1。通过这种分配方式，可能会导致 CL1 超过负荷，那些分配在 CL1 上的任务将浪费更多的时间等待资源，并且使得 CL2 和 CL3 上的一些闲置资源没有得到充分利用。但本文提出的算法没有遇到这样的问题，仅需要花费 121516 秒，相比而言，效率分别提升了 21.9%和 24.5%。通过实验数据，可以发现随着所给约束阈值的增大，本文提出的算法效率不断提升，反映到结果上的表现是，执行时间逐渐减少，这是因为约束阈值增大意味着负载均衡限制的放宽，也就使得所提出的算法有更多可能的数据放置组合进行选择，继而得到更好的结果。

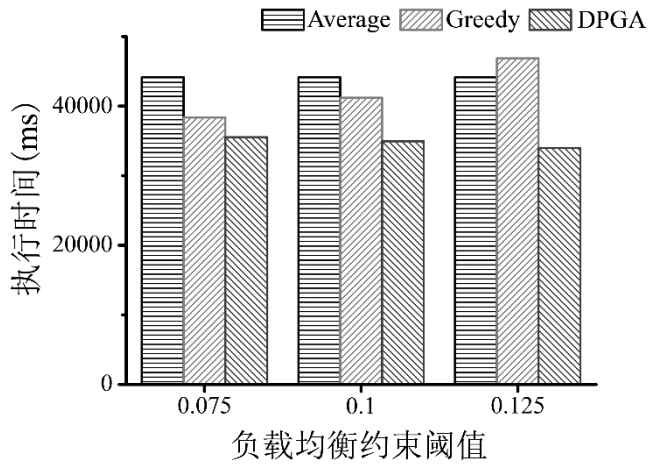


图 5.3 Map 任务数为 162 时，TeraSort 基准测试对比图

Figure 5.3 The TeraSort benchmark with 162 Maps

图 5.3 和图 5.4 分别是 Map 任务数为 162 时，TeraSort 和 Sort 的基准测试结果。整体趋势和变化情况与 WordCount 基准测试的情况相同，但在程序执行时间方面，相同任务量的 WordCount 要花费三到四倍左右的执行时间。这是因为 TeraSort 和 Sort 都是排序型的应用，相比于 WordCount 来说，不需要对文本进行切分，CPU 的计算量较小，最终的输出与数据本身的数据量大小一致，而 WordCount 的最终

输出通常要小于处理数据本身，因为排序不需要在 Reduce 阶段进行大量的合并相同键值对的工作，所以执行时间相对较短。

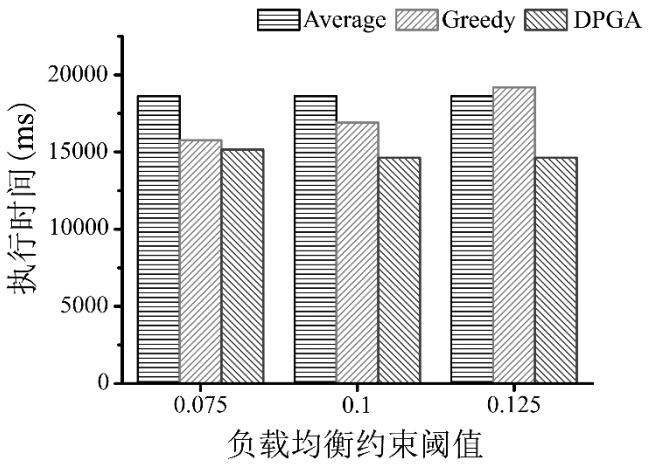


图 5.4 Map 任务数为 162 时，Sort 基准测试对比图

Figure 5.4 The Sort benchmark with 162 Maps

TeraSort 在负载均衡约束阈值分别为 0.075，0.1，0.125 的情况下，相对于 Average 算法和 Greedy 算法，分别提升了 19.5%和 7.4%，20.9%和 15.2%，以及 23.0%和 27.5%。Sort 在上述三种不同的阈值前提下，分别提升了 18.6%和 3.9%，21.4%和 13.5%，21.4%和 23.7%。可以发现在 Sort 基准测试中，当阈值为 0.1 和 0.125 时，使用本文提出的算法得到的结果运行时间都为 14628 毫秒。主要有两种可能情况：①因为在阈值增大之后，该算法没有在特定时间计算出更好的解；②因为在一定负载均衡约束的阈值范围内，已经求得了最优解，所以只有当限制阈值继续增大时，才有可能对问题的解造成影响。

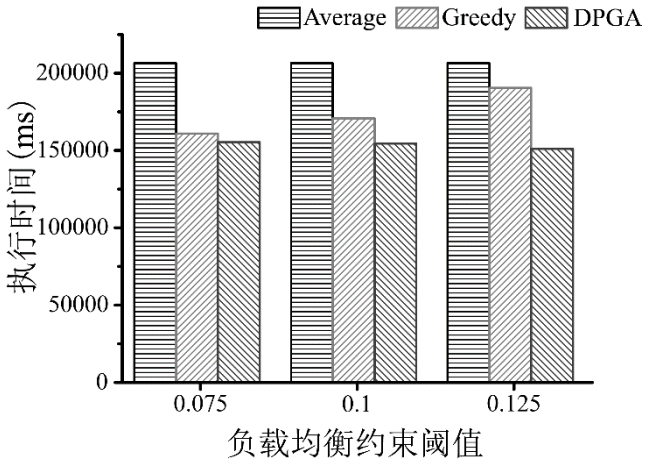


图 5.5 Map 任务数为 216 时，WordCount 基准测试对比图

Figure 5.5 The WordCount benchmark with 216 Maps

图 5.5 是 Map 数量为 216，Reduce 数量为 18 时，WordCount 基准测试的实验结果。前面三个实验中，在 T=0.125 时，使用 Greedy 算法得到的任务执行时间超过了使用 Average 算法得到的时间。而在图 5.5 中，当 T=0.125 时，三者分别需要花费 206384 毫秒，190506 毫秒以及 151058 毫秒。Average 算法相比于 Greedy 算法，仍旧需要花费更多的时间。通过分析可得，因为当数据规模越大时，系统的异构性对于系统整体的性能影响越大，所以采用 Average 算法要比其他两个算法在运行时间上开销更大。而 Greedy 算法由于考虑了设备的异构性，在数据规模变大的情况下，相比较 Average 算法可以获得更好的结果。

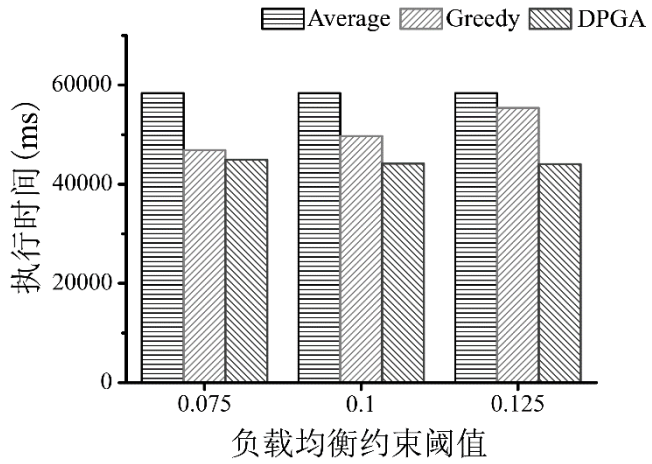


图 5.6 Map 任务数为 216 时，TeraSort 基准测试对比图

Figure 5.6 The TeraSort benchmark with 216 Maps

图 5.6 和图 5.7，分别是在 Map 任务数量为 216 时对应的 TeraSort 和 Sort 的测试结果。TeraSort 在阈值分别为 0.075，0.1，0.125 的情况下，相对于 Average 算法和 Greedy 算法，分别提升了 23.0%和 4.1%，24.4%和 11.1%，以及 24.6%和 20.5%。而 Sort 在上述三种阈值情况下，相比而言，分别提升了 24.3%和 3.3%，24.7%和 9.3%，以及 26.3%和 20.1%。

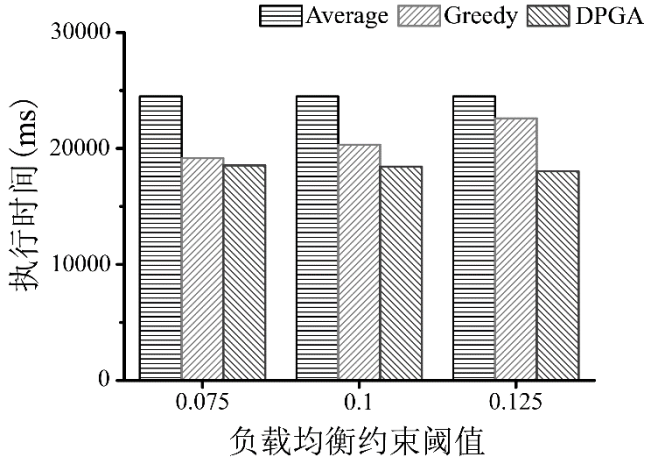


图 5.7 Map 任务数为 216 时，Sort 基准测试对比图

Figure 5.7 The Sort benchmark with 216 Maps

实验结果表明，与 Average 算法和 Greedy 算法相比较，本文提出的 DPGA 算法可以使 WordCount，TeraSort 以及 Sort 三个基准测试的平均完成时间分别降低 24.9%和 11.4%。并且在 WordCount 基准测试中，本文提出的算法与 Average 算法相比提高了 25.6%。

与上述另外两种算法相比，本文提出的算法有两个显著的优点。首先，对于一个给定的负载均衡约束的条件，它可以在数据规模较小时获得问题的最优解并在数据规模较大时，快速获得问题的最优解或者近似最优解。而不是仅仅通过将所有数据尽可能地分配在那些性能更好的存储设备上。其次，本文特别考虑了存储设备间不同的计算能力和集群之间不同网络的延迟差异，这些因素会很大程度影响整个应用程序的完成时间。实验结果表明，无论数据集的规模如何，DPGA 算法总是能够在不同的负载约束条件下，获得最好的数据放置策略，可以在不同程度上有效地缩短应用程序的执行时间。

5.3 本章小结

本章首先描述了具体的实验环境以及实验方法，然后从两个部分来阐述算法的有效性：①从计算时间和计算结果两个维度，对 DPGA 算法与 MILP 算法作比较，得出前者的优越性；②将 DPGA 算法与另外两种常见的放置算法进行对比，通过在两种不同数据规模大小的情况下，对比三种基准测试的结果，得出本文提出的算法针对于改进的 Ceph 架构是可行的，并且有显著的效果。

6 总结与展望

6.1 本文总结

本文主要研究的问题是，对原有的 Ceph 架构改进之后，根据 MapReduce 任务的特性，寻求最优的数据放置策略。

第一章节，首先对现有的分布式存储方案以及分布式计算框架的发展现状进行了一个概况说明，并在这一部分引出关于 Ceph 和 MapReduce 的发展现状。第二章节，主要针对 Ceph 的技术细节做了介绍，包括设计架构，读写流程以及 CRUSH 算法。并相应地对本文用到的分布式计算框架 MapReduce 做了大致的介绍，包括各阶段的数据流向等等。第三章节，通过分析现有 Ceph 架构的不足之处，如硬件异构，灾备策略不完善等，进而提出了一种基于 Ceph 改进的新的分布式架构。除此之外，还对改进后的数据分配，数据备份等过程进行了详尽的说明。第四章节，根据改进的 Ceph 架构，对系统和应用分别建模，并给出问题的定义，然后提出两种算法分别对定义的问题进行求解。第五章节，给出本文提出的两种算法在实际求解过程中执行时间的对比，以及所得结果的对比，并对其各自的优劣进行分析。接着，选择其中一种在实际中更适用的算法与另外两种常用算法进行对比。实验结果表明，在负载均衡的约束下，本文提出的算法在三种基准测试中，都能够使得系统性能得到显著地提高。

本文的研究成果有如下几点：

提出基于传统 Ceph 改进的新的 Ceph 架构。在原有的数据分配过程中，添加了一层概率表，用以将数据分散到位于不同地理位置的集群中，并在构建概率表的过程中，充分利用了在异构环境中，设备的计算能力和网络差异的特点，同时，还考虑负载均衡的约束。

对异构环境下面临的数据分配问题完成建模。在将 MapReduce 迁移到改进的 Ceph 分布式存储后，针对迁移之后面临的数据分配问题，分别对 Ceph 存储系统和 MapReduce 任务进行建模，将数据分配问题，转化为任务分配问题，然后进一步转化为求解任务的最晚完成时间。

提出用以解决 MapReduce 任务分配问题的 MILP 算法。将问题建模之后，通过对 MapReduce 的整个任务进行拆分，分析，对整个执行过程添加约束。并且，除了根据 MapReduce 的任务特性进行约束以外，还对系统内部集群的负载进行约束，进而利用该算法求取问题的最优解。

提出了一种高效的基于遗传算法改进的算法（DPGA 算法）。为了在短时间内可以求出数据放置方法，提出一种进化算法。算法借鉴了遗传算法的思想，将遗

传算法中每一个染色体当成一种数据放置策略。通过种群的迭代，对放置的位置进行变异或交叉，从而求出问题的近似最优解。

综上，本文所提出的架构以及算法可以有效地解决存储系统在异构环境下，面临的数据分配问题。并且该架构在灾备策略上，相比于传统的 Ceph 而言，安全性和可用性都大大提高。增加跨地域隔离级别之后，在遇到自然灾害等问题时，仍然可以保证不间断地对外提供服务。另外，本文提出的混合整数线性规划算法可以适用于任何使用 MapReduce 作为分布式计算框架的环境中。比之传统的 MapReduce 模型而言，增加了对 Shuffle 阶段的建模，使得整个计算框架更为完善。而 DPGA 算法，还可以适用于除了 MapReduce 任务以外的其他类似的数据放置问题。最后由实验结果给出的数据可以得出，该算法在改进之后的架构下，性能方面最高有 25.6%的提升，也证实了本文提出的架构与算法有一定的实际意义。

6.2 展望

在本文中，通过对 Ceph 的架构进行改进，分析，并对数据放置方法进行优化，旨在解决在异构环境中原有架构存在的问题。同时，在算法设计时，依然保证了原有 Ceph 中强调的负载均衡。

但在提出数据放置算法时，给出了一些假定条件，如在数据放置过程中，只考虑到了 Map 阶段的数据分配问题，而忽略了 Reduce 阶段的数据放置。又例如，在每个节点上，只有一个用于任务计算的槽；集群内部的节点为同构；在集群内部调用 CRUSH 算法时，数据分配完全均匀等。这些假设虽然在一定程度上比较合理，而且可以简化所研究问题的模型和求解复杂度。但是，如果要想完全适应生产需求，达到工业级的标准，必须要把这些影响系统整体性能的因素考虑在内。

除此之外，在实验过程中，假设了任务的执行时间仅由 CPU 频率决定。实际上，执行时间取决于很多因素，包括 CPU 频率、Cache 容量大小、内存带宽，磁盘转速以及 MapReduce 包含的配置参数等。本文在实验时，使用 MapReduce 时，采用的是默认的配置参数，但是针对不同的配置情况，程序的运行过程，各部分的时间开销可能会存在差异，所以应在多种不同的配置情况下，验证算法的有效性。而算法本身当数据规模较大时，可能无法获得最优解。所以为了获得问题的最优解，需要寻求一种更优秀的方法来求解问题。并且 DPGA 算法在执行效率上，仍有待改善。本文在对系统的高可用性方面验证时，受客观因素限制，无法很好的模拟实际的生产过程，可考虑在未来的研究中着重对该问题建立模型，用以模拟复杂的故障问题。

上述所说的这些问题，将会在今后的研究中加以考虑。

致 谢

转眼间，三年的研究生学习生涯就要结束了，从刚到重庆大学时，那个不经世事的学生，到现在具备勇气，热情澎湃，走向社会的青年。在这三年里，我从身边同学，老师，朋友身上学习到了很多东西，他们教会我如何在困难面前坚持，如何在恐惧面前坚强，如何在紧张面前放松。我仿佛还能清楚地看到研究生第一年班级篮球赛时，场上飞奔的我，也依然能想起那些数不清的在实验室写代码，调程序的昼夜。交作业，仿佛还是昨天的事，最后一节课，也好像才过了不久。也许未来可能再也没有机会，进入校园，但我也因此格外珍惜在这个地方遇到的人以及发生的故事。正因为有你们的存在，才让我变得懂事，学会成长。

首先，我要感谢冯亮老师对我的帮助，作为我的导师，教会我用严谨的态度去做任何一件事，哪怕是一些琐碎的细节。其次我要感谢沙行勉老师，在这三年里，教会了我如何对问题进行深度地思考，以及遇到困难时如何保持执着与冷静。这些不仅仅是在学术研究中该具备的素质，更是生活中难得可贵的精神。另外还要感谢诸葛晴凤老师和陈咸彰老师，在学术研究以及生活方面都给予了我宝贵的建议。除此之外，我还要感谢我的每一位代课老师。

然后，我要感谢我的师兄姜炜文，在很多大大小小的问题上给予我的帮助，还要感谢我的师姐，师弟和师妹。你们在我遇到问题时，同我一起并肩努力，攻克一个又一个的难关。在我写作过程中，也提出了很多有建设性的意见，以及教会我很多写作的方法。同时，我还要感谢我的父母，在我不知所措的时候站在我的身旁，为我提供坚强的后盾，耐心聆听我的想法，让我能在生活上和学习上都充满信心，你们是我人生最好的老师。

最后，十分感谢能够在百忙之中对论文进行评阅的老师，以及答辩现场的各位老师！

梁宇彤

二〇一八年四月于重庆

参考文献

[1] Ghemawat S, Gobioff H, Leung S T. The Google file system[C]//*SOSP*. 2003:290-43.

[2] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters[C]//*OSDI*. 2004:137-150.

[3] Chang F, Dean J, Ghemawat S. Bigtable: A distributed structured data storage system[C]//*OSDI*. 2006:305-314.

[4] The Apache Software Foundation. Apache Hadoop[EB/OL]. http://hadoop.apache.org. 2014.

[5] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system[C]//*MSST*. 2010:1-10.

[6] Lakshman A, Malik P. Cassandra: a decentralized structured storage system[C]//*SOSP*. 2010: 35-40.

[7] Seagate Technology LLC. Lustre[EB/OL]. http://lustre.org. 2018.

[8] Schmuck F B, Haskin R L. GPFS: A Shared-Disk File System for Large Computing Clusters[C]//*FAST*. 2002:231-244.

[9] MapR Technologies, Inc. MapR-FS[EB/OL]. https://mapr.com/products/mapr-fs. 2018.

[10] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system[C]//*OSDI*. 2006:307–320.

[11] IDC. Data Age 2025: The Evolution of Data to Life-Critical[EB/OL]. https://www.seagate.com/files/www-content/our-story/trends/files/Seagate-WP-DataAge2025-March-2017.pdf. 2017.

[12] Ceph Ecosystem[EB/OL]. http://www.wzxue.com/category/ceph-2. 2016.

[13] Sevilla M A. Mantle: a programmable metadata load balancer for the ceph file system[C]//*SC*. 2015:1-12.

[14] Ke Z, Ai H P. Optimization of Ceph Reads/Writes Based on Multi-threaded Algorithms[C] //*HPCC*. 2017:719-725.

[15] Zhang J, Wu Y, Chung Y C. PROAR: A Weak Consistency Model for Ceph[C]//*ICPADS*. 2017:347-353.

[16] Zaharia M, Chowdhury M, Franklin M J. Spark: cluster computing with working sets[C]//*HotCloud*. 2010:10-10.

[17] The Apache Software Foundation. Apache Flink[EB/OL]. https://flink.apache.org. 2017.

[18] The Apache Software Foundation. Apache Storm[EB/OL]. https://storm.apache.org. 2015.

[19] The Apache Software Foundation. Apache Beam[EB/OL]. https://beam.apache.org. 2018.

[20] Google Cloud. Cloud Dataflow[EB/OL]. https://cloud.google.com/dataflow. 2018.

[21] Tang Z, Liu M, Ammar A, et al. An optimized MapReduce workflow scheduling algorithm for heterogeneous computing[J]. Journal of Supercomputing, vol. 72, no. 6, pp. 2059-2017, 2016.

[22] Goiri I, Bianchini R, Nagarakatte S. Approxhadoop: Bringing approximations to mapreduce frameworks[C]//*ASPLOS*. 2015:383-397.

[23] Tian W, Li G, Yang W, et al. HScheduler: an optimal approach to minimize the makespan of multiple MapReduce jobs[J]. Journal of Supercomputing, vol. 72, no. 6, pp. 2376-2393, 2016.

[24] Cheng D, Zhou X, Lama P, et al. Cross-platform Resource Scheduling for Spark and MapReduce on YARN[J]. IEEE Transactions on Computers, vol. 66, no. 8, pp. 1341-1353, 2017.

[25] Cheng D, Rao J, Guo Y, et al. Improving Performance of Heterogeneous MapReduce Clusters with Adaptive Task Tuning[J]. IEEE Transactions on Parallel & Distributed Systems, vol. 28, no. 3 pp. 774-786, 2016.

[26] Dong Y, Milanova A, Dolby J. SecureMR: secure mapreduce using homomorphic encryption and program partitioning[C]//*PPoPP*. 2018:389-390.

[27] Oktay K Y, Mehrotra S, Khadilkar V. Semrod: Secure and efficient mapreduce over hybrid clouds[C]//*SIGMOD*. 2015:153-166.

[28] Wang Y, Fu H, Yu W. Cracking Down MapReduce Failure Amplification through Analytics Logging and Migration[C]//*IPDPS*. 2015:261-270.

[29] Fu H, Chen H, Zhu Y, et al. FARMS: Efficient MapReduce Speculation for Failure Recovery in Short Jobs[J]. Parallel Computing, vol. 61, pp. 68-82, 2017.

[30] Rahman M T, Gabriel E, Subhlok J. Performance Implications of Failures on MapReduce Applications[C]//*CLUSTER*. 2017:741-748.

[31] Coppa E, Finocchi I. On data skewness, stragglers, and MapReduce progress indicators[C]//*SoCC*. 2015:139-152.

[32] Nabavinejad S M, Goudarzi M. Energy efficiency in cloud-based MapReduce applications through better performance estimation[C]//*DATE*. 2016:1339-1344.

[33] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data[C]//*SC*. 2006:31-31.

[34] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares, and X. Qin. Improving MapReduce performance through data placement in heterogeneous hadoop clusters[C]//*IPDPSW*. 2010:1-9.

[35] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments[C]//*OSDI*. 2008:29-42.

[36] H. Chang, M. Kodialam, R. R. Kompella, T. Lakshman, M. Lee, and S. Mukherjee. Scheduling in MapReduce like systems for fast completion time[C]//*INFOCOM*. 2011:3074-3082.

[37] L.-Y. Ho, J.-J. Wu, and P. Liu. Optimal algorithms for cross-rack communication optimization in MapReduce framework[C]//*CLOUD*. 2011:420-427.

[38] F. Chen, M. Kodialam, and T. Lakshman. Joint scheduling of processing and Shuffle phases in MapReduce systems[C]//*INFOCOM*. 2012:1143-1151.

[39] C. Wang, Y. Qin, Z. Huang, Y. Peng, D. Li, and H. Li. Optas: optimal data placement in MapReduce[C]//*ICPADS*. 2013:315-322.

[40] Red Hat, Inc. Ceph Filesystem[EB/OL]. http://docs.ceph.com/docs/master/cephfs. 2016.

[41] Red Hat, Inc. Ceph Object Gateway[EB/OL]. http://docs.ceph.com/docs/master/radosgw. 2016.

[42] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn. Rados: a scalable, reliable storage service for petabyte-scale storage clusters[C]//*PDSW*. 2007:35-44.

[43] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling[C]//*EuroSys*. 2010:265-278.

[44] T. Beisel, T. Kenter, C. Plessl, M. Platzner *et al*. Performance-centric scheduling with task migration for a heterogeneous compute node in the data center[C]//*DATE*. 2016:912-917.

[45] J. Liu, Q. Zhuge, S. Gu, J. Hu, G. Zhu, and E. H. M. Sha. Minimizing system cost with efficient task assignment on heterogeneous multicore processors considering time constraint[J]. IEEE Transactions on Parallel & Distributed Systems, vol. 25, no. 8, pp. 2101-2113, 2014.

[46] S. Gu, Q. Zhuge, J. Yi, J. Hu, and H. M. Sha. Optimizing task and data assignment on multi-core systems with multi-port spms[J]. IEEE Transactions on Parallel & Distributed Systems, vol. 26, no. 1, pp. 1-1, 2014.

[47] W. Jiang, Q. Zhuge, X. Chen, L. Yang, J. Yi, and H. M. Sha. Properties of self-timed ring architectures for deadlock-free and consistent configuration reaching maximum throughput[J]. Journal of Signal Processing Systems, vol. 84, no. 1, pp. 1-15, 2016.

[48] Q. Zhuge, Y. Guo, J. Hu, and W. C. Tseng. Minimizing access cost for multiple types of memory units in embedded systems through data allocation and scheduling[J]. IEEE Transactions on Signal Processing, vol. 60, no. 6, pp. 3253-3263, 2012.

[49] C.-H. Liu, C.-F. Li, K.-C. Lai, and C.-C. Wu. Dynamic critical path duplication task scheduling algorithm for distributed heterogeneous computing systems[C]//*ICPADS*. 2006:365-374.

[50] T. N'Takpe and F. Suter. Critical path and area based scheduling of parallel task graphs on heterogeneous platforms[C]//*ICPADS*. 2006:3-10.

[51] J. Dummler and G. R ¨unger. Layer-based scheduling of parallel tasks for heterogeneous cluster platforms[C]//*ICA3PP*. 2013:30-43.

[52] L. Zhou, Y.-C. Wang, J.-L. Zhang, J. Wan, and Y.-J. Ren. Optimize block-level cloud storage system with load-balance strategy[C]//*IPDPSW*. 2012:2162-2167.

[53] Y. Xu, K. Li, J. Hu, and K. Li. A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues[J]. Information Sciences, vol. 270, pp. 255-287, 2014.

[54] Y.-S. Jiang and W.-M. Chen. Task scheduling for grid computing systems using a genetic algorithm[J]. The Journal of Supercomputing, vol. 71, no. 4, pp. 1357-1377, 2015.

[55] Wikimedia Foundation, Inc. Crossover_(genetic_algorithm)[EB/OL]. https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm). 2017.

[56] Wikimedia Foundation, Inc. Topological_sorting[EB/OL]. https://en.wikipedia.org/wiki/Topological_sorting. 2017.

[57] Red Hat, Inc. Using Hadoop with CephFS[EB/OL]. http://docs.ceph.com/docs/master/cephfs/hadoop. 2016.

[58] ESnet and L. B. N. Laboratory. iperf - the network bandwidth measurement tool[EB/OL]. https://iperf.fr. 2016.

[59] The Apache Software Foundation. Hadoop documentation[EB/OL]. http://hadoop.apache.org/docs. 2014.

附 录

A. 作者在攻读学位期间内发表的论文目录

[1] E.H.M. Sha, **Y. Liang**, W. Jiang, X. Chen, and Q. Zhuge. Optimizing Data Placement of MapReduce on Ceph-Based Framework under Load-Balancing Constraint[C]//*ICPADS*. 2016:585-592. **(CCF C 类，EI 索引)**

B. 作者在攻读学位期间内参加的科研项目

[1] 中国科技部国家高技术研究发展计划(863 计划), 面向大数据应用的新型内存计算系统软件及关键技术, 项目批准号: 2015AA015304, 2015 年 1 月至 2017 年 12 月.

[2] 中国科技部国家高技术研究发展计划(863 计划), 基于新型非易失性存储器的“统一内外存”系统结构及其关键技术, 项目批准号: 2013AA013202, 2013 年 1 月至 2016 年 11 月.

[3] 国家自然科学基金面上项目，基于非易失性内存的新型体系结构的系统优化关键技术, 项目批准号: 61472052, 2015 年 1 月至 2018 年 12 月.

[4] 华为 IT 产品部科研项目, 基于 SCM、NVM 介质和系统的关键技术研究, 2016 年 11 月至 2018 年 7 月.

[5] 华为中央研究院科研项目, SCM+NAND Flash 混合存储软件栈关键技术及算法研究, 2015 年 11 月至 2017 年 7 月.

C. 作者在攻读学位期间所获奖励目录

[1] 2016-2017 重庆大学优秀研究生.

[2] 2017-2018 研究生国家奖学金.

[3] 2017-2018 重庆大学优秀毕业研究生.