本科生毕业论文（设计）

题目：基于远程直接内存访问的分布式内存对象存储数据传输优化

姓　　名　兰 靖

学　　号　18340085

院　　系　计算机学院

专　　业　计算机科学与技术

指导教师　肖侬 (教授)

2022 年 4 月 12 日

基于远程直接内存访问的
分布式内存对象存储数据传输优化

Data Transfer Optimization of Distributed In-Memory Object Store Using RDMA

姓　　名	兰 靖
学　　号	18340085
院　　系	计算机学院
专　　业	计算机科学与技术
指导教师	肖侬 (教授)

2022 年 4 月 12 日

学术诚信声明

本人郑重声明：所呈交的毕业论文（设计），是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文（设计）不包含任何其他个人或集体已经发表或撰写过的作品成果。对本论文（设计）的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本论文（设计）的知识产权归属于培养单位。本人完全意识到本声明的法律结果由本人承担。

作者签名：

日　　期：　　年　　月　　日

【摘　要】

随着大数据处理、大规模智能模型训练、强化学习等多样化、高负载的计算需求成为学术界和产业界关注的焦点之一，作为计算基础设施的分布式计算框架（例如 Mapreduce, Spark, Ray）正在得到更多的关注和更广泛的应用。通常，计算框架软件负责为用户调度计算任务，同时管理和最大限度地利用计算集群的物理资源。上述场景所产生的快速增长的计算需求，正在使计算集群的性能愈发捉襟见肘：一方面，高性能计算 (HPC) 集群正在取代廉价的计算机器以应对高强度的计算。值得注意的是，前者所拥有的新型高性能网络，明显地区别于传统集群。以智能网卡为硬件基础，高性能计算机能够在无中央处理器（CPU）的介入下，对机器内存直接读写以传输数据。这种被称为远程直接内存访问（RDMA）的传输机制，相比以太网具有低占用、低延迟、高带宽等技术优势。另一方面，作为基础设施的软件框架，也在面临严峻的性能挑战：当前大多数分布式计算框架仍然缺乏对高性能硬件的支持，特别是无法充分利用集群中的高速网络资源。

Ray 作为面向强化学习等新型计算任务的分布式计算框架，同样存在着上述性能瓶颈。为此，通过设计高效、高性能的通信机制，我们能将现代网络的硬件优势转化为 Ray 系统整体的性能优势。本研究从该框架的核心组件：分布式内存对象数据库 Plasma 入手，首先测试和分析了其基于以太网和套接字传输数据而导致的性能瓶颈。然后，在高性能集群上，我们提出了一种支持 RDMA 特性的高吞吐对象传输机制。该机制针对大型数据对象，提出了基于单边读语义的传输协议，实现了用户态的内存零拷贝。并且该机制能够在运行时，根据数据大小动态地选择传输协议，以获得最佳性能。进一步，我们基于并行计算范式 MPI 构造了分布式、可扩展的多节点传输性能测试。在实验中，我们确定了传输协议的最优参数，并且展示了在天河高性能集群上，优化后的内存平台在常见大小的对象传输中实现至多 7 倍于原实现的吞吐率。

关键词： 数据传输，高性能网络，RDMA，键值对数据库，分布式缓存

[ABSTRACT]

Diverse and heavy computing workloads, including big data processing, large-scale model training, and reinforcement learning (RL), have increasingly drawn attention from the industrial and academic worlds. Distributed computing frameworks (Mapreduce, Spark, Ray, etc.) as the infrastructure recently also gained more attention and applications. Frameworks are responsible for scheduling users' tasks and efficiently utilizing the physical resources of the clusters. The growing need for computation that comes with the mentioned workloads challenges the performance of these machines. On the one hand, high-performance computing (HPC) clusters are replacing the traditional, cheap ones to handle intensified computation. These systems have distinct network architecture. With hardware support from smart NICs, HPC machines can directly read/write remote memory for communication. The transmission technique called Remote Direct Memory Access (RDMA) presents low occupancy, low latency, and high bandwidth as strengths that Ethernet doesn't own. On the other hand, the underlying software faces severe performance problems as well: most recent distributed computing frameworks cannot exploit high-performance hardware, especially high-speed networks with RDMA.

The above performance gap exists in Ray, a novel framework targeting emerging computing tasks such as RL. To handle this, we manage to turn advanced features of RDMA into better performance, by introducing a novel, high-performance transmission scheme. Our work focuses on Ray's core component: distributed in-memory object store named Plasma. We first identify the network bottleneck of Plasma through benchmarking and analysis. Then, we propose this scheme with hybrid transfer protocols. Besides the usual send/recv protocol, it has an extra one-sided read protocol for transferring large objects with zero-copy. For the best performance, we further implement a simple yet effective strategy to switch between the send-based and read-based protocols, given the sizes of the objects. Finally, we construct a distributed, scalable benchmark for testing the transmission overhead. Experiments demonstrate that, with the best-selected parameter, our improved memory store can achieve up to 7x throughput on our Tianhe HPC cluster.

Keywords: data transfer, high-performance network, RDMA, key-value database, distributed memory

目录

插图目录

表格目录

1 绪论

1.1 选题背景与意义

分布式计算框架是一种复杂的平台软件。传统的并行计算范式如信息传递接口（MPI[1]）和分区全局地址空间（PGAS[2]），通常为编程者提供丰富的调用接口和灵活的编程空间。然而，使用这些编程标准的门槛过高：编程者通常需要自己管理多台机器的状态，特别是内存的分配和使用；编程者还需要使用给定的标准原语，精确地规定多个进程之间的通信和协作方式，这通常需要大量时间和精力；另外，使用这些范式将导致程序和功能的强耦合——编程者很可能不得不重新编写代码来更新程序的逻辑。因此，现代分布式计算框架通常承担了上述硬件管理的角色，并针对目标任务类型，为用户设计尽可能少、但简单易用的功能接口，来降低集群的使用难度。早期的分布式计算框架，例如 Mapreduce[3] 和 Spark[4]，都是面向大规模数据分析这一类任务而设计的。然而，近些年以强化学习、复杂工作流等为代表的复杂计算需求，需要得到更灵活的框架支持。加州大学伯克利分校 RISELab 实验室提出的 Ray[5] 和芝加哥大学 Globus 实验室提出的 Parsl[6] 是其中两个典型。这些框架吸取了云计算领域“函数即服务（Function as a Service, FaaS）”的设计思想，能够细粒度地以函数为单位将任务调度到集群上执行，同时依然对用户隐藏绝大部分实现细节。Ray 通过分布式内存对象存储 Plasma[7]，实现了框架完全自主的集群内存管理，让用户可以专注于实现功能逻辑，而无需担心数据的存放和移动。Ray 在保持易用性的前提下，极大地提升了计算框架的灵活性，用户只需要修改几行代码就能将单进程程序扩展到整个集群，进而实现分布式机器学习等复杂模型。

高性能集群（或超算集群），是一种以高端处理器、并行加速卡、高性能网络、大容量存储为核心硬件的计算集群。随着大数据应用的丰富、超大规模人工智能模型的出现，高性能集群和超级计算机正在变得愈发重要。首先，高性能计算机拥有普通机器不能比拟的计算能力，主要表现为高端的多核处理器和并行加速卡。然而，随着大模型时代的到来，新应用对集群网络的需求快速提高，高性能网络所表现出的高带宽、低延迟等特性也逐渐受到了更多的关注。当前，高性能计算集群普遍使用的是英伟达 Mellanox 子公司的 Infiniband 高速网络[8]。这一网络架构的性能优势，很大程度上来自于对远程直接内存访问（RDMA）机制的支持。而对于传统使用套接字（Socket）通信的网络程序，该架构通过“基于 Infiniband 的

互联网协议（IPoIB）” 实现兼容。值得注意的是，这是一种依赖操作系统内核的非原生支持：已经有工作[9] 表明，其网络性能和直接使用 RDMA 技术相比具有明显差距。然而，要利用 RDMA，用户必须在应用中实现基于 RDMA 的通信机制，而不能依赖于操作系统内核提供的系统调用。因此，基于 Socket 的网络应用大多不能在高性能集群中直接获得显著的性能提升。分布式计算框架 Ray 并不例外，其分布式内存存储 Plasma 目前仅有对传统 TCP/IP 协议的支持，因而在超算集群上无法发挥出应有的网络性能，进而影响 Ray 运行在超算上的总体性能。

因此，本研究的目的是：我们是否能为分布式内存存储 Plasma，提出并实现一种支持 RDMA 机制的内存通信协议，从而让 Plasma 乃至整个 Ray 框架在现代超算集群上获得更好的性能？从超算研究的趋势来说，应用软件和先进超算硬件之间的隔阂，正在逐渐成为大家关注的热点。随着超级计算机和云计算两个领域的融合，会有越来越多的软件运行在高性能集群中。然而，它们中的大部分还没有针对高性能硬件提供软件支持——通过提供软件对高性能硬件的支持，我们能够将这些应用的运行性能提升到全新的水平。

1.2 国内外研究现状和相关工作

1.2.1 分布式计算框架

分布式计算框架是一种复杂的平台软件。在运行时系统层，它通过实现任务调度、并发执行、内存管理等基本组件，向用户透明地提供分布式计算的功能；在应用层，它通过设计和规范一系列接口，帮助用户高效地运行特定任务：例如大数据分析（Mapreduce、Spark），分布式机器学习（Ray）等等。Ray 是近年来最受关注的分布式计算框架，它以函数为单位调度任务执行，完全自主地管理数据移动，并且支持异步执行。借助这一平台，目前社区人员已经实现了机器学习（Ray ML）、强化学习（Ray RLlib）、模型部署（Ray Serve）、工作流（Ray Workflows）等上层应用库[10] 供用户使用。同时，用户也可以直接在框架上编写任意程序：使用常见的 Python 语法构建出完全分布式的计算程序，而且不会有任何表达能力上的限制。

代码 1.1 Ray 代码示例

```
import ray
ray.init()

@ray.remote
def f(x):
    return x * x

futures = [f.remote(i) for i in range(4)]
print(ray.get(futures)) # [0, 1, 4, 9]
```

在上述代码片段中，用户通过修饰符”@ray.remote” 将函数定义为远程函数，并在下方执行 4 次。值得注意的是，在 Ray 框架的调度下，这四次调用将会逐一调度到不同的进程上并发执行，因此 Ray 能够透明地为任何函数提供并行计算能力。而分布式内存存储 Plasma，则在系统中持续为并发任务调度所需的数据。在示例代码中，Plasma 将会在集群中查找变量 i 的位置，并将其转移到本进程中供任务使用。最后，Ray 原生支持异步执行，任何函数调用都会立刻返回，给予用户一个凭证（future[11]）——用户可以立刻用这一凭证作为参数继续调用其他函数，即使数据的真实值还没有求得。Ray 的执行引擎动态地构建和解决数据依赖，将凭证替换为真实值，然后继续调用依赖它的函数。这一异步特性将使 Ray 充分发挥并发任务的性能。

1.2.2 远程直接内存访问技术（RDMA）

RDMA 全称 Remote Direct Memory Access，即远程直接内存访问，是一种于二十一世纪之后逐渐兴起的新型网络通信技术。其核心思想和直接内存访问（DMA）相似——在早期计算机系统当中，内存读写、移动都需要由中央处理器（CPU）直接操作，从而带来相当的性能开销。而现代计算机系统分离出了内存子系统，将内存相关的负载卸载到子系统的专用硬件上。因此，CPU 只需要在访问开始和结束时同子系统协作，在漫长的访问延迟中可以执行其他计算任务，从而大大提高了计算机的总体性能。RDMA 技术在这一方向上更进一步：支持 RDMA 的现代智能网卡，不仅能从网卡端直接寻址并读写本机内存，还能够和远端的另一个网卡协作，直接读写远端机器的内存空间，从而实现通信。这一过程中 CPU 通常只需极少的介入、甚至完全不需要，因此具有相当明显的技术优势：

1) 旁路内核（Kernel-bypass）。基于 TCP/IP 协议的网络通信已经逐渐不能适应现代高并发、重负载的网络应用。基于高速 SSD 的存储系统、基于内存的缓

存和（键值对）数据库都需要在短时间内应对大量并发的 I/O 操作请求，工作[9] 表明这些应用的性能瓶颈位于 CPU，而不是 I/O 部分。臃肿的网络栈、用户态-内核态的切换、内核态的数据处理和拷贝等等，是影响网络系统性能的根本原因。智能网卡能代替 CPU 执行内存移动，而且 CPU 和网卡的大多数交互都实现在用户态，大大减轻了 CPU 的负担。

2) 零拷贝（Zero-copy）。零拷贝是当前操作系统领域的一项热门技术[12][13]，其思想在于提高内存的共享程度，例如减少进程-进程、用户态-内核态内存拷贝的数量，从而提高 I/O 的整体性能。在支持 RDMA 的应用中，由于内核旁路，一次内存操作往往能省去：用户缓冲到内核缓冲的拷贝、内核缓冲到硬件（驱动）缓冲的拷贝。数据直接在两端主存之间移动，因此产生可观的性能提升[12]。
3) 低延迟、高并发。简化的网络路径、零拷贝等特性大大降低了机器操作远端内存的延迟[14]。在并发应用中，更低的延迟意味着相同时间内更强大的并发处理能力。
4) 异步通信。RDMA 是原生异步的通信机制，进程需要主动访问完成队列（CQ）甚至直接检查内存数据，才能得知通信的发生。这一特性让程序的并发潜力大大提升，但于此同时，编程者需要自己设计同步机制来完成通信，同样增加了编程上的难度。

目前，RDMA 技术是智能网卡技术中较为成熟的一种。硬件支持 RDMA 的网卡通常都具有相当惊人的网络带宽以及其他诱人的硬件特性：目前，Mellanox NDR Infiniband 网卡能够支持高达 400Gb/s 的网络带宽；另外，Infiniband 标准实现了链路层的容错机制，这意味着通常意义上的丢包在 IB 网络上并不存在，大大降低了用户设计通信机制的难度。加上近十年来新型硬件的价格逐渐走低[9]，学术界和工业界争相尝试这一新技术，并已经有了相当多优秀的成果。尽管 RDMA 存在着其他实现方式，例如基于以太网的 RoCE，但本研究中的 RDMA 机制在硬件上基于超算集群中广泛使用的 Mellanox Infiniband 架构。

基于 RDMA 机制的网络编程，目前主流的做法是使用 Verbs 操作原语[8]。在 Infiniband 网络架构中，两端以队列对（Queue Pair，QP）图 1.1[8] 为基本模型进行通信，一次简单的发送-接收可以描述为如下步骤：

1) 两端程序各自创建一个队列对，并借助套接字等基础通信方式，交换队列对的基本信息，建立一对连接。
2) Infiniband 通信的基本单位是一个（发送/接收/完成）事务：接收方通过 Verbs 调用将一个构造好的接收事务压入到接收队列（Receive Queue）中。

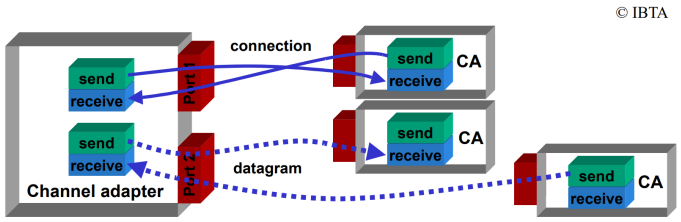


图 1.1 队列对（QP）示意图

3) 之后发送方将一个构造好的发送事务压入到发送队列（Send Queue）中。
4) 此时，硬件将为两端处理这一对事务，将本机内存从发送事务指定的内存地址发送到接收事务指定的远端内存地址。
5) 硬件完成一次事务操作后，通常将一个完成队列项（CQE）压入到完成队列（Completion Queue，CQ）中。
6) 用户进程可以在任何时刻通过弹出完成队列的头部来确认（发送/接收）事务执行完成，或者得到操作失败的错误代码。

以上流程实现了类似 TCP/UDP 的双边通信语义，不过，两者仍然有着本质的区别：RDMA 通信机制是基于事务模型的；以上操作均在用户态完成，无需任何系统调用；数据从一个进程直接发送到另一个进程。

除此之外，RDMA 机制还支持使用读/写（Read/Write）等单边通信语义。在执行这些单边操作时，远端应用不需要事先发送接收事务，也不会通过完成队列得知通信的发生。使用单边操作语义将会进一步降低远端 CPU 的介入和负担，从而支持更高强度的并发操作。不过，使用单边语义通常需要引入额外的同步机制以完成通信。

1.2.3 基于 RDMA 技术的内存系统

近年来，针对 RDMA 机制提供的高带宽、低延迟优势，研究人员在多个方向上尝试将其转变为实际应用上的性能提升。特别是在 RDMA 同时支持双边和单边语义的情况下，如何针对实际应用场景，设计合适的通信和协作机制，一直是这一领域研究的重点。从场景的角度来说，目前的主要研究方向是分布式内存系统和高并发的内存系统。

1.2.3.1 分布式内存系统

Infiniband 网络标准和 RDMA 技术最早应用于高性能计算（HPC）领域。俄亥俄州立大学的研究人员[15] 首次将 RDMA 技术应用于优化消息传递接口（MPI）的

通信机制，这也是最早的、较为完善的一个 RDMA 协作机制实现。这一设计针对 RDMA 单边操作所产生的协作困难问题，预先建立了固定映射的发送-接收缓冲区对，以及设计了自描述的消息块，深刻影响了后续 RDMA 通信方案的设计[9][16][17]。并且在 MPI 多年的演进中，其 RDMA 通信机制也不断有优化方案提出[18][19]。

然而，从集群内存空间的角度来看，RDMA 技术的逐渐成熟，让研究者看到了颠覆性、普适性优化的可能性：优异的访问带宽和极低的访问延迟，已经极大程度地拉近了机器内存之间的“距离”，使得集群规模的共享内存逐渐成为可能：Neugebauer[14] 发现 IB 网络的通信延迟和 PCIe 总线上的通信延迟处于一个数量级，这说明网络已经不再是集群通信的性能瓶颈。以 RDMA 为核心，HPC 社区和分布式系统社区均提出了一些分布式共享内存的实现方案，从而以较低的性能代价连接起整个集群的内存空间：前者包括 GasNet[20]，OpenSHMEM[13] 等支持分区全局地址空间（PGAS）计算的中间件；后者以 FaRM[21]，CoRM[22] 等系统为代表。这些研究的共同焦点是分布式的事务机制——除了本地内存的读-写冲突之外，还需要解决远端机器读写和本地访存之间的数据冲突。此外，也有研究避开了细粒度的内存管理，例如去中心化、可扩展的分布式页表系统 Infiniswap[23]，就通过粗粒度的内存页交换，较好地解决了集群中存在的物理内存使用不均的问题。

1.2.3.2 高并发内存系统

在另一个方向上，研究人员希望将 RDMA 低延迟、高并发的特性直接变为应用的性能——内存键值对（key-value）数据库是一个合适的领域。Redis[24]，Memcached[25] 等内存数据库通常被用作一种高速缓存，其他机器对数据库的访问也主要以并发读取为主，因此非常适合使用 RDMA 技术进行优化，特别是使用单边读等通信方式优化单机并发能力[9]。不过，基于单边操作的读取机制难以应对高强度并发中的数据冲突（Collision）问题，需要配合更优的哈希函数[26] 等其他优化手段才能获得较好的优化效果。因此近期的工作更是提出了双边语义和单边语义混合的通信机制[16][17]，从而在 CPU 负载、并发能力和编程难度上都获得较好的结果。总体来说，在过去的一段时间，针对应用场景不同，研究人员已经提出了众多具有不同性能特征的 RDMA 通信范式[27]。

1.3 论文主要研究内容

作为分布式计算框架 Ray 的核心组件，Plasma 并没有对 RDMA 通信的支持，因而在超算集群中存在性能提升的空间。该组件兼有分布式、高并发两方面的特

性：作为集群内存存储，Plasma 需要运行在集群的每个计算节点上，并且期望实现最大的并发传输能力，以支撑 Ray 快速的任务调度。不过，早期的 Plasma 并没有实现细粒度的并发机制，而是进一步依靠 Redis 等外部机制实现。这使得我们的研究重点倾向于优化大型数据对象的传输性能。在本文中，我们为 Plasma 提出了一种原生支持 RDMA 技术的通信机制，并且在现代超算集群上验证了其在各个数据大小的传输上都获得了更优的性能。

这一工作存在以下挑战：

1) 目前 RDMA 编程仍然是极为“小众”的技术，如何能在有限的资料和现有研究帮助下实现高性能的网络通信机制。
2) 如何在尽可能不破坏项目整体结构的情况下，为 Plasma 提供原生 RDMA 通信机制。这要求优化后的程序可以无缝地运行在以太网和 Infiniband 两种网络硬件架构上。
3) 针对 Ray 框架及其计算任务中可能出现的大小不一的数据，如何实现该机制使得 Plasma 能够在尽可能多的大小范围内都能获得最优的网络性能。

下面总结了本工作的主要贡献：

1) 我们通过实验分析了原 Plasma 实现在超算集群上的存储和网络性能，发现了其无法充分利用 Infiniband 高速网络的问题，从而驱使我们基于 RDMA 技术，研究 Plasma 数据传输的性能优化。
2) 我们针对小对象数据和分布式训练中常见的大对象数据，分别实现了基于双边和单边通信的传输机制。针对大对象，单边通信将实现用户态的零拷贝特性，从而降低了 CPU 数据拷贝而导致的负载和占用时间。
3) 我们基于消息传递接口 MPI 实现了分布式、可扩展的数据传输性能测试，比较了优化实现和原实现在各个数据大小上的性能。并且通过实验，我们确定了 RDMA 传输机制选择传输协议的最佳方案，从而在所有常见大小的数据传输上，我们的实现均显著优于原实现。

1.4 论文结构与章节安排

本文共分为五章，这些章节的内容安排如下：

第一章：绪论。简述了本文的研究背景和意义，简述了本研究的核心技术背景，并介绍了国内外相关工作和研究现状。

第二章：Plasma 分布式内存存储的架构和性能分析。这一章将简要分析 Plasma 的分布式架构，并且通过性能测试和常见内存存储 Redis 进行对比，分析了 Plasma

在传统网络结构上的性能瓶颈。

第三章:基于RDMA技术的混合通信机制实现。这一章将详细介绍基于RDMA技术的优化方案，并针对大小数据提出混合通信机制的实现。

第四章：实验和分析。这一章将在天河高性能集群上验证优化方案的性能提升，展示其在传输各大小数据上的显著优势。

第五章：总结和展望。这一章将总结本文的主要结果，并且进一步分析后续的工作方向。

2 Plasma 分布式内存存储架构和性能分析

2.1 Ray 架构简介

图 2.1[28] 展示了分布式计算框架 Ray 的整体架构，并展示了分布式内存对象存储 Plasma 在其中的位置和角色：

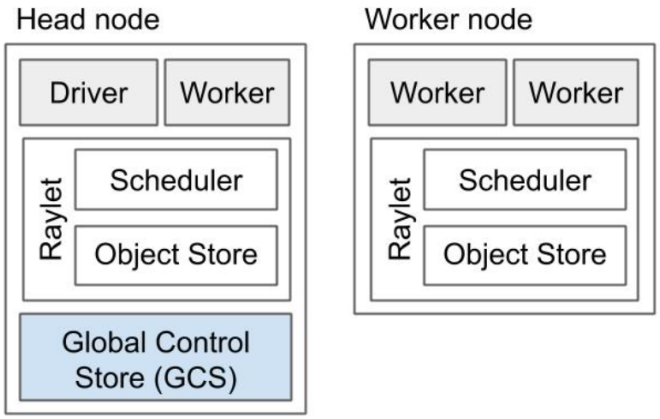


图 2.1 Ray 计算架构

Ray 架构简介：一个 Ray 集群由主节点（Head Node）和工作节点（Worker Node）构成。主节点上运行驱动（Driver）进程，即编写了 Ray 函数和其他逻辑（如节 1.1中代码片段）的 Python 程序；另外，还运行着全局控制存储（Global Control Store，GCS）：该数据库存放着 Ray 集群范围的关键配置信息。此外，所有节点都运行着 Ray 集群的核心中间件：Raylet。Raylet 主要由调度器和对象存储构成——调度器负责将 Driver 进程解析出的任务调度到集群的合适节点中运行；对象存储就是 Plasma 数据库，存放着计算所产生的数据——包括任务运行所依赖的输入数据和所产生的输出数据。

Plasma 工作机制： Plasma 对象存储运行在 Ray 集群的所有节点上。图 2.2[28] 展示了 Ray 驱动 Plasma 对象存储的基本机制：当 Ray 任务需要获取一个对象 x 以执行任务时，它将调用 “ray.get” 方法获取对象。如果该对象存放在本节点的 Plasma 存储中，后者将通过进程间通信（IPC）将对象传递给任务进程。否则，它首先需要以下图所示的方式从其他节点拉取数据：

1) 当前的 Ray 实现通过 “拥有（Ownership[11]）” 机制管理对象的存储位置。Plasma 向拥有该数据的进程（Owner）查询数据在集群中的位置。
2) Plasma 向该节点的另一个（位于节点 2 的）Plasma 进程发起对象请求。

3) Plasma 将数据传输并保存到本地。

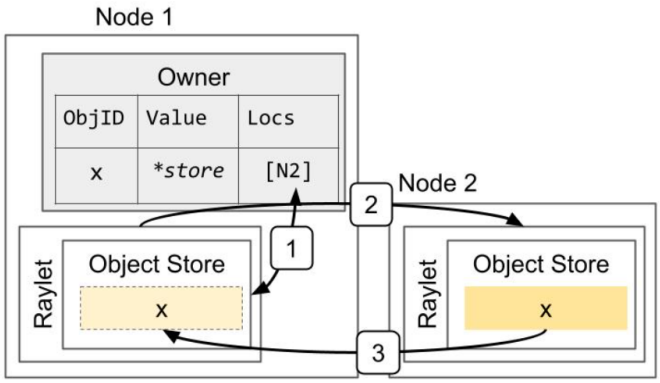


图 2.2 Plasma 工作机制

Plasma 对象的不可变特性：Ray 以及 Plasma 中的数据对象均是不可变的（Immutable）。这意味着，对已有内存对象做出任何修改都将生成一个全新的对象，并赋予不同的标识符。Plasma 的这一特性和 Redis 等键值对数据库有很大的不同：配合引用计数（Reference Counting）机制，不可变数据对象是分布式计算框架中常采用的一种设计。这对计算框架的设计有诸多好处：

1) 避免复杂的并发读写机制
2) 简化任务和资源调度机制的实现
3) 简化错误发现和恢复机制的实现

不过，不可变特性并不符合“零拷贝”的性能优化原则，因此会对计算框架的性能造成负担：由于对象存储之间、对象存储和任务进程之间的通信，对象传输机制存在较大的固定延迟，因此 Ray 频繁创建、移动小对象会显著影响其整体性能。

目前，Ray 的实际实现中规定仅有大于 100KiB 大小[28] 的对象通过 Plasma 对象存储实现集群共享——任务进程将更小的数据直接存放在进程的预留空间内，并通过将这些数据内嵌（inline）在 RPC 调用的参数中实现传递。这一实现策略指引我们在接下来的性能测试和分析中关注 Plasma 在大对象传输中的性能瓶颈。

2.2 Plasma 架构分析

Plasma 分布式存储架构由多个进程组成。通过将控制面、数据面上的任务解耦到不同的进程，我们可以较为方便地对其软件架构进行改进。图 2.3展示了 Plasma 集群的组织结构。

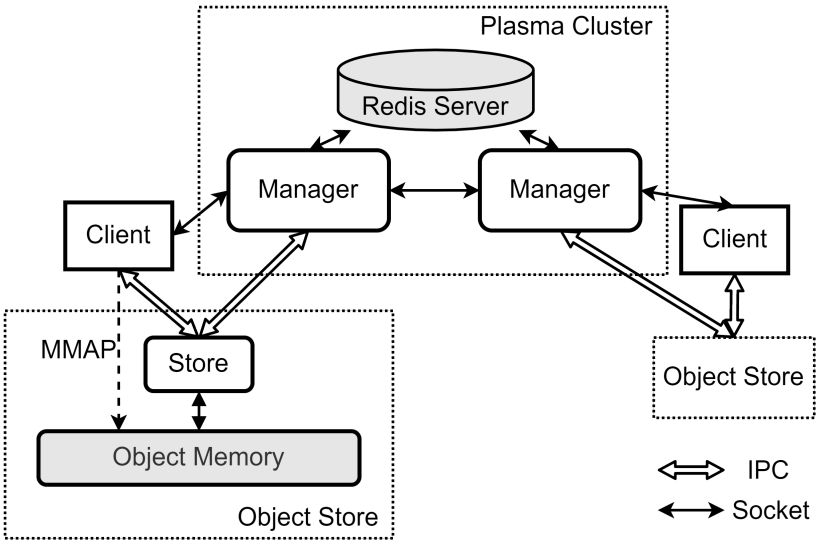


图 2.3 Plasma 存储架构

2.2.1 Plasma 对象存储架构

图 2.3中左下方部分展示了这一架构。对象存储的基本架构遵循用户端-服务端的特征。Store 进程负责直接操作和管理机器内存。当用户（Client）进程发起创建对象请求时，会将关键元数据——即对象编号（id）、元数据大小、数据大小等通过进程间通信（IPC）发送给 Store 进程。值得注意的是，Plasma 使用 Unix 域套接字（Unix Domain Socket）作为进程间通信方式，这一方式允许用户端以类似套接字的方式，通过操作系统内核与本机内的另一个进程交换数据。

Plasma 利用 mmap 映射机制对内存对象操作进行了性能优化。当前常见的内存数据库如 Redis，用户端进程和服务端需要直接交换数据以完成创建、读取等操作。而 Plasma 利用了 mmap 机制，使得客户端与服务端两个进程能够共享内存空间：

1) 创建对象时，服务端使用 mmap 映射分配内存空间，将其与一临时文件映射。
2) 服务端通过 IPC，将上述临时文件的文件描述符（file descriptor，fd）传递给用户进程。
3) 用户进程使用相同的文件描述符 fd 再次调用 mmap。

如此，mmap 向 Store 进程和用户进程返回同一个内存地址，用户进程能够直接读取、操作数据库中的内存对象。这一架构能够将数据库本地读取操作的开销优化到 $O(1)$ 时间复杂度，使 Plasma 针对大型对象仍然具有优秀的吞吐能力。Plasma 在对象存储架构中定义了如下操作原语：

表 2.1 Plasma 客户端接口

接口定义	接口描述
plasma_contain(id)	查询对象是否存在
buf ← plasma_create(id, fields, values)	创建参数/数据为 fields/values 的对象
plasma_seal(id)	封存对象使其对其他进程可见
plasma_release(id)	释放对象，引用计数减一（为 0 则删除）
buf ← plasma_get(id, fields)	读取对象
plasma_delete(id)	删除对象
plasma_connect(store_addr, manager_addr)	连接到 Store 和 Manager
plasma_subscribe(id)	等待一个对象在本机被创建

2.2.2 Plasma 集群架构

Plasma 存储将其集群架构以一个管理者（Manager）进程单独实现。多个 Manager 进程以及一个 Redis 数据库共同构成其集群架构，图 2.3中上方展示了这一架构。Plasma 在集群架构中定义了如下操作原语：

表 2.2 Plasma 集群接口

接口定义	接口描述
buflist ← plasma_fetch(ids)	将 id 列表中的对象拉取到本地存储

Manager 进程仅处理用户进程发出的一种请求，即输入一个 id 列表，执行一次拉取（fetch）操作：

1) Manager 对列表中的每个 id，从集群中查找对应数据对象的位置。
2) 本地 Manager 与存有这一对象的另一个 Manager 建立连接，拉取数据。
3) Manager 通过本地操作和 Store 交互，将数据保存到本地——对 Store 进程来说，Manager 只是普通的 Client 进程。

在这一过程中，Plasma 依赖于 Redis 提供的数据库服务。每当 Client 进程创建并封存数据对象时，它将借助 Manager 与 Redis 服务器建立通信，并更新该对象在集群中的分布。在这之后，集群中的其他 Manager 进程才能“看到”该对象的存在。因此 Manager 在拉取数据的过程中（上述步骤 2），必须先从 Redis 得到数据分布，才能开始一次数据传输。

2.3 基于套接字的 Plasma 通信机制

Plasma 实现了基于套接字（Socket）的数据传输机制。通过访问 Redis 服务器获得对象处在的目标节点后，Manager 会逐一尝试建立套接字连接，并拉取数据。

其通信机制如图 2.4所示：

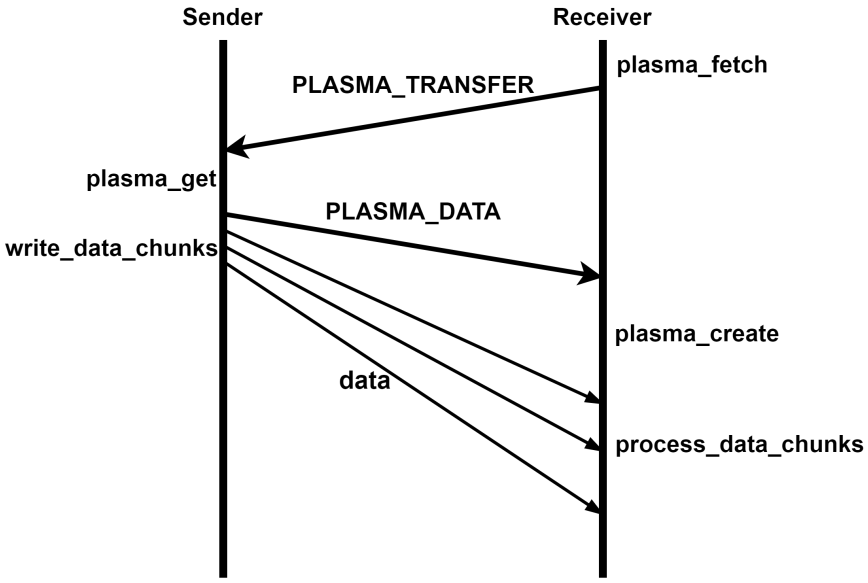


图 2.4 基于套接字的 Plasma 通信机制

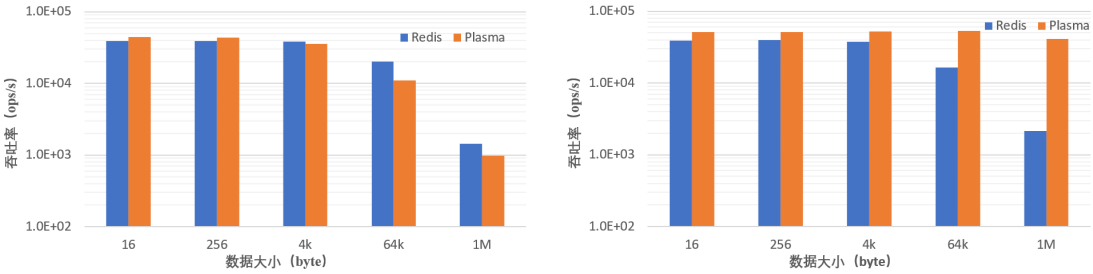
发起者（Receiver）首先将一条类别为 PLASMA_TRANSFER 的消息发送给存有数据对象的 Manager 进程。在得知需要发送的对象 id 后，发送方通过向 Store 调用查询操作获得了数据的缓冲区地址。需要注意的是，Plasma 的读取操作具有常数的时间复杂度，因而会马上返回。发送方（Sender）会将查询到的元数据（主要是数据大小）通过 PLASMA_DATA 消息返回给接收方，接收方便会向 Store 创建一个相同 id 的对象，获得分配的内存空间。最后双方分别进入发送/接收函数，接收方分批将数据转移到本地缓冲区中。在 Plasma 实现中，默认一次发送 4KB 大小的数据。

2.4 Plasma 存储和传输基准测试

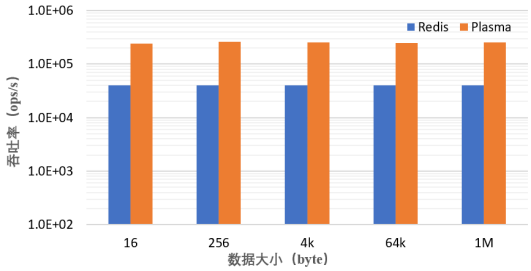
为了分析 Plasma 存储运行在超算上存在的性能瓶颈，得到可能的优化方向，我们在天河高性能集群上对 Redis 和 Plasma 进行了存储和传输数据的性能测试。通信性能测试以 IPoIB 机制运行在两节点的 Mellanox Infiniband 网卡上。

2.4.1 存储性能测试

如图 2.5所示，Plasma 单机存储架构的综合性能优于 Redis。Plasma 和 Redis 的 “put” 操作并不相同：对于前者，这一语义实际上是用户通过调用创建（Create）操作一段分配内存空间，然后将数据对象拷贝到该空间，最终释放（Release）并



(a) Put　　(b) Get



(c) Delete

图 2.5 Redis 和 Plasma 常见操作吞吐对比

封存（Seal）对象。可以看到，由于复杂的处理过程，Plasma 在这一操作上有稍高的延迟。由于 Plasma 通过 mmap 机制，在读取操作上支持了零拷贝特性，用户仅需通过 IPC 接收固定大小的文件描述符，因而在任何数据大小上都具有相似的性能。相反，Redis 使用 IPC 传输实际数据给用户进程。因此随着数据量增大到 1MB，Plasma 将对 Redis 有 20 倍的性能优势。Plasma 和 Redis 在删除上均拥有常数时间复杂度。

2.4.2　数据传输测试

进一步，我们在两个高性能节点上测试了 Plasma 和 Redis 在跨节点数据传输上的吞吐能力。对于 Redis，我们在远端节点启动 Redis 服务器，然后在本地基准测试中重复调用 Get 操作进行测试。对于 Plasma，在远端节点创建多个数据对象，然后在本地节点调用拉取（Fetch）操作将他们存到本地存储中。

如图 2.6所示，Plasma 在远程操作上的吞吐能力显著地劣于 Redis，在 1MB 大小的数据上前者有 3.6 倍的传输延迟。虽然，两者实际完成的语义很不相同——Plasma 需要先获取对象在集群中的位置，这本身就将访问一次 Redis 服务器；另外，Plasma 发起一次拉取操作包含了一次本地 Create 操作，这也将产生额外的延迟。而且，Plasma 一次仅以 4KB 为单位传输数据，也是其在大对象上传输性能明显不如 Redis 的原因。然而，从我们的实验结果总结来看，不论是 Redis 还是 Plasma

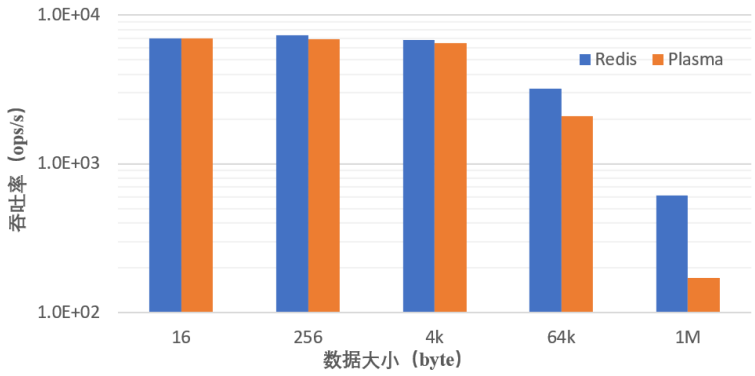


图 2.6 Redis 和 Plasma 数据传输吞吐对比

都存在明显的性能提升空间：它们均不能很好地利用超算网络的低延迟和高带宽。特别是支撑分布式计算框架的 Plasma，虽然相比 Redis 等单机数据库实现了分布式，但相当程度地牺牲了较大对象的传输性能，这是原实现没有解决的问题。

因此，这一观察促使我们基于超算网络对 Plasma 的数据传输机制进行针对性优化。在上述性能测试中，我们已经发现了 Plasma 在传输大对象时存在非常明显的性能缺陷，因此我们的优化重点应当放在大对象的传输机制上。不过，小对象的传输对网络延迟敏感，支持 RDMA 的传输机制也应该能降低其传输延迟。因此，我们希望优化的 Plasma 实现能够：

1) 为较大规模数据的网络传输提供明显更优的传输性能。
2) 同时，为较小数据的网络传输争取尽可能低的传输延迟。

3 基于 RDMA 的 Plasma 数据传输机制实现

在本章节中，我们将基于上一章对 Plasma 架构和性能的分析，提出并实现一种适应各大小数据的高吞吐数据传输机制。Plasma 中的跨节点通信单独实现在小节 2.2.2描述的 Manager 进程中。因此在本章节中，如无特殊说明，进程均指的是 Manager 进程。

3.1 连接的建立和复用

要进行双边（Send/Recv）和单边（Read/Write）通信，通信两端都需要首先为建立 Infiniband 连接。和套接字简单的连接过程不同，在双方能够通过发送事务传送数据之前，首先要完成如下准备工作：

1) 创建 IB 上下文（Context）；得到机器的 IB 设备列表，并打开其中一个设备（网卡）。用户可以在此时读取网卡的部分硬件参数。
2) 创建一个保护域（Protection Domain，PD）。这是一种管理机制，后续创建的数据结构只有处在同一个保护域中才能相互访问。
3) 创建发送/接收缓冲区，并将其注册为内存区域（Memory Region，MR）。这一操作的主要目的是使内存页面驻留在物理内存中，防止操作系统通过换页等机制将内存空间移出。由于 RDMA 实现在用户态，缓冲区的检查和数据收发都需要由用户正确地管理——否则会导致旧数据被覆盖等错误。
4) 创建完成队列（Completion Queue，CQ）。在双边通信中，用户通过拉取完成队列来检查通信事务的执行结果。
5) 双方进程创建并连接队列对（Queue Pair，QP）。使用可靠连接（RC）服务时，一对队列对在建立连接后只能相互发送数据。要建立一对连接，用户需要使用套接字等基础通信手段交换队列对的关键参数：网卡设备的本地编号（lid）和队列对编号（qp_num）。

IB 连接的建立流程有显著的时间开销。在实验中，我们通过测试发现这一流程将引入额外 300ms 左右的延迟开销。这一结果和 Frey 等[29] 的描述基本一致。尽管由于面向单机，这一开销在高并发内存系统的相关研究中并没有得到很大重视，然而这对于分布式内存系统 Plasma 来说是重要的。后者在每个节点都要部署进程，其 IB 连接以 $O(N^2)$ 速度扩大规模。加之 Plasma 具有不规律的通信特征，即便是基于套接字的实现，大量并发的连接请求也造成了显著的性能损失。因此在集群

中反复建立 IB 连接更是难以忍受的。

所以，我们必须在节点间复用 IB 连接。值得注意的是，步骤 3 创建并固定内存缓冲区占用了上述过程的主要时间开销，并且这一过程不依赖于队列对的连接。所以，在 IB 优化的 Plasma 实现中，我们先尝试建立队列对，**最后**才分配缓冲区。我们为每个进程维护了一个以网卡编号 lid 为键的哈希表。哈希表中的每个元素，以一个数据结构保存着一对 IB 连接的必要数据。因此，当两个进程尝试建立连接时，它们将首先通过套接字交换 lid，并查找哈希表。如果两个节点曾经建立过 IB 连接，那么双方将复用第一次分配好的缓冲区。通过这一改进，Plasma 仅有初次 IB 连接会感受到明显的传输延迟，而再次建立连接的时间开销大幅降低。

3.2 基于双边通信的传输协议

针对内存数据存储常见的较小数据，我们提出了基于双边通信机制的传输协议。这一机制的优势在于，能够充分利用两节点间**预先分配**的发送/接收缓冲区。其示意图如图 3.1所示：

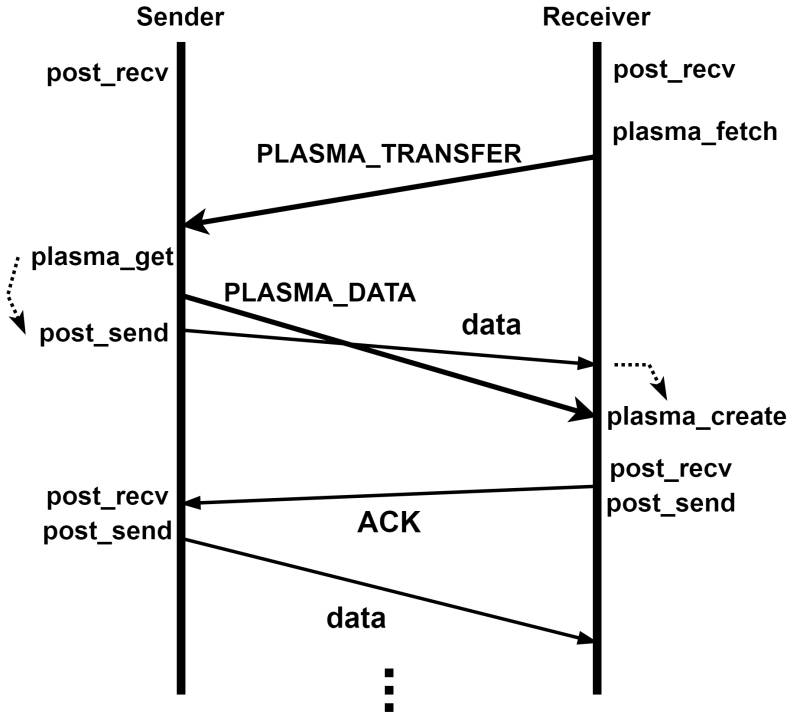


图 3.1 基于双边通信的传输协议

双边协议的实现：在该传输协议中，通信双方（Sender/Receiver）在握手发生前必须首先通过一次 post_recv 操作将一个接收事务放入到接收队列中。这是因为任何一个发送事务在完成时，连接的接收队列必须有可用的接收事务对应，否则通信无法发生。在本实现中，由于每个进程都维护着与其他进程的 IB 连接和对应

的缓冲区，因此做法是在连接建立时，进程都首先将一个接收事务放入到和该连接关联的接收队列中。这样我们就能确保第一次数据传输的握手发生前，接收队列总是存在可用的接收事务。

当数据发送方收到类型为 PLASMA_TRANSFER 的消息后，和套接字协议一样，它首先调用 plasma_get 函数得到了数据缓冲区的地址。然后，发送方将对象 id、数据大小等关键元数据通过 PLASMA_DATA 消息传递给数据接收方。之后，发送方将数据从得到的内存拷贝到预留的发送缓冲区，调用 post_send 将数据通过 RDMA 发送到接收方预留的接收缓冲区。值得注意的是：

1) 以此协议发送和接收数据，发送方和接收方在这一阶段都会产生一次用户态的内存拷贝。如图 3.1中虚线箭头所示：为了复用预先分配的内存缓冲区、避免重复的内存分配和注册，发送前数据必须要拷贝到固定缓冲区，接收时必须要从固定缓冲区拷贝。在较小数据的传输中，这两次内存拷贝的开销占比较低，因而是较优的通信方案。
2) RDMA 传输的数据和接收方调用 plasma_create 的顺序是不定的，但不影响该协议的正确性，图 3.1展示了其中一种顺序。数据可能先于元数据到达，也可能后于元数据到达。但不论事件以何种顺序发生，接收方都会在元数据到达后，调用 plasma_create 创建对象，然后轮询完成队列（CQ），确保对象数据已经到达。最后，接收方通过一次用户态的内存拷贝，将数据存到指定位置。在这里，轮询完成队列是一种同步机制——如果数据已经到达，它将立刻返回；否则将阻塞，直到数据在缓冲区准备就绪。这一现象展示了 RDMA 通信的异步性。

接受事务的补充：数据到达后，由于接收事务被消耗了，接收方在取出数据、确认缓冲区可覆盖之后，需要重新提交一个接受事务等待下次使用。在这一步，我们希望从完成队列里面获取接收数据的内存地址，以便将一个指向相同地址的接收事务补充到接收队列中。这一目的可以通过事务编号（Work Request ID）完成：IB Verbs 的事务编号是一个 64 位无符号整数，因此我们可以直接将接收地址作为事务的编号——这样还很方便地解决了事务编号的唯一性。

发送确认信息：和基于套接字的协议不同，此时接收方还需要发送一个确认（ACK）消息给发送方。这同样是一个同步机制。因为发送方并不知道接收方何时重新准备好了接收事务和缓冲区（而这在套接字编程中是无需担心的），这一 ACK 消息用于启动下一轮的数据发送。如果没有这一确认消息，发送方可能会在接收方还没有配对的接收事务时再次发送，进而导致程序错误。

3.3 基于单边通信的传输协议

双边协议的局限性：在实现了上述基于双边通信的传输协议后，在实验中我们已经观察到了相当的吞吐提升——即便完成传输所需的通信次数相比套接字协议可能更多，但 RDMA 通信的延迟极低。然而，进一步对程序在较大数据的传输测试中进行性能采集，我们发现当数据规模增加时，程序将大量 CPU 时间用于用户态的数据拷贝（即 memcpy 调用）。虽然，用户态的内存拷贝性能相当可观，在数据较小时产生的额外延迟并不显著，但随着数据增长到 1MB 大小，CPU 的处理能力无法跟上 RDMA 网卡的吞吐能力，成为了性能瓶颈。

对于上述双边通信的传输协议，选择用数据拷贝代替反复创建/销毁缓冲区是出于对后者较大开销的忌惮。这一前提在大型数据下是否还成立呢？结论是否定的，在第四章的实验和分析中，我们确认了内存空间分配 + 注册具有更好的扩展性：在数据较大时，反而是 CPU 直接复制同样大小的数据更耗费时间。这一观察驱使我们进一步为较大数据单独设计一个支持零拷贝的传输协议。我们实现了基于单边读（One-sided Read）汇聚（Rendezvous）协议，其示意图如图 3.2所示：

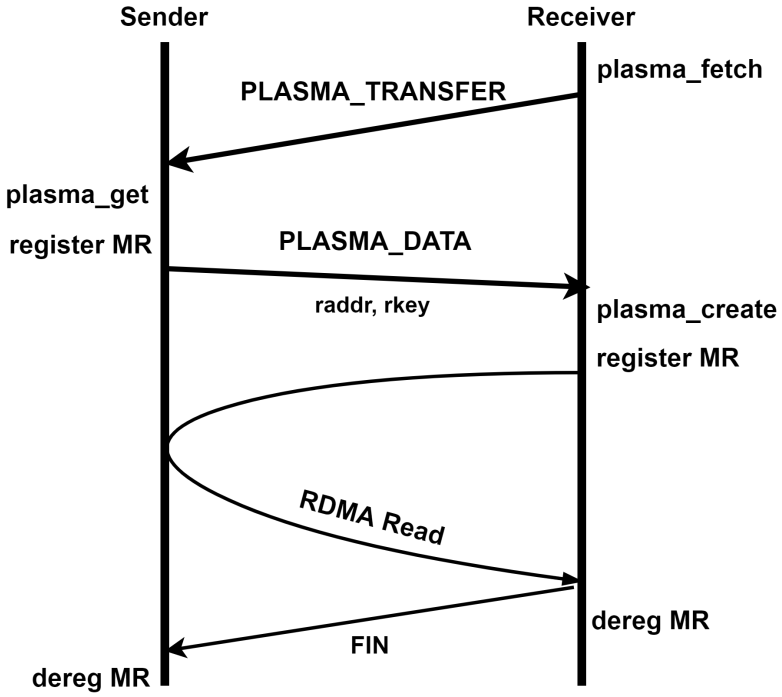


图 3.2 基于单边通信的传输协议

单边协议的实现：在基于单边读的传输协议中，使用了与传统通信原语极为不同的读语义：发送方不再事实上承担发送数据的任务，而是由接收方直接从远端内存拉取数据到本地。在接受到 PLASMA_TRANSFER 消息，且通过 plasma_get 函数调用获得了数据对象的地址后，发送方直接向 RDMA 网卡注册 MMAP 映射

返回的内存空间。因此，内存数据将在原地被网卡读取并发送，而不再依靠预先分配的固定缓冲区。接下来，接收方将会收到发送方传递的元数据、创建内存空间，并同样向 RDMA 网卡注册 plasma_create 返回的内存地址。需要注意的是，这一阶段发送方需要额外地将本地内存地址（raddr, rkey）发送到接收方处，才能正确地引导后者拉取内存数据。此时，双方的内存空间都已原地（in-place）成为可用的发送/接收缓冲区，接收方发出一个单边读请求，就能完成对象的复制。最后，接收方通过一个结束（FIN）消息通知对方注销缓冲区，也同时注销自己的缓冲区。一次数据传输就完成了。

单边协议的技术优势：在这一协议中，我们将内存数据所处空间原地注册为缓冲区，这一协议具有一些优势：传输仅需要两个来回（一次握手和一次传输），相比之下双边协议的来回数依赖于固定缓冲区的大小；实现比前者更简单；为发送方和接收方各节省了一次用户态的数据拷贝，实验表明这一特性在数据较大时具有明显性能优势。

3.4 传输机制的完整实现

两种传输机制的混合实现：上述的两种传输协议具有不同的性能表现，本质上是在两种固定开销中进行权衡：双边传输协议使双方增加了一次内存拷贝，避免了重复的内存注册/注销操作；单边传输协议每次传输都需要注册/注销内存，但真正实现了（用户态/内核态）内存零拷贝。直观上来说，前者适合发送小数据，而大型数据则应该采用后者传输。幸运的是，通过确定一个分界点，我们能将两种机制混合实现，而只需要对 Plasma 主逻辑做出微小改动。定义常量 IB_READ_MIN_SIZE 为采用单边协议的最小数据量，就能在函数调用前确定使用的协议。下方算法 3.1展示了 Plasma 发送端的处理逻辑:

算法 3.1: 服务端发送机制

输入: 连接上下文 $conn$，对象标识符 id

buf：缓冲区，S_d：数据大小，S_m：元数据大小
读取对象内存区域：$buf, S_d, S_m \leftarrow$ plasma_get(id)
发送元数据：send_message($conn, S_d, S_m, id$)
如果 $S_d + S_m \leq$ *IB_READ_MIN_SIZE* **则**
    发送本地地址：ib_send_read_info($conn, buf$)
    等待单边读：ib_wait_object($conn, buf$)
否则
    双边发送协议：ib_send_object_chunk($conn, buf$)
释放对象：plasma_release(id)

和发送方不同，接收方需要在收到对象元数据后在本地存储中创建该对象；并且，在通过 RDMA 接收数据并复制到合适位置后，还需要封存对象，使得对象能对集群中的其他进程可见。下方算法 3.2展示了 Plasma 接收端的处理逻辑：

算法 3.2: 客户端接收机制

输入: 连接上下文 $conn$

buf：缓冲区，S_d：数据，S_m：元数据大小，id：对象标识符
接收元数据：$S_d, S_m, id \leftarrow$ recv_message($conn$)
创建对象内存区域：$buf \leftarrow$ plasma_create(id, S_d, S_m)
如果 $S_d + S_m \leq$ *IB_READ_MIN_SIZE* **则**
    $raddr$：远端内存地址
    接收远程地址：$raddr \leftarrow$ ib_recv_read_info($conn$)
    单边读协议：ib_read_object($conn, buf, raddr$)
否则
    双边接收协议：ib_recv_object_chunk($conn, buf$)
确认对象：plasma_seal(id)
释放对象：plasma_release(id)

因此，优化后的 Plasma 程序能够在运行时通过判断拉取的数据大小来调整传输协议。那么，为了在不同的数据大小上都获得最优的传输性能，我们需要通过实验获得 IB_READ_MIN_SIZE 的最优取值。在之后的实验中，我们将对套接字协议、双边传输协议和单边传输协议分别进行详细的性能测试，从而确定出最优的参数，得到最优的总体性能。

IB 连接的完整实现：下方代码 11展示了支持混合传输机制的最终实现。每当

两个进程之间按照节 3.1所描述的流程建立一个 IB 连接时，它们将分别创建该数据结构的一个副本，并将其插入到哈希表中。之后，它们通过维护这个结构来完成后续的双边或者单边通信。

```
typedef struct {
  struct ibv_qp *qp;
  struct ibv_wc *wc;
  struct ibv_cq *cq;
  struct ibv_mr *recv_mr;
  struct ibv_mr *send_mr;
  struct ibv_mr *read_mr;
  uint8_t *ib_recv_buf; // send/recv buffers are pre-pinned
  uint8_t *ib_send_buf;
  uint8_t *ib_read_buf; // read buffer is on-the-fly pinned and registered
  int64_t bufsize;

  uint32_t rkey;
  uint64_t raddr;

  // key and handle to construct a hash table
  int slid;
  UT_hash_handle hh;
} IB_pair_info;
```

在该结构中，qp 为 IB 连接中本地的队列对结构，Verbs 通过它来确定消息的发出/目的地；cq 为完成队列，进程以此检查事务的完成情况；wc 则为存放完成队列项（CQE）的缓冲区——进程每次轮询完成队列 cq 之后，获得的元素都将暂存在这里等待检查。

此外，每个进程还要分别维护三个内存缓冲区 buf 和它们关联的内存空间（MR）。其中发送缓冲区和接收缓冲区如节 3.2所述，在连接建立时就将预先创建好，等待数据的发送/接收。而读缓冲区在此时则被赋值为空（NULL），只有在单边读协议发生时，该缓冲区才被即时（on-the-fly）分配、关联的内存空间同时被注册。如节 3.3描述，进程在单边读之前还需要获得对方读缓冲区的索引和内存地址，所以我们同样在结构体中为这些变量分配了位置。由于单边读协议并不复用缓冲区，一次通信完成后，进程还需要负责释放缓冲区和关联的内存空间。

4 性能测试与结果分析

4.1 实验环境配置

本段将简单介绍运行 Plasma 的必要环境配置。这包括进行性能测试的集群硬件配置，以及编译、运行 Plasma 所需的其他软件要求。

4.1.1 天河高性能集群硬件简介

运行测试的天河高性能集群有超过 100 台 CPU 服务器，并由 100GB 的 Infiniband 高速网络连接而成。每个 CPU 节点的关键配置如下所示：

表 4.1 天河 CPU 服务器硬件配置

硬件组件	数量	硬件型号	参数配置
CPU	2	Intel(R) Xeon(R) Gold 6150	18 核 @2.7GHz L1 Cache 64K L2 Cache 1024K L3 Cache 25344K
内存	12	/	16Gb DDR4 with ECC
以太网卡	1	Mellanox MT27710 Family [ConnectX-4 Lx]	25Gb/s
IB 网卡	1	Mellanox MT27700 Family [ConnectX-4]	100Gb/s

每个 CPU 服务器配备有双路 Intel 至强金牌处理器。每个 CPU 为 6 通道 16G 内存，因此每节点内存总量为 192Gb。两路处理器形成两个 NUMA 节点，每节点上分别挂载有一张网卡。该型服务器的硬件拓扑图如图 4.1所示。

需要注意的是，以太网卡和 IB 网卡分别挂在两个 CPU 的 PCIe Switch 上，因而 IB 网卡只有在访问本 NUMA 节点的一半内存时拥有最佳性能——而访问远离它的 CPU 所控制的一半内存将会有较大的性能损耗（图 4.1中红色访存路线）。这一特性对基于 RDMA 的网络通信性能有明显的影响，因此在实验中需要相应调整运行配置。

4.1.2 软件环境简介

表 4.2展示了本章节各性能测试所运行的软件环境。操作系统一栏展示了天河高性能集群使用的 Linux 操作系统版本；软件依赖指的是成功编译出可正常执行

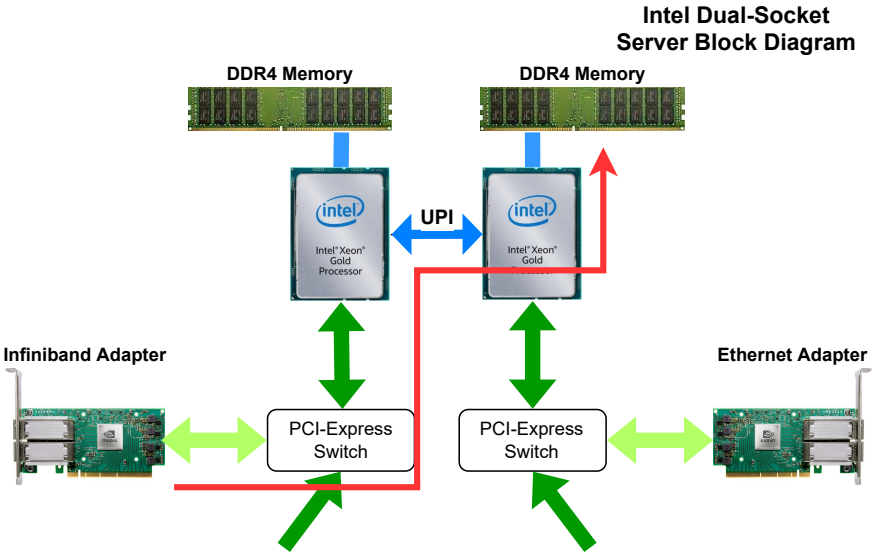


图 4.1 天河 CPU 服务器硬件拓扑

的 Plasma 程序所需要的前置软件；测试软件则表示为了编译基准测试程序、运行测试程序所需要的前置软件。所有软件均使用包管理器 spack[30] 安装和管理，软件名后的数字串表示这些软件在测试时使用的版本。

表 4.2 天河 CPU 服务器软件环境

软件类型	软件名称	作用描述
操作系统	CentOS Linux 7	Linux 内核版本 3.10.0-957.el7.x86_64
包管理器	spack@0.16.3	安装和管理下述软件依赖
软件依赖	Redis@3.2.3	Redis 服务器存放对象在 Plasma 集群的分布 ae 事件循环库驱动 Plasma 进程
	uthash utlist utarray utstring @2.0.1	基于宏的 C 语言头文件库， 封装了哈希表等高级数据结构
	gcc@9.3.0	编译 Redis 和 Plasma
测试软件	openmpi@4.1.1	编译基于 MPI 的多节点性能测试
	hwloc@2.5.0	查看服务器的 NUMA 拓扑结构
	numactl@2.0.14	控制 MPI 进程在 NUMA 架构下的绑核运行

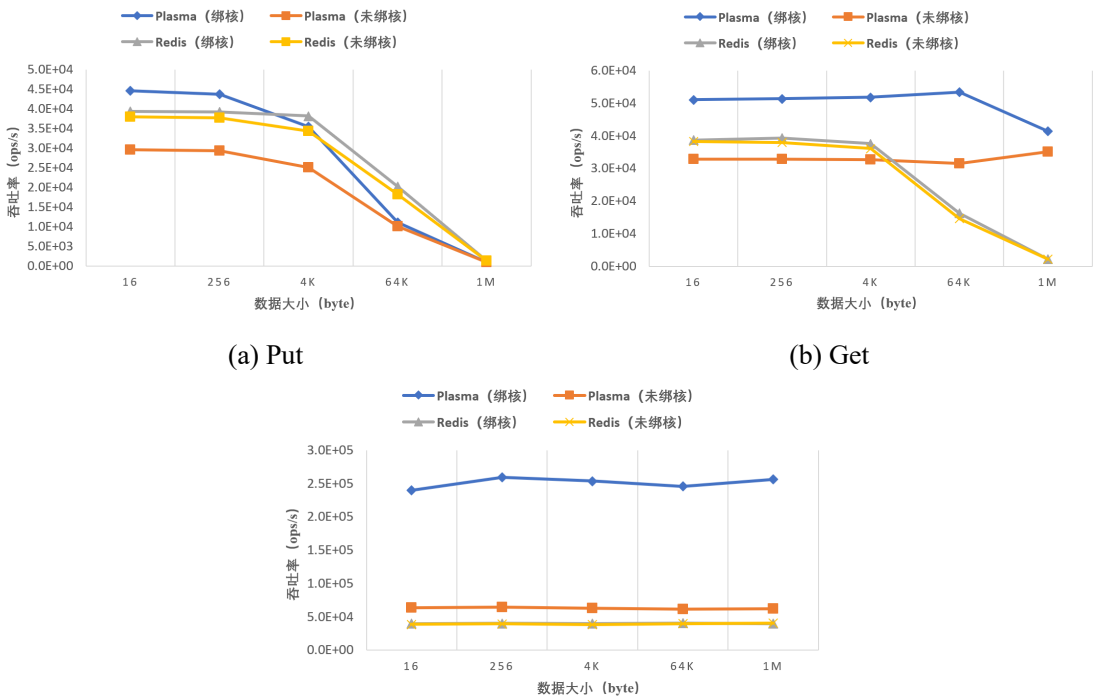
4.2 NUMA 架构下的 Plasma 存储测试

跨 NUMA 节点访存对性能测试的影响：上一章节中提到了天河高性能集群存在的 NUMA 访问架构。IB 网卡在 NUMA 架构中，对 “远端” CPU 所管理的内存发起内存访问需要通过两个 CPU 之间的 UPI 互联（The Intel Ultra Path Interconnect，

图 4.1中间蓝色连接），因而无法获得最优的访存延迟和带宽，对 Plasma 本地存储的性能产生较大的不利影响。由于 Plasma 传输数据前首先会（在发送端）访问或者（在接收端）创建内存对象，因而会进一步影响 Plasma 跨节点数据传输的延迟和吞吐能力。所以，为了获得最佳的传输测试结果，我们将首先通过简单的绑核以优化 Plasma 的本地存储性能。

在天河高性能服务器中，通过 hwloc 软件提供的 lstopo 命令可以查看 IB 网卡在服务器 NUMA 架构中所处在的位置：以太网卡挂载在 NUMA 节点 0 上，而 IB 网卡挂载在 NUMA 节点 1 上。这样，我们能通过 numactl 命令限制 plasma_store 和 plasma_manager 进程运行在 NUMA 节点 1，从而避免 IB 网卡的跨 NUMA 节点访存。

绑核性能测试：为了展示绑核对访存性能的影响，我们分别在绑核与未绑核的情况下测试了 Plasma 本地存储操作的性能，同时也对 Redis 进行测试作为参考。结果如图 4.2所示。很显然，不论是本地读还是本地写使用绑核操作后 Plasma 的程序性能都有了相当明显的提升：可以看到，在写入较小对象时，绑核的 Plasma 进程可以获得 1.5 倍的吞吐能力；不过，绑核运行时的写入性能随着数据增大逐渐下降，最后和不绑核时持平。在读取和删除对象操作上，绑核在所有数据大小上都有显著的优化效果，其中读取操作的吞吐率增加到原来的至多 1.55 倍，删除操作的吞吐率增加到原来的至多 5 倍。



(a) Put　　(b) Get

(c) Delete

图 4.2 绑核对 Plasma 常见操作吞吐率的影响

与之相比，绑核并不能给 Redis 带来更强大的吞吐能力——不论是否绑核，Redis 在三种常见操作上具有相近的吞吐能力；且在读取/删除操作中，Plasma 性能显著更优。这种区别很有可能是 Redis 和 Plasma 传输数据的实现方式不同而导致的：

- 如节 2.2所述，Plasma 使用 mmap 机制获取对象地址后，使用用户态内存拷贝（如 memcpy）进行数据读写。
- Redis 使用的是 TCP 环回地址（127.0.0.1）向服务器读写数据。

Redis 在访存上更依赖于操作系统，因此控制 Redis 进程的 NUMA 绑核并不会得到明显的性能提升。

由于本地访存的延迟大大降低，且 Plasma 对象传输过程包括了本地访存操作，因此绑核情况下的多节点传输测试更能够体现出基于 RDMA 的网络协议在传输性能上的优势。在之后的性能测试中，基于套接字以及 RDMA 的传输协议都将以绑核状态运行。

4.3 Plasma 网络协议测试

针对 Plasma 通过以太网络传输数据所导致的性能瓶颈，我们设计了原生支持 RDMA 通信机制的数据传输协议。针对可能出现的小对象，我们设计了使用预注册缓冲区的双边通信协议；并且，针对可能出现的大对象，设计了具有完全零拷贝特性的单边传输协议。因此，优化的 Plasma 通过混合两种机制，能在不同大小的数据都取得较好的吞吐能力。在 Plasma 优化实现的性能测试中，我们将测试基于 IPoIB 的套接字、RDMA 双边通信以及单边通信在各个数据大小上的传输性能，一方面验证 RDMA 机制在性能上的优越性，另一方面可以确定混合机制的关键决策阈值，从而获得最佳的综合性能。

性能测试案例的实现：我们实现了基于 MPI 编程范式的多进程网络性能测试。该测试能够在一次运行中对 Plasma 内存存储的主要操作（本地创建/读取/删除和远程读取）进行性能测量。该测试采用主-从架构，能够支持任意多从进程并发传输的测试。其流程如下所示：

1) （0 号）主进程阻塞在 MPI_Barrier 处，等待（编号为正整数的）从进程创建数据对象。
2) 其他进程创建一批数据对象。如果进程编号为 N-1，测试 Plasma 本地操作的性能并输出到日志。
3) 所有进程在 MPI_Barrier 处同步。之后，主进程通过 MPI_Gather 接口，从其他 MPI 进程处汇集所有内存对象的标识符。

4) 主进程以随机顺序向其他进程处发送 plasma_fetch 请求，拉取所有数据对象。

小对象传输性能：在天河高性能集群上，我们对 N=2，即点对点拉取数据的情况进行性能测试。如图 4.3所示，基于双边通信（IB Send/Recv）的传输协议在 16K 字节范围内的具有最好的传输性能：协议在该数据范围内的传输延迟，相较基于 IPoIB 的套接字协议和使用单边通信（IB Read）的协议，均有 20 微秒的延迟优势。在这一大小范围内，基于 IPoIB 的套接字协议和使用单边通信的传输协议性能相当——因此后者同样对双边通信协议具有显著的性能差距。这一结果验证了我们在章 3中的性能分析：即在较小数据的传输中，使用预注册的内存缓冲区和一次用户态的内存拷贝，相比重复注册内存缓冲区具有更小的时间开销，因而是更好的策略。

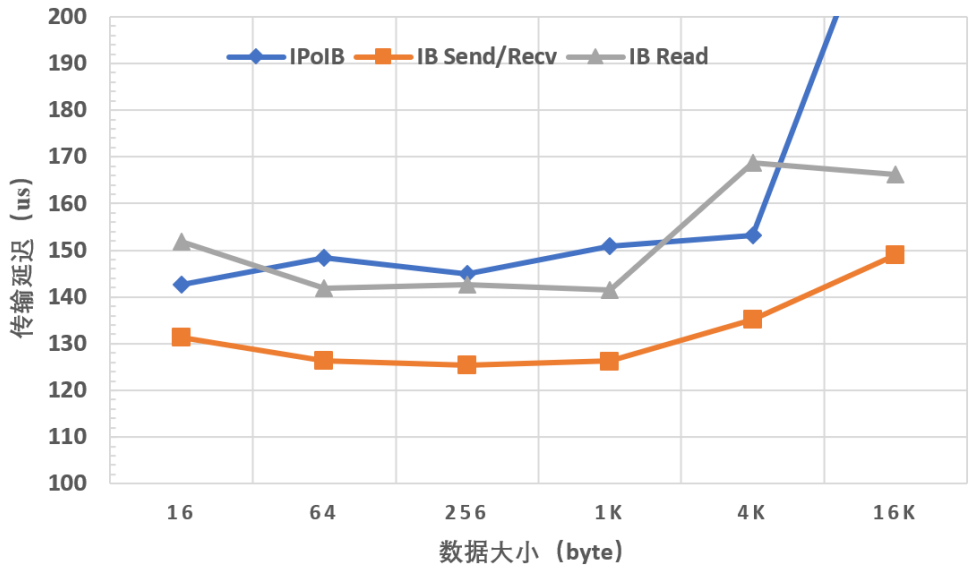


图 4.3 三种机制在小对象上的传输延迟

大对象传输性能：并且，上述性能对比在大于 16K 字节的数据传输中得到了相反的结果，进一步验证了我们对实现的分析。如图 4.4所示，原 Plasma 实现采用的基于 IPoIB 支持的套接字协议，在每次传输的对象大小大于 4K 时传输延迟开始快速上升。与之相比的是，随着数据量逐渐增加，使用 RDMA 机制的传输表现出了更好的扩展性：不论是基于 RDMA 的双边通信还是单边通信的传输协议，都表现出了明显的性能优势。使用套接字协议传输 1M 字节大小的内存对象，RDMA 双边传输协议可以在相似时间内传输四倍大小的数据量。进一步，在这一大小范围内，使用 RDMA 单边读的传输协议，对使用双边通信的传输协议也具有明显的性能优势。当对象大小大于 16K 字节时，前者表现出更优的传输延迟，并且该优势随着数据量增大而显著增大——在 4M 字节的对象传输中，使用单边 RDMA 操作能将吞吐率再增加一倍。在这一数据大小上，RDMA 的较优通信方案表现出了

7 倍于原实现的传输性能。

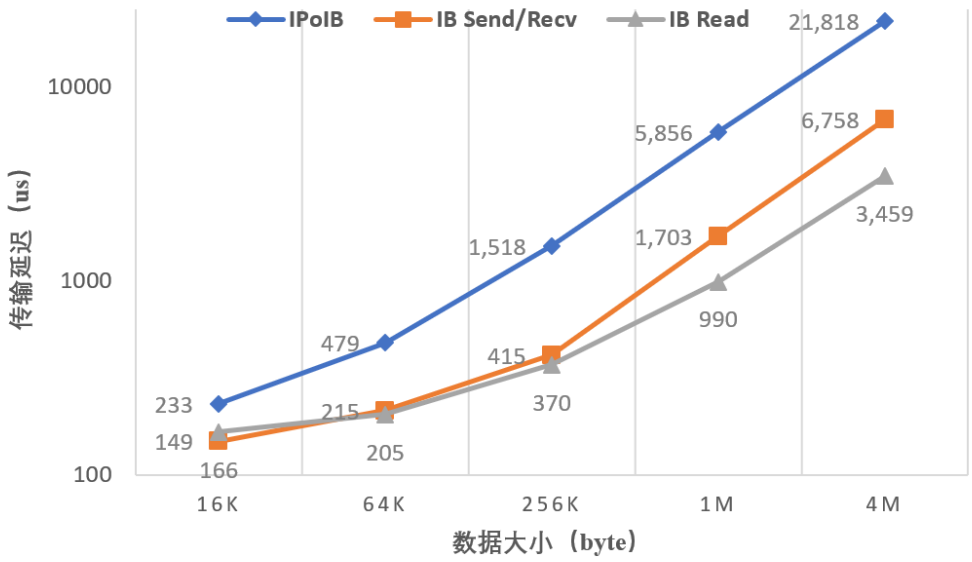


图 4.4 三种机制在大对象上的传输延迟

混合机制的选择：上述性能测试在各个常见的数据范围内，展示了原生 RDMA 机制对传统套接字通信的性能优势。并且，在章 3中我们实现了两种通信协议，并分析了它们的预期性能特征——这一讨论在实验中也得到了充分验证。为了让 Plasma 能在各个数据范围内充分利用 RDMA 机制的性能优势，我们通过观察实验数据，选择常量 IB_READ_MIN_SIZE 的值为 32KB。以该常量值为分界线，当对象大小小于该值时使用双边传输协议，反之使用单边传输协议。此时，我们能够将两种机制的性能特征简单而有效地结合起来，从而达到 Plasma 实现的最大优化。值得注意的是，最优的参数应当随着 CPU、IB 网卡、内存等硬件的性能发生变动。不过，通过我们在本研究中实现的性能测试程序，确定最优的策略应该是容易的。

与 Redis 对比：在确认了 Plasma 最佳的混合传输策略之后，我们可以将优化的 Plasma 实现和 Redis 作性能对比。两者传输各个大小数据的延迟如图 4.5和图 4.6所示。和节 2.4不同，经过 RDMA 传输优化后的 Plasma，已经能在常见大小的对象传输上获得比 Redis 显著更优的性能。然而还需要强调的是，Plasma 在传输测试中所完成的操作，明显要比 Redis 更多——Plasma 每次拉取对象到本地内存中，都会再次创建该对象、分配等大的内存空间、拷贝数据并封存；而 Redis 在性能测试中并不会这么做。总结来说，这是 Plasma 为了支持 Ray 分布式缓存、分布式访问所需的额外代价。然而，通过借助 RDMA 的硬件优势，我们优化了 Plasma 内存存储的传输机制，使其同时拥有了如下技术优势：

1) 支撑分布式计算框架 Ray 的集群内存管理，相比 Redis 数据库，Plasma 是分布式、可扩展的。
2) 即便是实现了分布式内存对象存储，Plasma 在任意单节点上的性能仍然显著地优于 Redis 数据库。

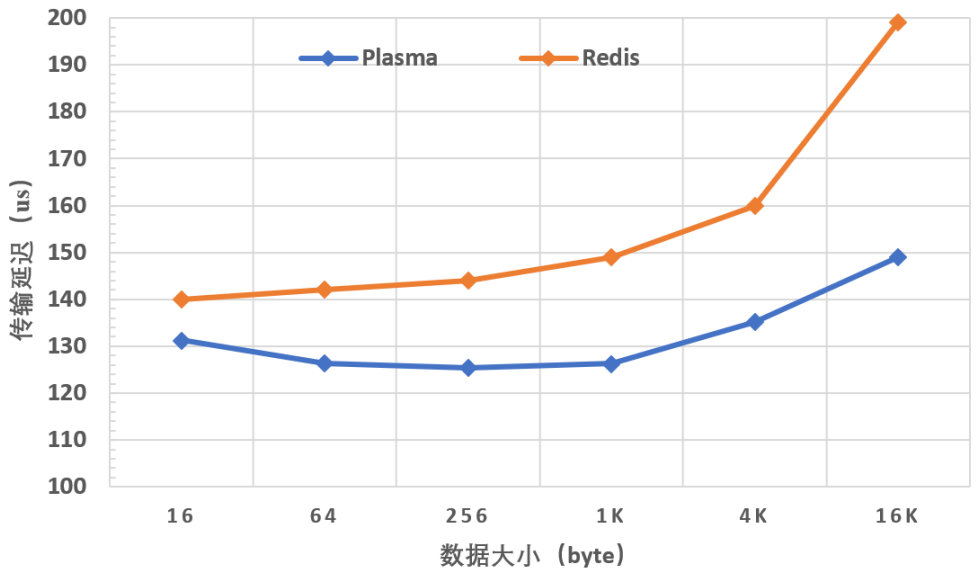


图 4.5 Redis 和 Plasma 在小对象上的传输延迟

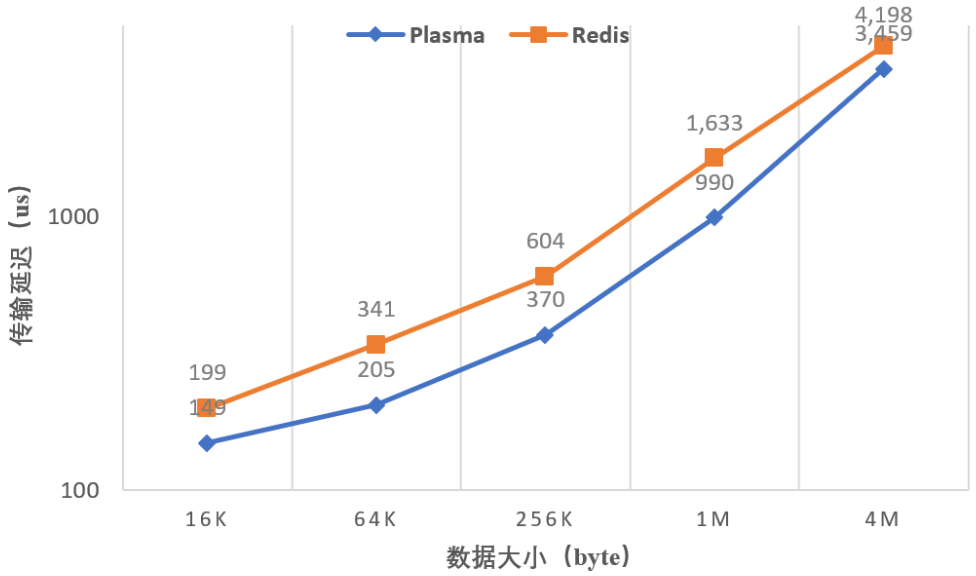


图 4.6 Redis 和 Plasma 在大对象上的传输延迟

5 总结与展望

5.1 本研究工作总结

在本文的研究中，我们探索了一种在最新的分布式计算框架和现代高性能硬件之间进行适配的性能优化。Plasma 作为支撑分布式计算框架 Ray 的分布式内存管理组件，在设计上并没有考虑过超算集群独特的硬件特性，特别是支持 RDMA 通信机制的高速网络，因而无法在超算集群上得到最佳的传输性能。我们首先通过跨节点的性能测试验证了这一设想，特别是发现了其在大对象传输上存在明显的性能缺陷。

之后，面对 Ray 产生的大小不一的内存对象，我们利用 RDMA 设计了一种全面优于原实现的对象传输机制。针对小对象的网络传输，我们基于 RDMA 双边通信原语，设计了一种低延迟的对象传输协议。该机制能够让接收方的本地访存操作和网络通信重叠，因此具有更优的传输延迟。此外，针对大对象的网络传输，我们使用 RDMA 单边通信原语设计了一种兼具低延迟、高带宽的传输协议。该协议所需的通信次数较双边协议更少，并且实现了零拷贝特性，因此大大提升了 Plasma 在超算网络上的吞吐能力。我们的优化实现能根据传输的数据大小从上述两种协议中选取合适的执行。

最后，我们对优化实现在各个大小的数据传输进行了性能测试。我们通过实验确定了混合传输机制的选择参数为 32KB。双边协议在小于 32KB 大小的对象传输上实现了最低的传输延迟。而对于大于 32KB 的数据对象，使用单边协议则提供了最优的传输带宽，能在 4MB 及以上大小的数据上提供接近一个数量级的性能提升。此时的 Plasma 相比 Redis 数据库，不仅提供了分布式内存管理的支撑机制，在单节点上的传输性能依然具有明显优势。

5.2 未来工作设想

本文的工作在没有大幅度变动 Plasma 代码结构的前提下，实现了显著的性能优化——然而，针对分布式内存对象存储，仍然有很多可以探索的空间：

1) 即便是采用了 RDMA 机制用作对象传输，Plasma 管理者进程仍依赖 Redis 事件循环库[31] 驱动函数调用。目前来看，这一部分仍未支持 RDMA，从而限制了 Plasma 在多节点上的可扩展性。因此后续可以设计基于 RDMA 的 RPC

调用组件进一步重构 Plasma，以提升其吞吐能力。

2) 目前的 Plasma 实现和 Ray 的对象管理机制[11] 没有做到紧耦合，因而存在协同设计（co-design）的可能性。特别是引入支持 RDMA 特性的高性能网络后，我们是否能进一步设计：适应 RDMA 高速网络的内存对象管理机制，以及支撑该机制的高性能内存存储组件。

参考文献

[1] GABRIEL E, FAGG G E, BOSILCA G, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation[C]//European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting. [S.l.]: Springer, 2004: 97-104.

[2] ZHENG Y, KAMIL A, DRISCOLL M B, et al. Upc++: a pgas extension for c++[C]//2014 IEEE 28th International Parallel and Distributed Processing Symposium. [S.l.]: IEEE, 2014: 1105-1114.

[3] DEAN J, GHEMAWAT S. Mapreduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113.

[4] ZAHARIA M, CHOWDHURY M, FRANKLIN M J, et al. Spark: Cluster computing with working sets[C]//2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10). [S.l.: s.n.], 2010.

[5] MORITZ P, NISHIHARA R, WANG S, et al. Ray: A distributed framework for emerging {AI} applications[C]//13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). [S.l.: s.n.], 2018: 561-577.

[6] BABUJI Y, WOODARD A, LI Z, et al. Parsl: Pervasive parallel programming in python[C]// Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing. [S.l.: s.n.], 2019: 25-36.

[7] The plasma in-memory object store[J/OL]. The Plasma In-Memory Object Store - Apache Arrow v7.0.0. https://arrow.apache.org/docs/python/plasma.html.

[8] PFISTER G F. An introduction to the infiniband architecture[J]. High performance mass storage and parallel I/O, 2001, 42(617-632): 102.

[9] MITCHELL C, GENG Y, LI J. Using {One-Sided}{RDMA} reads to build a fast,{CPU-Efficient}{Key-Value} store[C]//2013 USENIX Annual Technical Conference (USENIX ATC 13). [S.l.: s.n.], 2013: 103-114.

[10] Ray[J/OL]. Ray, . https://www.ray.io/.

[11] WANG S, LIANG E, OAKES E, et al. Ownership: A distributed futures system for {Fine-Grained} tasks[C]//18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21). [S.l.: s.n.], 2021: 671-686.

[12] TARANOV K, BRUNO R, ALONSO G, et al. Naos: Serialization-free {RDMA} networking in java[C]//2021 USENIX Annual Technical Conference (USENIX ATC 21). [S.l.: s.n.], 2021: 1-14.

[13] CHAPMAN B, CURTIS T, POPHALE S, et al. Introducing openshmem: Shmem for the pgas community[C]//Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model. [S.l.: s.n.], 2010: 1-3.

[14] NEUGEBAUER R, ANTICHI G, ZAZO J F, et al. Understanding pcie performance for end host networking[C]//Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication. [S.l.: s.n.], 2018: 327-341.

[15] LIU J, WU J, PANDA D K. High performance rdma-based mpi implementation over infiniband[J]. International Journal of Parallel Programming, 2004, 32(3): 167-198.

[16] KALIA A, KAMINSKY M, ANDERSEN D G. Using rdma efficiently for key-value services[C]//Proceedings of the 2014 ACM Conference on SIGCOMM. [S.l.: s.n.], 2014: 295-306.

[17] SU M, ZHANG M, CHEN K, et al. Rfp: When rpc is faster than server-bypass with rdma[C]//Proceedings of the Twelfth European Conference on Computer Systems. [S.l.: s.n.], 2017: 1-15.

[18] SUR S, JIN H W, CHAI L, et al. Rdma read based rendezvous protocol for mpi over infiniband: design alternatives and benefits[C]//Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming. [S.l.: s.n.], 2006: 32-39.

[19] WANG H, POTLURI S, BUREDDY D, et al. Gpu-aware mpi on rdma-enabled clusters: Design, implementation and evaluation[J]. IEEE Transactions on Parallel and Distributed Systems, 2013, 25(10): 2595-2605.

[20] BONACHEA D, HARGROVE P. Gasnet specification, v1. 8.1[R]. [S.l.]: Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2017.

[21] DRAGOJEVIĆ A, NARAYANAN D, CASTRO M, et al. {FaRM}: Fast remote memory[C]//11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14). [S.l.: s.n.], 2014: 401-414.

[22] TARANOV K, DI GIROLAMO S, HOEFLER T. Corm: Compactable remote memory over rdma[C]//Proceedings of the 2021 International Conference on Management of Data. [S.l.: s.n.], 2021: 1811-1824.

[23] GU J, LEE Y, ZHANG Y, et al. Efficient memory disaggregation with infiniswap[C]//14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17). [S.l.: s.n.], 2017: 649-667.

[24] Redis[J/OL]. Redis. https://redis.io/.

[25] JOSE J, SUBRAMONI H, LUO M, et al. Memcached design on high performance rdma capable interconnects[C]//2011 International Conference on Parallel Processing. [S.l.]: IEEE, 2011: 743-752.

[26] PAGH R, RODLER F F. Cuckoo hashing[J]. Journal of Algorithms, 2004, 51(2): 122-144.

[27] LI T, SHI H, LU X. Hatrpc: hint-accelerated thrift rpc over rdma[C]//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. [S.l.: s.n.], 2021: 1-14.

[28] Ray architecture whitepaper[J/OL]. Ray Architecture Whitepaper, . https://docs.ray.io/en/latest/ray-contribute/whitepaper.html.

[29] FREY P W, ALONSO G. Minimizing the hidden cost of rdma[C]//2009 29th IEEE International Conference on Distributed Computing Systems. [S.l.]: IEEE, 2009: 553-560.

[30] GAMBLIN T, LEGENDRE M, COLLETTE M R, et al. The spack package manager: bringing order to hpc software chaos[C]//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. [S.l.: s.n.], 2015: 1-12.

[31] Redis event library[J/OL]. Redis Event Library. https://redis.io/docs/reference/internals/internals-rediseventlib/.

致谢

兰靖

2022 年 4 月 12 日