MPC 的通信架构设计思路

梁俊勇

2025-11-21

Contents

MPC 通信常见问题

在底层通信里，我们常常会遇到这样的场景：
一段代码需要重复调用 send()或者 recv()方法
但是这里有较大的性能问题

场景 1：一方向另一方发送多个数据


```python
# 发送方
for item in items:
    socket_to_peer.send(item)
```

```python
# 接收方
for i in len(items):
    items[i] = socket_to_peer.recv()
```

场景 2：一方向多方发送数据


```python
# 发送方
for peer in peers:
    socket_to_peer.send(data)
```

```python
# 接收方
for peer in peers:
    data = socket_to_peer.recv()
```

这其中的问题在于，每次发送与接受，都会存在数据的依赖关系。具体表现为：

- 阻塞式操作： 同步编程中，`send()` 和 `recv()` 是阻塞的，程序会暂停执行，直到当前通信操作完全完成。
- 严格串行： 在循环中，后一次的通信操作（无论是发送还是接收）必须等待前一次操作彻底完成后才能启动。
- 性能瓶颈： 这种强制性的顺序等待，极大地限制了通信效率，尤其是在需要与多个参与方进行大量数据交换的场景下，导致系统无法充分利用并行处理能力和网络带宽。

以场景 2 为例

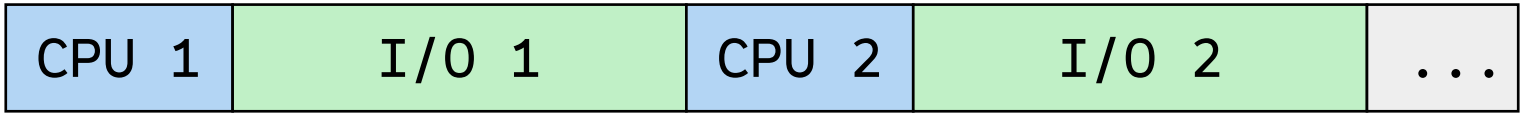
Python

```python
# 手动展开发送方循环代码
# peers = [alice, bob, carol]
# for peer in peers:
#     socket_to_peer.send(data)

socket_to_alice.send(data)
socket_to_bob.send(data) # 在alice没有确认收到数据前，此行代码不会执行
socket_to_carol.send(data) # 在bob没有确认收到数据前，此行代码不会执行
```

以场景 2 为例

Python

```python
# 手动展开接收方循环代码
# peers = [alice, bob, carol]
# for peer in peers:
#     socket_to_peer.recv(data)

socket_to_alice.recv(data)
socket_to_bob.recv(data) # 在没有接收到alice数据前，此行代码不会执行
socket_to_carol.recv(data) # 在没有接收到bob数据前，此行代码不会执行
```



串行化的计算与处理

问题 2： 传统通信协议的瓶颈

我们常用的 TCP 和 HTTP/1.1 协议虽然可靠，但在大规模、高并发的 MPC 场景下会遇到显著的性能瓶颈：

- 连接开销大：
  - TCP 三次握手： 每次新建连接都需握手，带来固有延迟。
  - TCP 慢启动： 连接初期传输速度受限，短连接无法有效利用带宽。
- 协议效率低：
  - HTTP/1.1 队头阻塞： 单个慢请求会阻塞同一连接上的后续所有请求。
  - 冗余的报文头： HTTP 头部是文本格式，存在大量冗余信息。
- 数据传输未经优化：
  - 默认无压缩： 协议本身不强制压缩数据，增大了网络负载。
  - 序列化开销： 使用 JSON 等文本格式比二进制格式体积更大，处理速度更慢。

预备知识

在我们讲解后续思路与方案之前，我们先来了解一下异步编程。

异步编程是实现并发的一种主要方式。

我们可以用在餐厅点餐的例子来理解：

- 同步（等待的方式）：　你在柜台点完餐，然后就站在那里一直等到餐做好。中间什么也做不了。
- 异步（不等待的方式）：　你在柜台点完餐，服务员给你一个震动的取餐器。你可以回到座位上玩手机、聊天。当取餐器震动时，你再去取餐。

在程序中：

- “点餐”就是发起一个网络请求。
- “取餐器”就是一种通知机制。
- “玩手机”就是程序在等待期间可以去执行的其他代码。

简单来说，它们是“目标”与“手段”的关系。

- 并发是我们的目标：
  - 我们希望程序能同时处理多个任务，提高效率。
- 异步编程是达成目标的手段之一：
  - 它是我们编写并发程序的一种具体方法。

好比我们的目标是“同时做好一顿饭（炒菜、煮饭、煲汤）”。

- 多线程方案：
  - 请三个厨师，每人负责一项。速度快，但“雇佣成本”高，且需要协调。
- 异步方案：
  - 一位经验丰富的厨师，他先按下电饭煲和汤煲的开关（发起异步操作），然后在等待的间隙去炒菜。通过合理安排，一个人高效地完成了所有事情。

为什么不选择多线程方案

尽管多线程也能实现并发，但在网络编程中，它常常带来一些挑战：

- 数据竞争与复杂性：
  - 当多个线程同时尝试修改同一份数据时，很容易出现混乱，导致程序出错。
  - 就像多个人同时在同一块白板上写字，最终结果可能一团糟。
  - 为了避免这种混乱，程序员需要使用复杂的“锁”机制来协调，这非常容易出错，并且难以调试。
- 特定语言的限制：
  - 像 Python 这样的语言，由于其内部机制（全局解释器锁 GIL），即使使用多线程，也无法真正同时运行多个计算任务，这限制了其在 CPU 密集型场景的性能。
  - 即使是 C/C++这类语言，虽然没有 GIL，但手动管理线程间的同步和数据安全，也是一个巨大的挑战。

为什么不选择多线程方案

- 逻辑直观性：
  - 即使是像 Rust 这样能保证线程安全的语言，多线程的逻辑也可能不如异步编程模型（特别是 async/await）那样直观和易于理解，尤其是在处理大量网络 I/O 时。

因此，在许多网络设计场景中，异步编程往往是更简洁、高效且易于维护的选择。

通过有向无环图(DAG)实现通信批次化

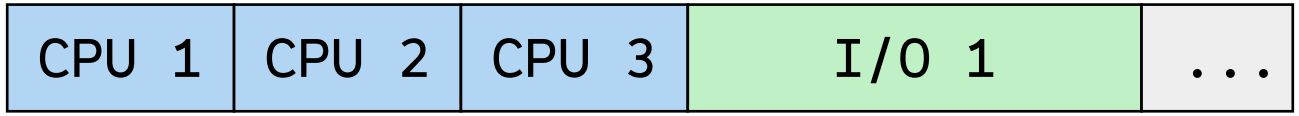
我们可以将数据依赖转换成图问题，进而使用拓扑排序来批次化无依赖关系的数据。
例如刚刚的循环发送。我们可以创建一个缓冲区，将数据先填充至缓冲区，然后在一轮循环结束的时候，统一发送。
接收方同理，创建缓冲区，通过循环解包出对应的数据。

手动批次化示例

Python

```python
# 手动批次化发送方代码
# items = [item1, item2, item3]
# for item in items:
#     socket_to_peer.send(item)

buffer = []
for item in items:
    buffer.append(item)
socket_to_peer.send(buffer)
```

批量化流程图表示



批次化的计算与传输

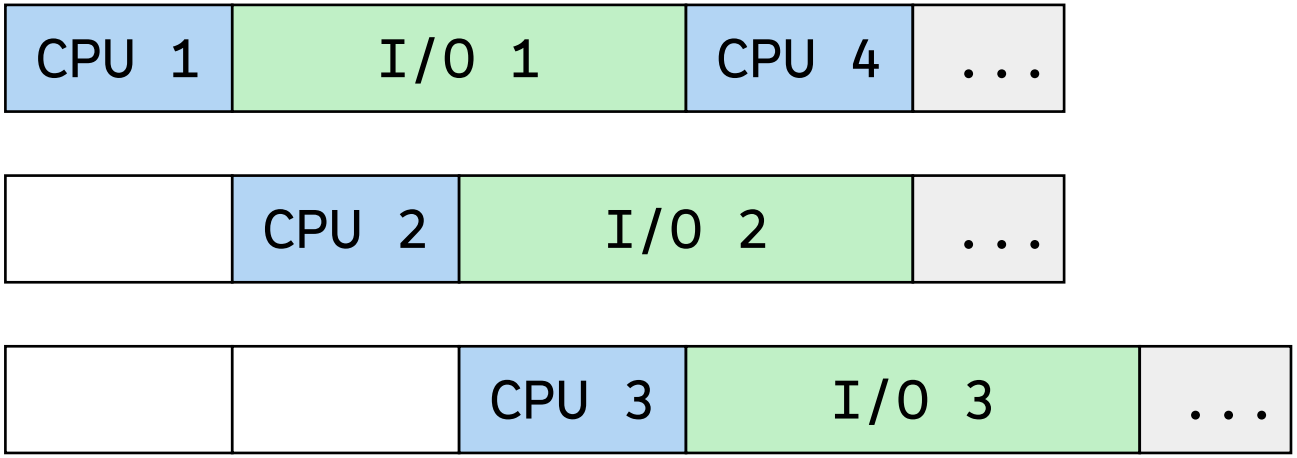
然后是并发化，如果多个步骤可以并行进行，那么拓扑排序也可以将这些步骤一起发送处理。
例如刚刚的发送给多个参与方的模式。我们便可使用一次并发，在等待确认的时候，发送其他数据。

并发化示例

Python

```python
# peers = [socket_to_alice, socket_to_bob, socket_to_carol]

# 创建一组并发的发送任务
tasks = [
    socket_to_peer.send(data) for peer in peers
]

# 同时执行所有发送任务
await asyncio.gather(*tasks)
```

并发流程图表示



异步化的计算与传输

主流的动态 DAG 框架是阿里的隐语框架 [1]。

此方案的核心思想是：

- 运行时构建：计算图（DAG）不是预先生成的，而是在程序执行过程中动态构建。这允许更灵活的编程，例如，计算的流程可以根据秘密计算的中间结果发生改变。
- 通用语言：开发者使用 Python 来编写 MPC 逻辑。
- 分布式框架：底层依赖一个强大的分布式执行框架（Ray [2]）来调度和执行图中的计算任务。

优点：

- 灵活性高，易于上手，可以处理具有复杂、数据依赖控制流的算法。
- 中文社区，有问题回复较及时。

缺点：

- 运行时动态调度和框架自身的开销较大(如传输的是整个 Python 对象)，性能不如静态方案。
- 框架组件多，使用起来复杂度高。

主流的静态 DAG 框架是 MP-SPDZ [3]。其工作流程完全不同：

- 编译时构建：开发者使用一种专门为 MPC 设计的领域特定语言（DSL）编写协议。
- 提前优化：编译器将 DSL 代码转换成一个固定的、静态的计算图，并进行大量优化（如指令调度、通信批处理）。最终生成高效的字节码。
- 虚拟机解释：项目使用 C++实现了一个虚拟机，在运行时加载并执行这些预先编译好的字节码。由于所有依赖关系都已确定，执行过程非常高效。

优点：

- 性能极高。由于在编译时就掌握了全部计算信息，可以进行全局优化，运行时开销非常小。

缺点：

- 灵活性差。计算流程必须在编译前完全确定，很难实现依赖于秘密数据的动态分支。
- 需要学习相应的 DSL。

选择快速的通信协议

传统方案是使用同步 TCP 来实现。但是这个方案因为 TCP 的握手和同步阻塞问题，性能上比较差。
对此，我们有多种优化方案。

非常简单的一个方式就是使用 TCP 长连接。这个方案需要额外维护一个连接列表。可以解决频繁建立 TCP 连接的握手开销。

核心思路是实现一个 ConnectionManager，它内部持有一个 HashMap 作为连接池。
当需要通信时，我们向管理器请求一个连接。

- 如果连接已存在，管理器直接返回它。
- 如果不存在，管理器负责建立新连接，存入池中，然后返回。

这样，无论上层逻辑是发送还是接收，都无需关心连接是否已经建立。

Rust

```rust
use std::collections::HashMap;
use std::net::{TcpStream, ToSocketAddrs};
use std::io::{self, Write};

struct ConnectionManager {
    connections: HashMap<String, TcpStream>,
}
```

```
8
9   impl ConnectionManager {
10      // 获取或建立一个新连接
11      // 这个函数体现了“维护链接”：调用者只管要连接，不用管是否已存在
12      fn get_connection<A: ToSocketAddrs + ToString>(
13          &mut self, addr: A
14      ) -> io::Result<&mut TcpStream> {
15          let addr_string = addr.to_string();
16
17          // .entry().or_insert_with() 是更地道的写法，这里为了清晰而展开
18          if !self.connections.contains_key(&addr_string) {
19              let stream = TcpStream::connect(addr)?;
20              self.connections.insert(addr_string.clone(), stream);
```

```
21              }
22
23              Ok(self.connections.get_mut(&addr_string).unwrap())
24          }
25      }
```

序列化：数据的高效编码

序列化是将内存中的数据结构（如对象、结构体）转换为可以存储或传输的格式（如字节流）的过程。在 MPC 中，我们需要在网络间传输大量数据，因此序列化的效率至关重要。

- 文本格式（如 JSON）：
  - 优点：人类可读，易于调试。
  - 缺点：体积庞大，解析速度慢，对于性能敏感的 MPC 通信是巨大的瓶颈。
- 二进制格式（如 Protobuf, Bincode）：
  - 优点：极其紧凑，解析速度飞快，是高性能场景的首选。
  - 缺点：人类不可读。

在 Rust 生态中，`serde` 框架配合 `bincode` 库是实现高效二进制序列化的黄金搭档。

序列化代码示例（Rust）

通过 serde，我们只需在结构体上添加一个派生宏，就能轻松实现序列化和反序列化。

Rust

```rust
use serde::{Serialize, Deserialize};

// 1. 使用serde的宏来自动实现序列化/反序列化
#[derive(Serialize, Deserialize, PartialEq, Debug)]
struct MyData {
    id: u32,
    payload: String,
}

fn main() {
    let original = MyData {
```

序列化代码示例（Rust）


```
12          id: 101,
13          payload: "hello".to_string(),
14      };
15
16      // 2. 使用bincode将结构体序列化为字节
17      let serialized_bytes: Vec<u8> =
        bincode::serialize(&original).unwrap();
18      println!("Serialized: {:?}", &serialized_bytes);
19
20      // 3. 从字节反序列化回结构体
21      let deserialized: MyData =
        bincode::deserialize(&serialized_bytes).unwrap();
22      println!("Deserialized: {:?}", &deserialized);
23
```

序列化代码示例（Rust）

```
24      assert_eq!(original, deserialized);
25  }
```

在序列化之后，我们可以通过一些压缩算法来降低传输的数据量，从而减少网络传输时间和带宽消耗。流式压缩特别适用于 MPC 中需要传输大量中间计算结果的场景。

优点：

- 减少数据量： 直接降低网络传输的数据大小，加快传输速度。
- 降低带宽需求： 对于带宽受限的环境尤其重要。
- 实时性： 流式压缩/解压缩可以在数据传输的同时进行，不会引入额外的显著延迟。

常用算法：

- Zlib/Deflate： 广泛使用的通用压缩算法，兼顾压缩比和速度。
- Snappy： Google 开发，以极快的压缩和解压缩速度著称，但压缩比略低于 Zlib，适用于对速度要求更高的场景。
- LZ4： 另一种非常快速的无损压缩算法，解压缩速度尤其快。

在选择压缩算法时，需要根据具体的 MPC 协议和网络环境，权衡压缩比、压缩/解压缩速度以及 CPU 开销。

gRPC 是由 Google 开发的一个现代、高性能的 RPC（远程过程调用）框架，它能很好地解决我们之前提到的一些通信瓶颈。

- 基于 HTTP/2：
  - gRPC 使用 HTTP/2 作为其传输协议，天然支持多路复用。这意味着可以在单个 TCP 连接上同时处理多个请求和响应，彻底解决了队头阻塞问题，也实现了长连接复用。
- 高效的 Protobuf：
  - 默认使用 Protocol Buffers (Protobuf) 进行序列化。相比于 JSON，Protobuf 是二进制格式，体积更小、解析更快。
- 支持流式通信：
  - 除了常规的“请求-响应”模式，gRPC 还原生支持客户端流、服务端流和双向流。这对于需要连续交换大量数据的 MPC 场景非常有用。

QUIC 是一个更前沿的传输层协议，它被用作 HTTP/3 的基础。它的设计目标是彻底解决 TCP 的固有顽疾。

- 构建于 UDP 之上：
  - QUIC 抛弃了 TCP，选择在更底层的 UDP 上重新实现了可靠传输、拥塞控制等功能。
- 解决了真正的队头阻塞：
  - HTTP/2 在单个 TCP 连接上多路复用，但如果一个 TCP 数据包丢失，整个连接上的所有流都必须等待它重传。这是传输层的队头阻塞。
  - QUIC 将“流”作为一等公民。每个流的数据包被独立处理，一个流的丢包不会阻塞其他流。

- 更快的连接建立：
  - QUIC 将传输层的握手（类似 TCP 三次握手）和加密握手（TLS）合并了。对于已有连接，它甚至可以实现 0-RTT（零往返时间）的连接恢复，速度极快。

为了更好地理解不同通信协议的优劣，我们来对比一下它们在 MPC 场景下的表现。

特性	传统 TCP/HTTP/1.1	gRPC（HTTP/2 over TCP）	QUIC（HTTP/3 over UDP）
传输层	TCP	TCP	UDP
队头阻塞	应用层和传输层	传输层（TCP HOLB）	无（流独立）
连接建立	慢（TCP 3 次握手 + TLS）	慢（TCP 3 次握手 + TLS）	快（0-RTT/1-RTT）
多路复用	无	有（应用层）	有（传输层）
序列化	通常文本（如 JSON）	Protobuf（二进制）	协议无关（二进制）
性能	较低	中高	高
适用场景	简单请求/响应	微服务/RPC	实时通信/高并发

使用异步的发送可以让我们无须等待 I/O。也就是说，我们在调用 `send()/recv()`的时候，线程不会一直阻塞，而是会将 CPU 时间片交给其他任务。

异步编程有多种模型，最主流的是 `async/await` 方案。

优点：

- 代码直观： 代码的线性逻辑使其看起来像同步代码，非常易于读写和维护。
- 资源占用低： 基于无栈协程，单个任务内存开销极小，可以轻松创建海量并发，且上下文切换在用户态完成。
- 统一的错误处理： 可以使用语言内建的错误处理机制（如 Rust 的 `?`），避免了回调地狱中的错误处理难题。

缺点：

- 心智负担： 需要理解 `Future`、执行器等概念，并手动处理 CPU 密集型任务（放入线程池）。
- 函数染色： `async` 关键字具有传染性，`async` 函数不能被同步代码直接调用，反之亦然，割裂了生态。

Rust 中的异步实现对比

在 Rust 生态中，同时存在 Stackful 和 Stackless 两种异步实现，代表了不同的设计哲学。

Stackful（may 框架）：

- may 提供了类似 Go Goroutine 的体验，每个协程拥有自己的栈。
- 开发者可以像写普通同步代码一样进行编程，网络 IO 等操作会被框架自动调度，对用户透明。
- 同样是“无色”函数，不强制区分同步和异步函数。

Stackless（tokio 框架）：

- 这是 Rust 官方和社区的主流方案，基于 async/await 语法。
- 编译器将异步代码转化为状态机，内存占用极低。
- 必须遵循 async/await 的语法规则，存在“有色”函数问题，但换来了更高的性能和更精细的控制。

设计异步化的通信流程

我们目前学习的方案大部分是同步的协议，即协议的数据几乎要依赖于上一步的状态或结果。 Damgård [4] 提出一个全异步的协议设计概念，即协议的正确性不依赖与消息是否在规定时间内到达。 同时这篇文章提出 VIFF 框架（现已 archive，且继任为 MP-SPDZ）

近年来，还提出了一些新的方案。如 hbMPC[5] ，还有 Dumbo-MPC[6]。

总结

综合前面讨论的通信瓶颈、异步编程思想以及各种优化协议，我们可以勾勒出 MPC 中异步化通信流程的设计思路：

- 核心思想：非阻塞与并发
  - 充分利用异步编程模型，确保通信操作不会阻塞主计算线程，从而提高 CPU 利用率。
- 通信调度：DAG 驱动
  - 将通信任务抽象为有向无环图（DAG），通过拓扑排序实现通信的批处理和并发执行，最大限度地减少等待时间。
- 协议选择：高效可靠
  - 优先选用基于 HTTP/2（如 gRPC）或 QUIC（如 HTTP/3）的协议，利用其多路复用、快速握手和高效序列化等特性。
- 连接管理：长连接与复用
  - 维护连接池，实现长连接的复用，避免频繁建立和关闭连接的开销。

References

[1] J. Ma *et al.*, "SecretFlow-SPU: A Performant and User-Friendly Framework for Privacy-Preserving Machine Learning," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, Boston, MA: USENIX Association, July 2023, pp. 17–33. [Online]. Available: https://www.usenix.org/conference/atc23/presentation/ma

[2] P. Moritz *et al.*, "Ray: A Distributed Framework for Emerging AI Applications." [Online]. Available: https://arxiv.org/abs/1712.05889

[3] M. Keller, "MP-SPDZ: A Versatile Framework for Multi-Party Computation," in *Proceedings of the 2020 ACM SIGSAC*

Conference on Computer and Communications Security, 2020. doi: 10.1145/3372297.3417872.

[4] I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen, "Asynchronous Multiparty Computation: Theory and Implementation," in *Public Key Cryptography – PKC 2009*, S. Jarecki and G. Tsudik, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 160–179.

[5] D. Lu *et al.*, "HoneyBadgerMPC and AsynchroMix: Practical AsynchronousMPC and its Application to Anonymous Communication." [Online]. Available: https://eprint.iacr.org/2019/883

[6] Y. Su, Y. Lu, J. Li, Y. Wang, C. Dong, and Q. Tang, "Dumbo-MPC: Efficient Fully Asynchronous MPC with Optimal

Resilience.” [Online]. Available: https://eprint.iacr.org/2024/1705