分布式的未来

本文涉及到的技术 distributed futures 以 Ray 和 futures 为基础。关于 Ray，可以阅读其论文或查阅官网获得相关信息；关于 futures，可以阅读我之前的博客以及其参考资料。当然，如果你不了解它们也不会对阅读本文有很大影响。

今天分享的技术来自于 Stephanie Wang 的两篇论文：

- In Reference to RPC: It's Time to Add Distributed Memory
- Ownership: A Distributed Futures System for Fine-Grained Tasks

两篇文章基本上说的是同一回事，所以我只展现其最终状态。

RPC 的问题

不论我怎么理解，Wang 是这样引入她的工作的：传统的 RPC 会造成不必要的数据移动开销和阻塞问题。一般我们认为阻塞问题是编程层面的，不过这里我继续延续作者的思路：考虑一个如下的工作流程：

```
a = f()
b = f()
c = add(a, b)
```

现在我们已经有能力让具体的计算发生在不同的 worker 节点，而 driver 只是作为 coordinator。于是计算的具体过程可以表示成下图：

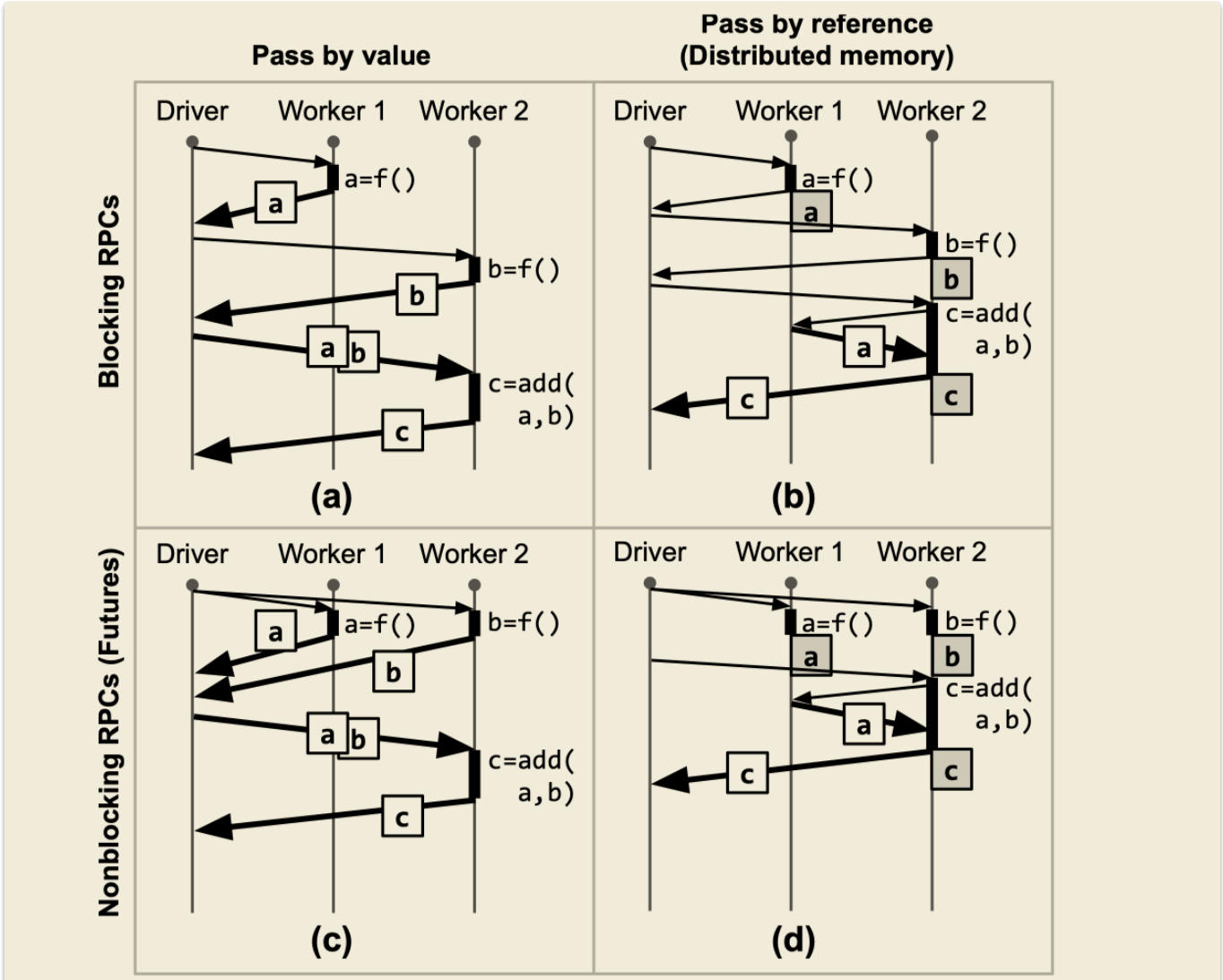

Figure 2: Example executions of the program from Figure 1. **(a)** With RPC. **(b)** With RPC and distributed memory, allowing the system to reduce data copies. **(c)** With RPC and futures, allowing the system to manage parallel execution. **(d)** With distributed futures.

这个图中的(a)表示传统的 RPC，它至少有两个可以优化的方向：

- 一共发生了5次数据移动，而其中至少有两次移动是没有必要的。在高性能计算场景或者说绝大多数分布式计算场景中（微服务除外），因为数据的体积比代码大很多，我们一般使用移动代码而非数据的方式来优化性能；
- 每次计算都是阻塞的，driver 必须等待上一个计算完成后才开始进行下一个计算。

简单优化

正如之前提到的，阻塞问题一般可以在客户端层面解决，即使用 futures 编程技术。这个方向可以达到简单的并行优化，结果就是(c)。

而数据移动问题稍微复杂一些。理想的情况当然是(b)，总共只发生了2次数据移动。使用引用传递或称为分布式内存就可以做到这一点。其思路和我们在日常编程中传递指针是一样的。当然，在分布式场景里面我们不可能真的传递指向本地内存的指针，一些扩展性的技术可以帮我们实现这一点，例如 RDMA，或者我们可以手动管理一个类似于 Redis 的分布式缓存，把它当做分布式内存，计算完成后将计算结果存放到 Redis 中，而向下游计算节点或 driver 传递它的 key。

但是，单纯使用分布式缓存有这么几个不方便的地方：

- 首先我们需要修改计算的逻辑，它们需要知道参数从哪里获得，结果存放到何处
- 我们还需要手动管理这些对象的垃圾回收
- 存取仍然是有延迟代价的。例如(b)中发生的一次数据传递，在分布式缓存中需要先存一次再取一次，直观上延迟代价至少翻倍了
- 尽管计算在各个节点发生，充分发挥了分布式系统的计算潜能，但对于数据密集型应用，最后的复杂可能就积压到了分布式缓存

后面我们会看到相较于分布式缓存，ownership 解决问题的方式更彻底。

最后一个图(d)就是 distributed futures 能达到的效果，既可以减少数据移动开销，也可以并行化任务。我们可以假想，要设计这样一个系统，至少需要做这么几件事：

- 负责整个系统的对象管理，包括对象的远程访问和回收
- 负责任务的调度，例如这个例子中系统怎么才能知道在哪个节点上计算 `f` 和 `add` 才能让总体开销最小
- 负责容错

关于动机，文章还说了一些融合计算的需求，简单来说就是当系统很庞大时，传统的各个组件通信的方式是通过 RPC，而 distributed futures 就可以用同样的思路让 RPC 变得非常高效。这也是 Ray 正在推行的概念，即让整个系统的各个组件都在 Ray 下编写、部署，最终就会使得系统的管理变得很容易，性能更高。但我认为这是不太可能的事情，原因你自己去思考，什么都跟你讲得明明白白你就不能成长。

接下来我们具体看看 distributed futures 怎么减少数据移动和并行化。

编程接口

首先是并行化，正如我之前说过的，使用 futures 就可以实现。Ownership 在实现上也提供了 asyncio 接口，但不用 asyncio 也可以，因为 Python 本来也支持多线程（你好，GIL）和阻塞 API。

接口是这样的：

Operation	Semantics
`f(DFut x) → DFut`	Invoke the remote procedure `f`, and pass `x` by reference. The system implicitly dereferences `x` to its `Value` before execution. Creates and returns a distributed future, whose value is returned by `f`.
`get(DFut x) → Value`	Dereference a distributed future. Blocks until the value is computed and local.
`del(DFut x)`	Delete a reference to a distributed future from the caller's scope. Must be called by the program.
`Actor.f(DFut x) → DFut`	Invoke a stateful remote procedure. `f` must execute on the actor referred to by `Actor`.
`shared(DFut x) → SharedDFut`	Returns a `SharedDFut` that can be used to pass `x` to another worker, without dereferencing the value.
`f(SharedDFut x) → DFut`	Passes `x` as a first-class `DFut`: The system dereferences `x` to the corresponding `DFut` instead of the `Value`.

Table 1: Distributed futures API. The full API also includes an actor creation call. A task may also return a `DFut` to its caller (nested `DFuts` are automatically flattened).

这和最早 Ray 在论文中提到的 API 有些差别，而事实上 Ray 现在也没有采用这里的 `shared` 和 `del`。另外，Actor 相关的事情我想放到后面再说，所以忘了这张图吧，看我的极简版描述：

```
remote : (f : a → b) → DFut a → DFut b
get : DFut a → a
```

`remote` 可以让一个函数变成远程函数，从而可以远程执行；而远程函数执行的参数和返回值都是引用，也就是 DFut 类型。在真正的实现中，如果传参是值，则会自动包装成 DFut。下面是一个可以直接运行的 Python 例子（注释可以暂时跳过）：

```python
import ray

ray.init() # 这一步在最新版的 ray 中可以省略

@ray.remote # 这个装饰器相当于 f = ray.remote(f)，将 f 变成远程函数即 task
def f():
  return 10

@ray.remote
```

```
def add(a, b)
    return a + b # 参数不需要手动解引用

a = f.remote() # a 只是一个引用，当然我们知道引用指向值10
b = f.remote()
c = add.remote(a, b) # 传入两个引用作为参数

print(ray.get(c)) # 这一步会真正开始运算并阻塞，直到获得结果，打印出`20`
```

为了和普通的函数区分，Ray 特意要求必须以 `f.remote()` 的形式才能调用函数，否则直接写 `f()` 会报错。

这段代码最重要的特点在于第15行参数隐式解引用，这使得函数可以几乎无痛地转变为远程函数，从而轻松地利用分布式计算资源。13~15这三行代码构建了一个计算图，只有当最后需要结果的时候，才会真正开始执行整个计算图。这其实就是延迟求值。对于 Ray 的场景而言，延迟求值的最大好处就是有充足的信息和时间可以用来编排任务和优化计算图。

作为优化的一个例子，上述计算流程中，15行的 `add` 依赖于前两行的计算结果，而13行和14行则没有依赖关系。因此，让13行和14行的计算并行执行在两个节点（或者，一个多核的 worker）上，减少时间；`add` 的计算则可以复用前一个 `f` 的计算节点，减少一次数据移动。

为什么要延迟求值呢？其实，什么时候求值并不会影响整个程序的语义，函数调用返回的都是 DFut。延迟求值只是一种实现，这个地方实现成非延迟也是可以的。第13行代码执行后 `f` 的计算就在别的节点上异步地开始进行了，还是到第17行才由依赖图点火执行，对 driver 程序是透明的。Ray 的选择是正确的，因为求值过程发生得越晚，优化的空间就越大。举一个极端的例子：

```
x = f()
y = g()
ray.get(y)
```

最后的结果没有用到 `f` 的执行结果，所以第一行代码是可有可无的（Ray 的 task 是没用副作用的），而延迟求值策略就可以通过依赖图避免执行第一行代码造成的性能损失。

延迟求值有这么几个常见的劣势：难以调试和反复确认计算结果的性能损失。

难以调试是因为代码执行的时空和计算真实发生的时空不一样，没法以传统的方式打断点调试。这个问题只能通过扩展调试手段来解决了。

性能损失主要是指，延迟求值在使用每一个值的时候，需要先确认这个值是否已经求值了，如果求值了就直接使用，否则就点火它的求值过程。考虑到 Ray 程序的 task 仍然有一定粒度，相比起执行一个 task 的代码传输、任务图编排、数据传输来说，确认的时间可以忽略不计。

不可变性

或许你已经注意到了，Ray 不支持形如 `set(k, v)` 的状态修改操作。文章中提及了一个原因：为了保持和传统 pass-by-value 的 RPC 语义一致。这个原因是相当次要的，主要原因是下面四个：可变性和延迟求值冲突了，可变性导致数据竞争，可变性和 actor 语义冲突了，可变性和自动垃圾回收冲突了。

前面我们说，延迟求值只是一种求值策略，不会改变语义。但如果允许多个 task 同时访问某个对象，task 运行的时间就成了问题。

数据竞争比较好理解，允许多个正在执行的任务修改同一个对象，就意味着并发问题。不可变性直接避免了并发问题的发生。

由于 actor 和 distributed futures 本身关联不大，我特意没有提及。简单来说，Ray 支持有状态的 actor 计算而不仅仅是 task，而 actor 的消息传递模型要求消息内容不可变。

自动垃圾回收，也就是自动内存管理，在下面将会详细介绍。`set(k, v)` 形式的操作隐含地假设了一个全局存在的键值存储，只要不显式地删掉某个键，它就应该能被访问。因此在所有任务执行完成前，没有任何对象是可以回收的。

内存管理

Ownership 的对象存放在各个节点的 object store 上，运行 object store 的节点和计算节点是不做区分的。我仍然用这个例子来解释中间发生的内存管理过程：

```
a = f()
b = f()
c = add(a, b)
get(c)
```

Object store 提供了类似 kv 的操作接口，如下：

Operation	Semantics
`Create(ObjID o, Value v)`	Store an object.
`Pin(ObjID o, NodeID loc)` → `bool`	Pin `o` on `loc` until released. Returns false if `loc` failed.
`Release(ObjID o)`	Object `o` is safe to evict.
`Get(ObjID o)` → `Value`	Get the object value. May fetch copy from remote node.

结合这些 API，我们完整地过一遍：

1. 前三行代码构建计算图，第四行代码开始，对 `c` 发起 `get` 操作，这一操作会被转换为上图中的 `Get(c)`
2. `Get(c)` 会找到一个空值，该空值和一个 task 关联，从而触发 `c = add(a, b)` 这个 task 的执行，假设这个 task 被调度到 N0 执行
3. 同样的方式，`Get(a)` 和 `Get(b)` 会触发 `a = f()` 和 `b = f()` 这两个 tasks 的执行
4. 执行 `a = f()` 这一 task 的节点 N1 会在自己的 ownership table 中保存一个 ObjectID 为 `a` 的映射，内容包含其值（现在是空）、其 owner N1、其 reference N0 和 N1
5. 当这个 task 执行完毕后，N1 调用 `Create(a, 10)`，将表中的空值部分填充为10（当数据量很大时，这里填充一个指向 object store 的指针）
6. 随后，如果这个值很大，需要存储道 object store 中，N1 会调用 `Pin(a, N1)`，指将这个值钉在这个第一次产生它的节点，防止其移动。为了降低访问延迟，Ownership 会在必要的时候复制某些对象，而复制对象就只会调用 `Create`，不会调用 `Pin`，只有当任务第一次产生对象时才会调用 `Pin`。当内存不够用时，那些没有被钉住的对象就会被释放，而钉住的都是最初始的副本，不会被释放，否则会造成对象丢失
7. 接下来，由于 `a = f()` 这一 task 完全执行完了，所以在 ownership table 的 `a` 这一行中把 N1 从 references 列表删除。现在 reference 只剩 N0
8. `Get(a)` 返回，10作为值被传送到 N0 节点上。大概就在差不多的时间，`Get(b)` 也返回了，此时 N0 同时有了 `a` 和 `b` 的值
9. `c = add(a, b)` 发生，ownership table 的创建、`Create` 和 `Pin` 的调用也和上述过程如出一辙
10. 最后，由于这个 task 执行完了，N0 不再 refer to `a` 和 `b`，所以两个 table 中 N0 会从 references 列表删除
11. 当一个对象的 references 的列表为空时，调用 `Release(a)`，这个 table 的记录被删除，object store 的相应内存被释放

12. 最终 `Get(c)` 返回，同样的释放逻辑再执行一遍

这里我还忽略了一些细节，比如这个表单的完整内容：

Field	Value
`*ID`	The `ObjectID`. Also used as a distributed memory key.
`*Owner`	Address of the owner (IP address, port, `WorkerID`).
`*Value`	(1) Empty if not yet computed, (2) Pointer if in distributed memory, or (3) Inlined value, for small objects (Section 4.2).
`*References`	A list of reference holders: Number of dependent tasks and a list of borrower addresses (Section 4.2 and appendix A).
`Task`	Specification for the creating task. Includes the `ObjectIDs` and `Owners` of any `DFuts` passed as arguments.
`Locations`	If `Value` is empty, the location of the task. If `Value` is a pointer to distributed memory, then the locations of the object.

Table 2: Ownership table. The owner stores all fields. A borrower (Section 3.2) only stores fields indicated by the *.

只有 owner 节点才会持有表单的完整条目，而其它的对象借用者只持有前四项。

整体上看，Ownership 就是一个分布式的引用计数。说到引用计数，你可能立刻会想到环状引用问题。但不用担心，Ray 的对象都是不可变的，意味着你不可能制造出环形的数据结构，所有的对象一定是树状组织的。

后续有空再讲容错。

（本文定价1元）