4.8 并行计算

::: details INFO
译者：sumingfirst

来源：4.8 Parallel Computing

对应：无
:::

从 1970 年代到 2000 年代中期，单个处理器内核的速度呈指数级增长。这种速度的提高大部分是通过增加时钟频率（处理器执行基本操作的速率）来实现的。然而，在 2000 年代中期，由于功率和发热限制，这种指数增长突然结束，从那时起，单个处理器内核的速度增长要慢得多。相反，CPU 制造商开始在单个处理器中放置多个核心，从而可以同时执行更多操作。

并行性并不是一个新概念。大型并行机已经使用了几十年，主要用于科学计算和数据分析。即使在具有单个处理器内核的个人计算机中，操作系统和解释器也提供了并发的抽象。这是通过上下文切换或在不同任务之间快速切换来完成的，而无需等待它们完成。因此，即使一台机器上只有一个处理核心，多个程序可以在其同时运行。

鉴于当前处理器内核数量增加的趋势，为了更快的运行，各个应用程序现在必须利用并行性。在单个程序内部，必须安排计算，使尽可能多的工作可以并行完成。但是，并行性在编写正确代码方面引入了新的挑战，尤其是在存在共享的可变状态的情况下。

对于可以在函数模型中有效解决的问题，没有共享的可变状态，并行性几乎没有问题。纯函数提供引用透明性，这意味着表达式可以替换为其值，反之亦然，而不会影响程序的行为。这样就可以并行计算不相互依赖的表达式。正如前一节所讨论的，MapReduce 框架允许使用最小的编程工作量来指定并行运行的功能性程序。

不幸的是，并非所有问题都可以使用函数式编程有效地解决。伯克利视图项目已经确定了科学和工程中的 十三种常见计算模式，其中只有一种是 MapReduce。其余模式需要共享状态。

在本节的其余部分，我们将看到可变共享状态如何将错误引入并行程序，以及许多防止此类错误的方法。我们将在两个应用程序的上下文中研究这些技术，一个 网络爬虫 和一个 粒子模拟器。

4.8.1 Python 中的并行性

在我们深入研究并行性的细节之前，让我们先探讨一下 Python 对并行计算的支持。Python 提供了两种并行执行方式：线程和多进程。

线程：在 *线程* 中，单个解释器中存在多个执行“线程”。每个线程独立于其他线程执行代码，尽管它们共享相同的数据。然而，CPython 解释器是 Python 的主要实现，一次只解释一个线程中的代码，在它们之间切换以提供并行的错觉。另一方面，解释器外部的操作（例如写入文件或访问网络）可以并行运行。

该 `threading` 模块包含允许创建和同步线程的类。下面是一个多线程程序的简单示例：

```
>>> import threading
>>> def thread_hello():
        other = threading.Thread(target=thread_say_hello, args=())
        other.start()
        thread_say_hello()

>>> def thread_say_hello():
        print('hello from', threading.current_thread().name)

>>> thread_hello()
hello from Thread-1
hello from MainThread
```

`Thread` 构造函数创建一个新线程。它需要新线程运行的目标函数，以及该函数的参数。在 `Thread` 对象上调用 `start` 标志着它准备运行。该 `current_thread` 函数返回与当前执行线程关联的 `Thread` 对象。

在此示例中，打印可以按任何顺序进行，因为我们尚未以任何方式同步它们。

多进程：Python 还支持多进程，这允许程序生成多个解释器或进程，每个解释器或进程都可以独立运行代码。这些进程通常不共享数据，因此必须在进程之间通信任何共享状态。另一方面，进程根据底层操作系统和硬件提供的并行级别并行执行。因此，如果 CPU 有多个处理器核心，Python 进程可以真正并发运行。

该 `multiprocessing` 模块包含用于创建和同步进程的类。以下是使用进程的 hello 示例：

```
>>> import multiprocessing
>>> def process_hello():
        other = multiprocessing.Process(target=process_say_hello, args=())
        other.start()
        process_say_hello()

>>> def process_say_hello():
        print('hello from', multiprocessing.current_process().name)

>>> process_hello()
hello from MainProcess
>>> hello from Process-1
```

如本示例所示，`多进程` 中的许多类和函数类似于 `线程` 中的类和函数。此示例还演示了缺少同步如何影响共享状态，因为显示可以被视为共享状态。在这里，来自交互式进程的解释器的提示出现在另一个进程的打印输出之前。

4.8.2 共享状态的问题

为了进一步说明共享状态的问题，让我们看一个在两个线程之间共享的计数器的简单示例：

```
import threading
from time import sleep

counter = [0]

def increment():
    count = counter[0]
```

```
    sleep(0) # try to force a switch to the other thread
    counter[0] = count + 1

other = threading.Thread(target=increment, args=())
other.start()
increment()
print('count is now: ', counter[0])
```

在此程序中，两个线程尝试递增相同的计数器。CPython 解释器几乎可以随时在线程之间切换。只有最基本的操作是原子的，这意味着它们似乎是即时发生的，在评估或执行期间不可能切换。递增计数器需要多个基本操作：读取旧值、向其添加一个值和写入新值。解释器可以在这些操作中的任何一个之间切换线程。

为了显示当解释器在错误的时间切换线程时会发生什么，我们尝试通过休眠 0 秒来强制切换。运行此代码时，解释器通常会在调用时 `sleep` 切换线程。这可能会导致以下操作序列：

```
Thread 0                    Thread 1
read counter[0]: 0

                            read counter[0]: 0
calculate 0 + 1: 1
write 1 -> counter[0]

                            calculate 0 + 1: 1
                            write 1 -> counter[0]
```

最终结果是计数器的值为 1，即使它增加了两次！更糟糕的是，解释器可能很少在错误的时间切换，这使得调试变得困难。即使有调用 sleep ，此程序有时也会生成正确的计数 2，有时生成不正确的计数 1。

仅当存在共享数据时，才会出现此问题，共享数据可能被一个线程更改，而另一个线程访问它。这种冲突称为**争用条件**，它是仅存在于平行世界中的 bug 的示例。

为了避免争用条件，必须防止可能被多个线程更改和访问的共享数据。例如，如果我们可以确保线程 1 仅在线程 0 完成访问后访问计数器，反之亦然，我们可以保证计算出正确的结果。我们说共享数据如果受到并发访问保护，则共享数据是**同步**的。在接下来的几个小节中，我们将看到提供同步的多种机制。

4.8.3 不需要同步时

在某些情况下，如果并发访问不会导致不正确的行为，则不需要同步对共享数据的访问。最简单的示例是只读数据。由于此类数据永远不会发生突变，因此所有线程将始终读取相同的值，无论它们何时访问数据。

在极少数情况下，发生突变的共享数据可能不需要同步。但是，要了解何时适用这种情况，需要深入了解解释器以及底层软件和硬件的工作原理。请考虑以下示例：

```
items = []
flag = []

def consume():
    while not flag:
        pass
    print('items is', items)

def produce():
    consumer = threading.Thread(target=consume, args=())
    consumer.start()
```

```
    for i in range(10):
        items.append(i)
    flag.append('go')

produce()
```

在这里，生产者线程将 items 添加到 `items`，而消费者等到 `flag` 为非空。当生产者完成添加 items 时，它会向 `flag` 添加一个元素，允许消费者使用。

在大多数 Python 实现中，此示例将正常工作。但是，在其他编译器和解释器甚至硬件本身上，一个常见的优化是重新排序在单个线程内不依赖于彼此数据的操作。在这样的系统中，语句 `flag.Append ('go')` 可能会被移到循环之前，因为它们之间都不依赖于对方的数据。通常，除非确定底层系统不会对相关操作重新排序，应避免使用此类代码。

4.8.4 同步数据结构

同步共享数据的最简单方法是使用提供同步操作的数据结构。`queue` 模块包含一个 `Queue` 类，该类提供同步的先进先出（FIFO）访问数据。`Put` 方法将 items 添加到 `Queue`，`get` 方法检索 items。`Queue` 类本身确保这些方法同步，因此无论线程操作如何交错，items 都不会丢失。下面是一个生产者/消费者示例，它使用 `Queue`：

```
from queue import Queue

queue = Queue()

def synchronized_consume():
    while True:
        print('got an item:', queue.get())
        queue.task_done()

def synchronized_produce():
    consumer = threading.Thread(target=synchronized_consume, args=())
    consumer.daemon = True
    consumer.start()
    for i in range(10):
        queue.put(i)
    queue.join()

synchronized_produce()
```

除了使用 `Queue`、`get` 和 `put` 函数，此代码还进行了一些更改。我们已将消费者线程标记为 *守护进程*，这意味着程序退出之前不会等待该线程完成。这允许我们在消费者中使用无限循环。但是，我们确实需要确保主线程退出，但只有在从队列中消耗了所有 items 之后才退出。消费者调用 `task_done` 方法来通知 Queue 它已经完成了一个项目的处理，主线程调用 `join` 方法，直到所有项目都被处理完，确保程序只有在处理完之后才退出。

使用一个 `Queue` 更复杂的示例是 并行网络爬虫，它搜索网站上的死链接。这个爬虫跟踪同一站点托管的所有链接，因此它必须处理多个 URL，不断向 `Queue` 添加新的 URL 并删除要进行处理的 URL。通过使用同步 `Queue`，多个线程可以安全地同时添加和删除数据结构。

4.8.5 锁

当特定数据结构的同步版本不可用时，我们必须提供自己的同步。锁是执行此操作的基本机制。它最多可以由一个线程获取，之后没有其他线程可以获取它，直到它被先前获取它的线程释放。

在 Python 中，该 `threading` 模块包含一个提供锁定 `Lock` 的类。 Lock 具有 `acquire` 和 `release` 方法来获取和释放锁，并且该类保证一次只有一个线程可以获取它。当锁已经被持有时，所有其他试图获取锁的线程都被迫等待，直到锁被释放。

为了让锁保护特定的一组数据，所有的线程都需要遵循一个规则：除非拥有那个特定的锁，否则任何线程都不能访问共享数据。实际上，所有的线程都需要将他们对共享数据的操作 "包装" 在获取和释放锁的调用中。

在 并行网络爬虫 中，使用一个集合跟踪任何线程遇到的所有 URL，以避免多次处理特定 URL（并可能陷入一个循环）。但是，Python 不提供同步集合，因此我们必须使用锁来保护对普通集合的访问：

```
seen = set()
seen_lock = threading.Lock()

def already_seen(item):
    seen_lock.acquire()
    result = True
    if item not in seen:
        seen.add(item)
        result = False
    seen_lock.release()
    return result
```

这里需要一个锁，以防止在该线程检查该 URL 是否在集合中和将其添加到集合之间，另一个线程将其添加到集合中。此外，向集合添加不是原子的，因此并发地尝试向集合添加可能会破坏其内部数据。

在此代码中，我们必须小心，直到我们释放锁后才返回。通常，我们必须确保在不再需要时释放锁。这可能非常容易出错，尤其是在存在异常的情况下，因此 Python 提供了一个 with 复合语句来处理为我们获取和释放锁：

```
def already_seen(item):
    with seen_lock:
        if item not in seen:
            seen.add(item)
            return False
        return True
```

`with` 语句确保在执行其套件之前获取 `seen_lock`，并且在由于任何原因退出套件时释放它。(with 语句实际上可以用于锁定以外的操作，不过我们在这里不讨论其他用途。)

必须相互同步的操作必须使用相同的锁。但是，必须只与同一操作集中的操作同步的两个不相交的操作集，应该使用两个不同的锁对象，以避免过度同步。

4.8.6 障碍

在 Python 中，该 threading 模块以 `Barrier` 实例 `wait` 方法的形式提供了一个屏障：

```
counters = [0, 0]
barrier = threading.Barrier(2)
```

```
def count(thread_num, steps):
    for i in range(steps):
        other = counters[1 - thread_num]
        barrier.wait() # wait for reads to complete
        counters[thread_num] = other + 1
        barrier.wait() # wait for writes to complete

def threaded_count(steps):
    other = threading.Thread(target=count, args=(1, steps))
    other.start()
    count(0, steps)
    print('counters:', counters)

threaded_count(10)
```

在此示例中，读取和写入共享数据发生在不同的阶段，由障碍隔开。写入发生在同一阶段，但它们是不相交的; 这种隔离是必要的，以避免并发写入相同的数据在同一个阶段。由于此代码已正确同步，因此两个计数器在末尾将始终为 10。

多线程粒子模拟器 以类似的方式使用屏障来同步对共享数据的访问。在模拟中，每个线程都拥有许多粒子，所有这些粒子在许多离散的时间步长过程中相互作用。粒子具有位置、速度和加速度，并且根据其他粒子的位置在每个时间步长中计算新的加速度。粒子的速度必须相应地更新，其位置必须根据其速度进行更新。

与上面的简单示例一样，有一个读取阶段，其中所有粒子的位置都由所有线程读取。每个线程在此阶段更新其自身粒子的加速，但由于这些是不相交的写入，因此不需要同步它们。在写入阶段，每个线程都会更新其自身粒子的速度和位置。同样，这些是不相交的写入，它们通过屏障免受读取阶段的影响。

4.8.7 消息传递

最后一种避免共享数据不当变化的机制是完全避免对相同数据的并发访问。在 Python 中，使用多进程而不是线程自然会导致这种情况，因为进程在单独的解释器中运行，具有自己的数据。多个进程所需的任何状态都可以通过在进程之间传递消息来传达。

`多进程模块` 中的 `Pipe` 类提供了进程之间的通信通道。默认情况下，它是双工的，即双向通道，但传入参数 `False` 将导致单向通道。`Send` 方法通过通道发送一个对象，而 `recv` 方法接收一个对象。后者是阻塞的，意味着调用 `recv` 的进程将等待，直到接收到对象。

以下是使用进程和管道的生产者/消费者示例：

```
def process_consume(in_pipe):
    while True:
        item = in_pipe.recv()
        if item is None:
            return
        print('got an item:', item)

def process_produce():
    pipe = multiprocessing.Pipe(False)
    consumer = multiprocessing.Process(target=process_consume, args=(pipe[0],))
    consumer.start()
    for i in range(10):
        pipe[1].send(i)
    pipe[1].send(None) # done signal
```

```
process_produce()
```

在此示例中，我们使用消息 `None` 来表示通信结束。我们还在创建消费者进程时，将管道的一端作为参数传递给目标函数。这是必要的，因为状态必须在进程之间显式共享。

粒子模拟器 的多进程版本使用管道在每个时间步中的进程之间传递粒子位置。事实上，它使用管道在进程之间建立整个循环管道，以最小化通信。每个过程都将自己的粒子位置注入其管道阶段，最终通过管道的完整旋转。在每次旋转的步骤中，过程都会将当前处于其自身管道阶段的位置的力施加到其自身的粒子上，因此在完全旋转后，所有力都已施加到其粒子上。

`Multiprocessing` 模块为进程提供了其他同步机制，包括同步队列、锁，以及从 Python 3.3 开始的 barrier。例如，可以使用锁或屏障将打印同步到屏幕，从而避免我们前面看到的不正确的显示输出。

4.8.8 同步陷阱

虽然同步方法对于保护共享状态很有效，但它们也可能被错误地使用，无法完成正确的同步、过度同步或导致程序由于死锁而挂起。

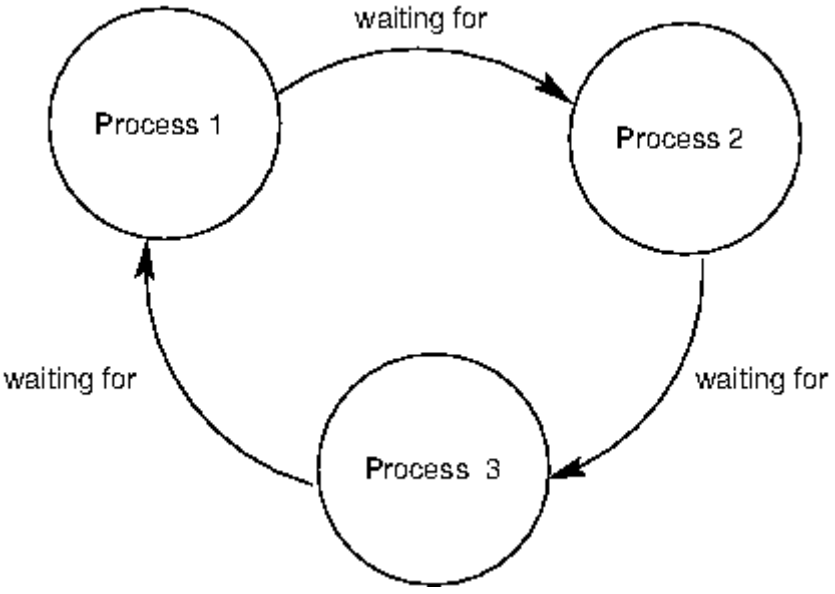
同步不足。并行计算的一个常见缺陷是忽略正确同步共享访问。在 set 示例中，我们需要将成员资格检查和插入同步在一起，以便另一个线程无法在这两个操作之间执行插入。即使这两个操作是单独同步的，但未能将两个操作同步在一起是错误的，。

过度同步。另一个常见错误是过度同步程序，使得不冲突的操作不能并发发生。举个简单的例子，我们可以通过在线程启动时获取主锁并仅在线程完成时释放它来避免对共享数据的所有冲突访问。这将序列化我们的整个代码，因此没有并行运行。在某些情况下，这甚至可能导致我们的程序无限期挂起。例如，考虑一个消费者/生产者程序，其中消费者获得锁并且从不释放它。这可以防止生产者生产任何物品，这反过来又阻止了消费者做任何事情，因为它没有什么可消费的东西。

虽然此示例微不足道，但在实践中，程序员经常在某种程度上过度同步他们的代码，从而阻止他们的代码充分利用可用的并行性。

死锁。由于同步机制会导致线程或进程相互等待，因此很容易出现死锁，即两个或多个线程或进程卡住，等待彼此完成的情况。我们刚刚看到了忽略释放锁如何导致线程无限期地卡住。但是，即使线程或进程正确释放了锁，程序仍可能达到死锁。

死锁的来源是循环等待，下面用进程说明。任何进程都无法继续，因为它正在等待等待它完成的其他进程。



例如，我们将设置一个包含两个进程的死锁。假设它们共享一个双工管道，并尝试相互通信，如下所示：

```
def deadlock(in_pipe, out_pipe):
    item = in_pipe.recv()
    print('got an item:', item)
    out_pipe.send(item + 1)

def create_deadlock():
    pipe = multiprocessing.Pipe()
    other = multiprocessing.Process(target=deadlock, args=(pipe[0], pipe[1]))
    other.start()
    deadlock(pipe[1], pipe[0])

create_deadlock()
```

两个进程都尝试首先接收数据。回想一下，该方法会 `recv` 阻止，直到某个项可用。由于两个进程都没有发送任何内容，因此两个进程都将无限期地等待另一个进程向其发送数据，从而导致死锁。

必须正确对齐同步操作以避免死锁。这可能需要在接收之前发送管道，以相同的顺序获取多个锁，并确保所有线程在正确的时间到达正确的屏障。

4.8.9 结论

正如我们所看到的，并行性为编写正确和高效的代码带来了新的挑战。在可预见的未来，随着硬件级别并行性增加的趋势将持续下去，并行计算在应用程序编程中将变得越来越重要。有一个非常活跃的研究机构，使并行更容易和程序员更不容易出错。我们在这里的讨论仅作为对计算机科学这一关键领域的基本介绍。