

目 录

1	目标.....	1
2	假设.....	2
2.1	串行算法复杂度	2
2.2	并行实现正确性	3
2.3	大数场景分析	4
2.4	并行优化方案设计	5
3	方法.....	8
4	结果.....	9
5	结论.....	10

1 目标

- 分析串行算法复杂度
 - 时间复杂度
 - 空间复杂度
- 分析并行实现的正确性
 - 可行算法的描述分析
 - 工作分解
 - 选择编程模型
- 大数场景分析
- 并行优化方案设计

2 假设

2.1 串行算法复杂度

解决串行 akari 问题的方法是基于回溯法的。首先题目要求的是求 akari 的串行时间复杂度。

首先要明确的是，回溯法的时间复杂度研究比较复杂，对不同的 case，时间复杂度的分析也不尽相同。

在这里尤其是 akari 问题的串行时间复杂度分析，及其复杂。不像 n 皇后那样有确定的解空间（n 皇后解空间为 $n!$ ，时间复杂度则为 $O(n!)$ ），因此这个分析对每个例子都有或多或少的不同。在这里直接假设最多的解空间，假设每一个方块都是 2（2 周围的放置能有 6 中情况，最多），但是这样的话是不可能符合实际案例的。因为这样的话很可能使得这个 puzzle 没有解，因此时间复杂度计算确实特别复杂，在这里只是尽可能计算一个上限，并且是比较符合实际情况的上限。但是这种情况还没有覆盖后面填充空白方块的情况，因此计算出来的时间复杂度只能说是一个参考，及其不准确。

时间复杂度分析：

上文说了，直接计算最多情况的 akari puzzle。每个方块 2 都能分裂 6 个新任务。根据 <https://www.puzzle-light-up.com> 上面的实例来看， $n \times n$ 的棋盘上带数字的黑格平均个数在 n ，这样的话，那么解空间为 6^n ，因此时间复杂度粗略估计为 $O(6^n)$ 。

2.2 并行实现正确性

实验指导要求使用 `pthread` 来实现并行。大部分具体实现体现在实验报告中，而不是本思考题中。因此本文主要是比较串并行以及分析。

P t h r e a d

这种方法是比较常见的实现并行的方法，也是用的比较多的方法。但是在这里 `p t h r e a d` 的效果可能不是很好。

但是要使用 `pthread` 的话，需要考虑比较多的实现细节，而这些都在实验报告中呈现了，因此在此不赘述，只是粗略讲解。

第一如果要使用多线程的话，需要为每一个 `task` 复制一张当前棋盘，这样会及其浪费内存，也耗费了大量的时间。

第二需要确定并行的粒度，不同并行粒度获得的效果是不一样的，可能会很大程度上影响并行效率。这个将在后面的并行优化中看到效果。

2.3 大数场景分析

实验中大部分测试集都是 7×7 与 10×10 的，在这种情况下数据量比较小，这样的话有可能创建线程的开销占比巨大，造成整体的效率下降，这样的话，有可能最后程序达到的加速效果非常差。

测量 14×14 数据集的话，会花时间在计算上，计算占比逐渐增大，因此可能加速效果要比小数据集的效果要好。

除此之外，解决 **akari** 问题要分为两个步骤，第一个步骤是解决黑色方格四周的灯泡放置问题，其实从 7×7 变化到 14×14 ，需要处理的黑色方格只是增加了 7 个，但是解空间呈指数趋势上涨，因此计算的比例增大的很厉害。

第二个阶段就是处理剩余空格了，这个阶段需要将棋盘都用灯光充满，对于 14×14 的棋盘，可能填满了黑色方格周围的灯泡后，还需要有很多空格填充棋盘，因此在大数情况下，第二阶段的并行也非常重要。

2.4 并行优化方案设计

此处所说的并行优化方案主要指的是确定并行粒度。在实验报告的分析中可以看到对于三种不同的并行粒度，并行加速效果也不相同。

1. 每个解空间的节点分裂均产生新任务

这个方案就是为每一个孩子节点创建线程进行处理。如图 2-1

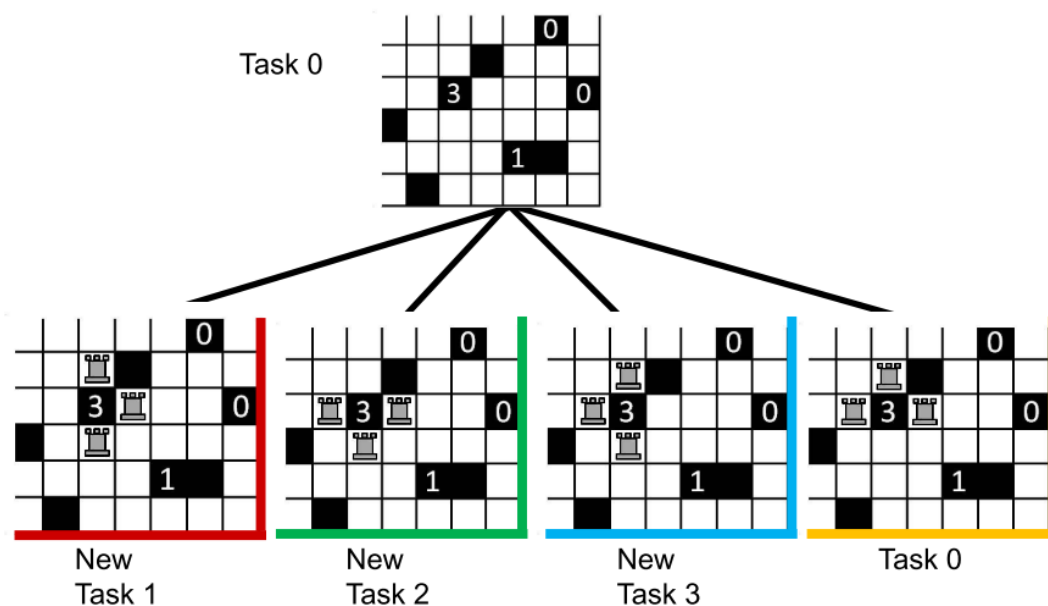


图 2-1 方案 1

这样做优点很明显，就是设计很简单，并且容易实现。每个 task 几乎都是一样的工作量，十分平衡。

但同样的，缺点也十分明显。第一个就是太多的 task 被创建了，这样使得创建开销巨大，降低效率。第二个就是每个 task 分配的任务量太少了，导致没有发挥足够的效率，不足以抵消创建的开销。第三个就是，每次创建新线程都得将当前的棋盘复制一遍，这样及其消耗内存。

2. 当且仅当解空间树节点第一次分裂的时候产生新任务

这种方案仅在第一次分裂时产生新任务，如图 2-2。

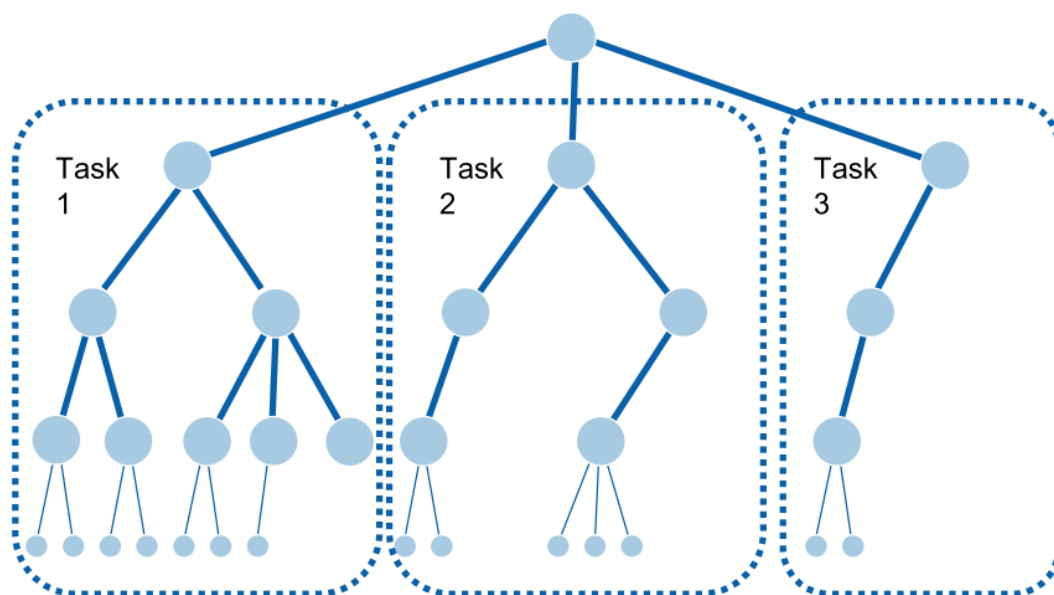


图 2-2 方案 2

这样做的优点就是创建 task 的时间开销小了，没有过大的开销限制效率。并且内存使用也比较小。

但是，这样很明显地，没有充分利用并行的优点。第一子树的变化太大，每个 task 的执行时间可能大不相同。也就是说每个 task 的负载太不相同了，严重降低了执行效率。第二它的扩展性较差，只在第一次分裂产生新任务，对于不同的情况，表现波动大，不是一种合适的方案。

3. 当且仅当解空间树节点在数字“3”的时候分裂时产生新任务

这种方案其实就是上面两种方案的折中。

如图 2-3：

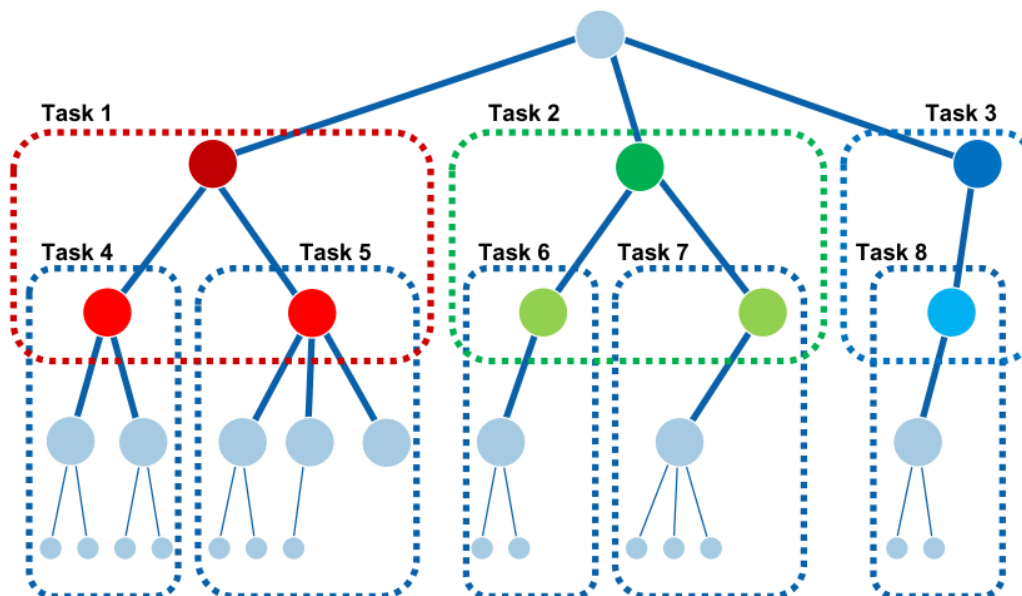


图 2-3 方案 3

这样做集合了 1 和 2 的优点，第一它产生任务的时间缩短了，并且每个 task 的负载也比 2 平衡的多。

但是缺点是实现比较困难，并且这种方案有较大的随机性，过于依赖树的特性。但是如果能在合适的时候分裂任务的话，应该能获得最大的效率。

但是如果仅当数字为 3 时才产生新任务的话，这样所实现的效果还不如每次都分裂产生新任务来的好。

最终经过粗略推导，方案 2 的效率肯定是最底的。那么剩下的就在方案 1 和方案 3 中间抉择了。对于不同的测试集，两种方案的效率可能会略有差别，但是总体上，应该是方案 1 是这三种中最好的。

在这里不对代码上的并行优化做过多的工作，主要研究粒度对并行效果的影响。

3 方法

本次主要就是计算加速比，因此首先就要得到原始串行算法的运行时间，因此首先测试串行算法所需时间。

然后对于 `p t h r e a d` 并行编程方法，使用同样的方法进行测试。

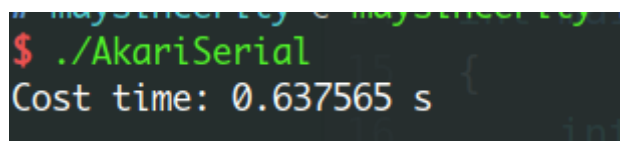
需要注意的是，这里的时间并不是所有的时间，在这里只考虑了解决 `akari` 问题所需要的时间，后面进行转换棋盘的过程，输出棋盘的过程不进入考虑。

这样，再计算不同粒度的并行方案所需要的时间，进行比较即可。

4 结果

1. 串行算法时间

在本机上测试 14×14 的数据集，也就是测试网站最后一个测试数据。测试多次，发现测试时间平均维持在 0.63s 左右



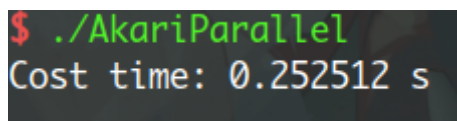
```
$ ./AkariSerial  
Cost time: 0.637565 s
```

图 4-1 串行时间

测试了 10 次，最后得出总时间为 6.48712s。

2. 并行算法时间

并行方案最终选择每次分裂都产生 task，这个方案从三个方案来看效果是最好的。同样地，测试相同的 14×14 的数据集，发现测试时间略有波动，不过多次测试平均时间维持在 0.25s 左右



```
$ ./AkariParallel  
Cost time: 0.252512 s
```

图 4-2 并行时间

测试 10 次，得到总时间为 2.43817。

经计算，加速比为 2.66。

5 结论

本次实验串行复杂度较高，在这种情况下需要考虑串行程序并行化来提高效率。

在大数场景下，并行程序要表现得更好一些，能够充分发挥出并行的效果，并且容易得到结论。

然后就是 3 种不同粒度的并行方案了，也就是在并行优化这一块，分别是全粒度，只在第一次产生新任务时，只在黑色方格为 3 的时候产生新任务。

这三种并行方案确实有较大的差距，综合实验结果来看，从这三种方案中方案 1 的效果最好，方案 2 的效果最差。其实从推论中也可以得出结论，详细分析见实验报告。

在下面就讨论 2 个思考题。

1. 除了实验中给出的串行程序并行化方法外，针对此 Akari 问题，你是否还有更好的并行程序设计方法。

除了实验中给出的三个并行方案，其实还有更佳方案。

那就是在黑色方格数字为 3, 2, 1 的时候创建新 task。首先 4, 0 不用创建线程，因为它们只有一种情况，并行没有必要，反而增加了创建线程时的开销。而在情况较多的黑色方格 3, 2, 1 分裂任务。这样的话，应该加速比能够达到一个更好的状态。后来自己测试了一下，发现加速比能够达到 20+。

这样的话，能够比实验中三种方案都好，并且加速比超出太多太多。

2. 使用阿姆达尔定律计算出的理想加速比与实际测得的加速比有一定的差距，这是由于开销造成的，请结合操作系统原理说明，在实验中产生的开销具体是什么？并提出减少开销的方法。

实验中产生的具体开销有线程的创建，内存同步开销，调度开销，上下文切换开销。

线程创建的开销是非常明显的，内存的开销就是指需要维持线程的栈，这需要内存空间，因此产生了开销。调度开销就是指占用了操作系统的资源来管理线

程，协调它们之间的关系，调度管理。而上下文切换的开销就是当 CPU 由线程 A 切换到线程 B 时产生的一系列操作，包括保存线程 A 的现场以及加载线程 B 的线程，这个代价是巨大的。

因此有时候，多线程并不能带来真正的效率，有可能不合理使用多线程还会使程序执行效率降低，执行时间边长。

而减少开销的方式就有几种。

第一是尽量合理安排线程的工作量，如果负载设置不合理，每一个线程都分配极少的负载，那么上下文切换就会异常频繁，导致 CPU 时间都用在这个上面，反而降低了整体效率。因此避免创建不必要的线程是很关键的。

第二就可以避免竞争锁，有时多线程竞争锁会引起上下文切换，这个可以使用一些策略来避免。比如将数字按照 hash 划分，不同线程可以处理不同数据。

其他的策略，经过上网查阅资料，还可以使用锁分离技术和 CAS 技术等等，总之需要在减少上下文切换这个重点上做一些工作，对于基本的改良方法，都侧重在减少不必要的线程，但是线程也不能过少，否则就像实验指导方案 2 中的那样，效率反而很差。