OSLab2-Life Game

写在前面
工具
    main.c 编写
    验证脚本
    案例生成
    可视化
并行计算
    方案
    线程池
    人为分配
    总结

写在前面

我们在计算加速比时，采用ubuntu的外部指令 `time` 。这里并不采用 `time()` 或者 `clock()` 的原因为：

- `time()` 的精度只有秒
- `clock()` 对于线程休眠的情况，没有计算在内

工具

main.c 编写

为了更好适应后续的测试，我需要重构main.c函数，使其能够接受以下的输入参数：

- `steps` ： 生命游戏运行的周期数
- `init environment` ： 生命游戏初始环境的文件路径
- `num threads` ： 使用的线程数，我们定义 `num threads=1` 时调用 `simulate_life_serial` 函数；否则调用 `simulate_life_parallel` 函数，并且创建 `num threads` 个线程用于并行计算
- `output file` ： 输出文件，默认是 `num threads=1` 时，输出在 `output/serial.txt` ；否则输出在 `output/parallel.txt`

验证脚本

我将编写一款脚本，用于自动执行并记录，验证输出结果，计算加速比的脚本。使用提示词：

> 我现在具有两个脚本：life-serial 和 life-parallel，我希望实现一个脚本，自动执行并搜集程序的执行时间，并计算两个程序的耗时比。
> 首先，程序的执行格式为： ./life-serial 100 input/23334m 需要输入一个steps 和 初始环境
> 其次，我们具有以下五种初始环境：23334m，make-a，o0075，o0045-gun，puf-qb-c3
> 接着，我希望你使用python程序编写脚本，能够实现以下功能：
> 1. 对于其中一个程序，需要遍历执行steps = 10, 50, 100, 500, 1000 五种情况，并且还得遍历执行五种初始环境，记录执行时间
> 2. 对于同一个steps，同一个初始环境，需要计算两个程序的执行时间比，并且转换为加速比
> 3. 最终生成一个markdown表格，用于存储：加速比，执行时间
> 4. 每一次测算，都要重复3次，取平均值

Gemini可以生成对应的脚本，见 `scripts.py`

案例生成

为了更好地测试，验证，我需要一款自动生成初始环境的代码。使用提示词：

> 请你按照下面的格式，生成一个生命游戏初始环境的生成代码，要求：
> 1. 列数固定为2000，行数分别为 100， 500， 1000， 1500， 2000，3000，4000，5000，6000，7000，8000， 9000，10000
> 2. 生成为无后缀格式，并且开头为 列数 行数
> 3. 命名为 列数_行数
> 4. 图中需要有 20%左右的 'o'，80%的 '.'
> 5. 图中的四个边缘都是 '.'
> 参考格式如下：<下面省略>

Chatgpt4o-mini可以生成对应的脚本，见 `generate.py`

可视化

为了更好地观察结果，还编写了`draw.py`，它负责读取`scripts.py`的执行输出文件，并可视化加速比-steps-rows-cols等关系，便于进一步观察和验证。

并行计算

方案

首先需要对任务进行划分，不失一般性，我对行进行划分，每一行都由唯一一个线程负责。

我设计了两种方案：

- 线程池。首先创建一个线程池，每个线程每次执行一行，执行完一行就去领取新的一行来执行。没有固定的任务分配
- 人为分配。人为分配任务给每一个线程，比如，分配上半区给线程一，分配下半区给线程二。最后只需要等待所有线程执行完毕

两种方案各有优劣，在后续将进行分析。

线程池

整体方案为：

- 创建线程池，每一个子线程都在等待任务的到来，领到任务之后就执行，执行完接着等待任务。否则休眠
- 主线程负责派发任务，并唤醒子线程执行，自己休眠，直到任务完成

使用线程池的优势在于：

- 动态分配任务，相比较人为分配，它允许早执行完的线程领取到更多的任务。从而避免受限于最慢线程；

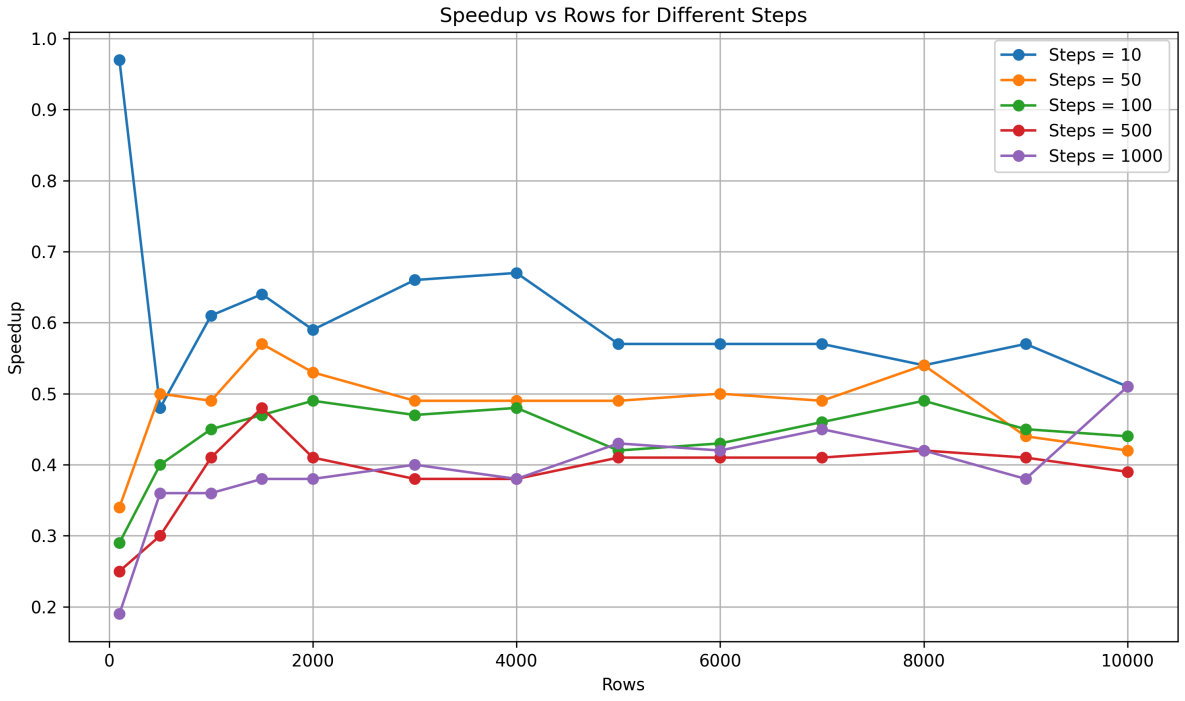
但是它的劣势有：

- 由于是动态分配任务，中间需要大量的互斥锁来保证执行逻辑，当线程数增加时，在同步等方面将会耗费大量的时间；
- 由于是按行分配任务，在列数比较小的时候，线程之间的执行时间差距不大，更易造成争抢互斥锁的情况，导致性能的大量降低；

总结就是，线程池在：

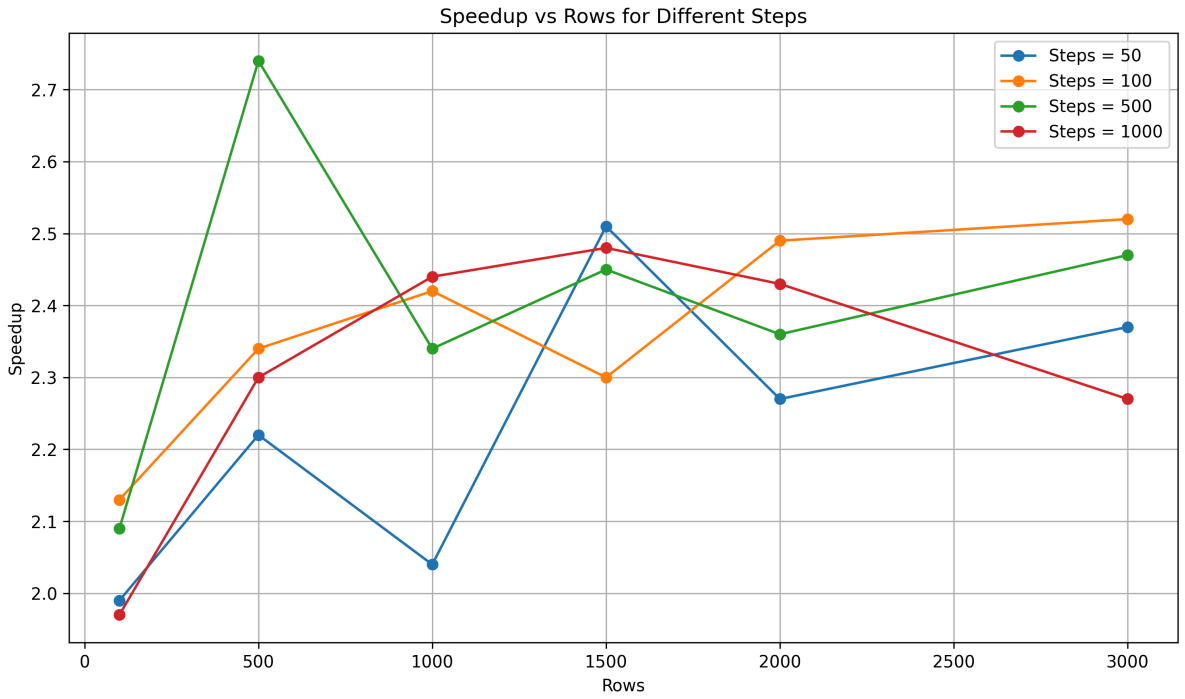
- 列数大，此时线程之间出现争抢的概率低
- 线程少，此时线程之间出现争抢的概率低

的情况下，比较有优势。



上图是使用线程池，4线程，列数固定为40的 加速比-行数 图，可以看到：

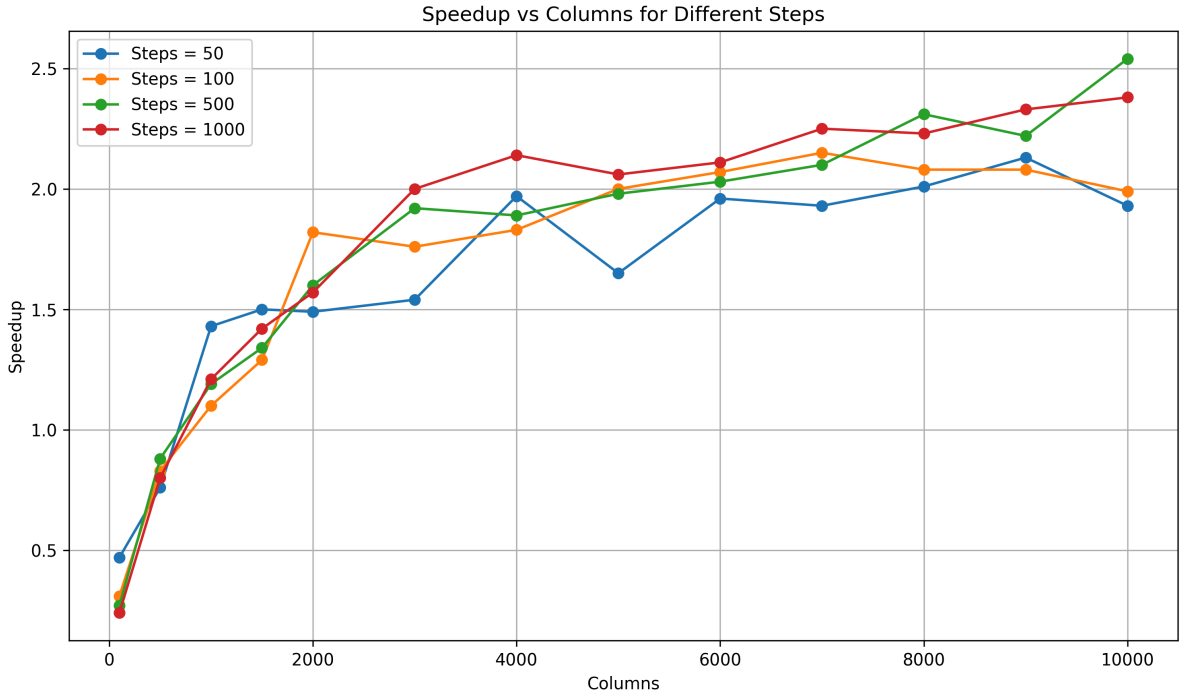
- 加速比基本和行数无关，这是符合直觉的。因为我们是根据行划分的
- 加速比甚至还不如串行，可见此时线程之间的争抢情况非常常见
- 加速比基本和steps无关，这是符合直觉的。因为我们并未对steps进行优化



上图是使用线程池，4线程，列数固定为2000的 加速比-行数 图，可以看到：

- 加速比更行数无关，且基本稳定在 2.5x ~ 2.6x

由于加速比和行数，steps的关系并不大，那么接下来我们就关注列数和加速比之间的关系：



上图是使用线程池，4线程，行数固定为40的 加速比-列数 图，可以看到：

- 加速比随着列数的增加而上升，并逐步收敛。符合直觉，因为列数比较大时，出现争抢的情况小

人为分配

整体的方案为：

- 对于每一个steps循环，使用 `交错分配` 的方式来分配任务，比如对于两个线程，将奇数行分配给线程一，将偶数行分配给线程二
- 主线程只负责创建线程，并分配任务，等待结束即可

整体的思路还是比较简单的，该方案具有以下的优点：

- 由于人为分配了任务，所以基本没有使用互斥锁等，避免了争抢所带来的时间损耗
- 在线程增加的情况，也能保持很好的robustness

缺点有：

- 整体的执行时间取决于最慢线程

总结起来，人为分配在

- 线程多，此时不需要调配

的情况下，执行的效果要比线程池好。但是在

- 少线程，列数大

的情况下，不如线程池，因为它会被最慢线程拖累

总结

我们需要结合两种方案的特点，线程池可以免受最慢线程的拖延，人为分配可以避免互斥锁的争抢。于是我们可以设计这样的策略：

- 对于线程数量比较多，计算规模比较小的情况，使用人为分配

- 对于线程数量比较少，计算规模比较大的情况，使用线程池
- 对于线程数量比较大，计算规模比较大的情况，使用 人为分配+线程池

我重点介绍一下最后一种策略，不失一般性，我们同样假设是使用行划分。

当计算 N 行，共 n 个线程，此时我们可以将 N 行划分成 p 组，每一组 N/p 行，且由 n/p 个线程负责。相当于人为分配了 p 组，每组内的工作量由线程池动态分配。

时间有限，暂时就开发到这里，最终上交的版本是线程池版本。