高级数据结构Lab3：并行编程

0. 背景

如何让CPU更快地执行指令一直是硬件工程师梦寐以求的目标。为此开发人员提出了多种技术以提升单核运行速度。然而随着计算任务越来越复杂，数据规模越来越大，单核运算的瓶颈逐渐显现，多核协作运算现已成为主流计算方式。但是在多核上编程并不是一件容易的事情。相比于线性执行的单核程序，多核并行的程序更难以正确执行且debug更为困难。如何让程序正确地在多核上运行并充分发挥多核并行带来的好处，对软件开发人员提出了更高的要求。

1. 算法并行化框架

在本次Lab中你将学习并实现一个算法并行化框架，它能将一个单线程程序并行化并使之能够正确地跑在多核机器上。

基本思路

CPU是用来做计算的，每个运行在核上的程序本质上都是一系列计算任务。通常来说，制约某个核向下执行计算的因素是数据依赖：即当前任务需要获得一定的输入数据才能执行计算。如果输入数据尚未准备好，那么只能等待或切换至其他任务。输入数据的生产者可能是另一个核，也可能是设备甚至用户，这里我们主要考虑数据生产者是另一个核的情况。算法并行的核心是设计一种合理的任务切分方式，将一个大的并行任务划分为多个互相无依赖的子任务使之同时在多个计算核心上执行。

本次Lab的并行计算框架包含两种抽象：

- scheduler：拥有对计算任务的全局视角，知道当前有哪些任务可以被计算，将可以被执行的任务分发给worker。
- worker：接受scheduler分配的任务，执行计算并在完成后通知scheduler。

为了便于实现，我们将scheduler与worker的数量关系设置为一对多。

框架设计

线程通信

在本次lab中我们使用基于shared memory的队列完成线程通信。它的基本要素包括：

- 专属于每个worker的工作队列。为了避免多个worker线程竞争同一个工作队列，我们为每个worker线程配置一个单独的工作队列。scheduler负责将计算任务提交到worker的工作队列中，worker随即从工作队列中取出任务执行计算。
- scheduler的完成队列。worker在完成计算后需要提交完成消息到scheduler的完成队列中以将计算结果报告scheduler，以便scheduler汇总计算结果以及寻找新的可计算任务。

轮询 vs 睡眠

当worker线程的工作队列为空时，worker线程可以选择死循环等待直到工作队列不为空；也可以选择先去睡觉，等待scheduler唤醒（思考一下：这两种方式各有什么优缺点？）。scheduler线程同理，如果完成队列为空，也可以选择轮询或者睡觉。 在本次lab中，我们选择睡眠-唤醒方式。

避免data race

为了防止多个线程同时读写相同数据结构，通常会为数据结构加锁以避免data race。具体到本次lab，线程在向队列（包括任务队列以及完成队列）中`push`以及`pop`元素时需要上锁（事实上并不是所有情况下都必须上锁，如果你比较好奇无锁实现可以参考文档的最后一章）。

整体流程

通过以上三点相信你对本次Lab中所描述的计算框架已经有了初步认识，现在让我们将前面讲过的内容串联起来，观察一下框架的整体工作流程。

1. scheduler找到当前可以计算的任务，按照一定的负载均衡策略从所有worker线程中选择若干个，将计算任务加入他们的工作队列并唤醒worker线程。如果scheduler发现当前已经不存在可以分发的任务则睡眠。
2. worker被唤醒后从自己的工作队列中pop出任务执行计算。
3. worker完成计算后将完成消息加入scheduler的完成队列，唤醒scheduler。如果worker发现自己的工作队列已空，说明当前已没有可执行的任务，继续睡觉。否则如果工作队列不空则回到步骤2。
4. scheduler被唤醒后从完成队列中取出完成消息，更新任务间的依赖关系，回到步骤1。

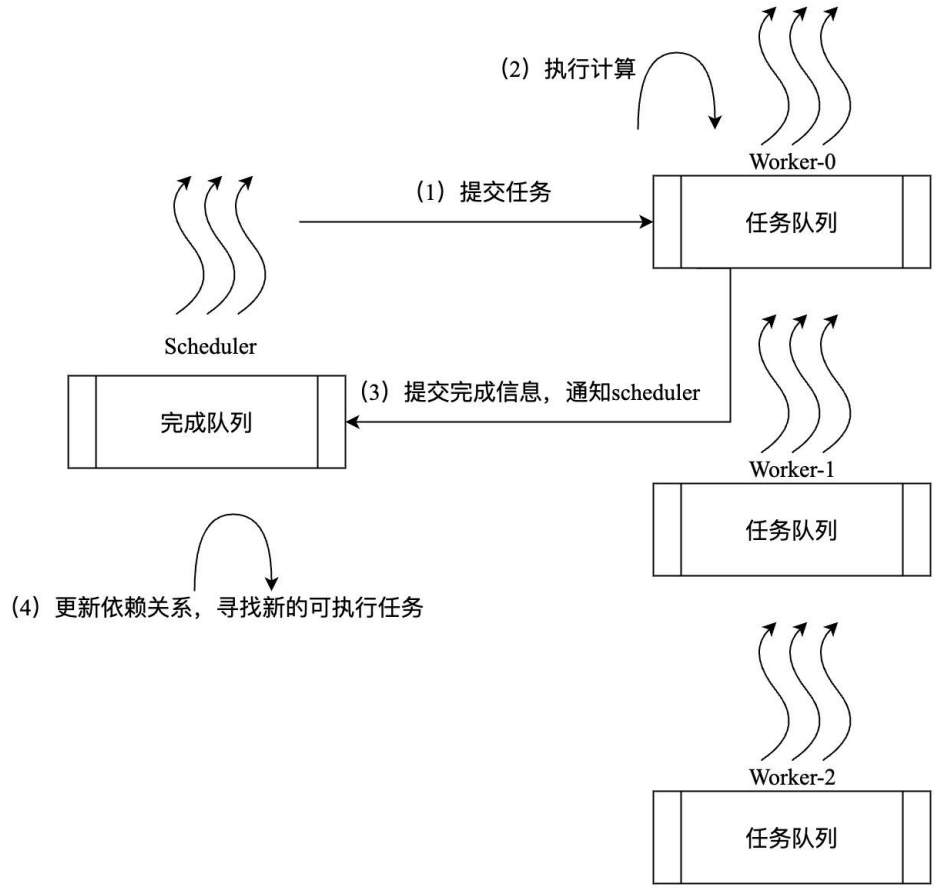

图 1: 并行计算框架整体架构

框架示例代码

下面的示例代码简要描述了上文所述的整体流程以及scheduler和worker的相互协作方式。变量含义如下：

- worker_mtx[]：每个worker工作队列的互斥锁，用于保证scheduler和worker在访问同一个工作队列时不会出现data race。
- worker_cond[]: 每个worker的条件变量，用于实现worker的睡眠与唤醒
- scheduler_mtx：scheduler完成队列的互斥锁
- scheduler_cond: scheduler的条件变量，用于实现scheduler的睡眠与唤醒
- wq：即worker的work queue
- cq：即scheduler的completion queue

- worker_list：保存所有worker线程

```
void worker_fn(int tid) {
    while (!task_all_done()) {
        // 1. 等待有任务到达或者任务全部完成
        unique_lock<mutex> w_lock(worker_mtx[tid]);
        worker_cond[tid].wait(w_lock, [&] {
            return wq[tid].size() || task_all_done();
        });

        if (wq[tid].empty()) {
            assert(task_all_done());
            break;
        }

        // 2. 从工作队列中提取任务并进行计算
        auto task = wq[tid].front(); wq[tid].pop();
        w_lock.unlock();
        int ret = task.run(task.args);

        // 3. 将计算结果放入scheduler的完成队列并唤醒scheduler
        unique_lock<mutex> s_lock(scheduler_mtx);
        cq.push(ret);
        scheduler_cond.notify_all();
    }
}
void scheduler_fn() {
    // 1. 创建THREAD_NR个worker threads
    for (int i = 0; i < THREAD_NR; i ++ ) {
        worker_list.push_back(move(thread(worker_fn, i)));
    }

    while (!task_all_done()) {
        // 2. 找到当前可计算的任务并向worker线程提交
        submit_task();

        // 3. 如果没有可计算的任务则睡眠
        unique_lock<mutex> s_lock(scheduler_mtx);
        scheduler_cond.wait(s_lock, [&]() {
            return has_computable_task();
        });

        // 4. 根据完成队列中worker thread的汇报结果，更新任务间依赖关系
        while (cq.size()) {
            int cur = cq.front();
            cq.pop();
            update_task(cur);
        }
    }

    // 5. 任务全部完成，等待worker thread退出
    for (int i = 0; i < THREAD_NR; i ++ ) {
        worker_cond[i].notify_all();
        worker_list[i].join();
    }
}
```

算法并行化示例问题

借助上述算法并行化框架可以将普通的单线程算法并行起来，下面是两个应用该框架的示例问题。

问题一 计算矩阵乘法

对于矩阵乘法$A * B = C$（注意问题一是普通的矩阵，不要和Lab0的稀疏矩阵乘法搞混），有：

$$C_{x,y} = \sum_{i=1}^{k} A_{x,i} * B_{i,y}$$

给定矩阵A和B，求解C。

问题二 按照递推式求解矩阵

假设我们有一个$n * n$的矩阵f。按照下面的递推式计算矩阵中每一项的值：

$$f_{0,0} = 1 \quad (1)$$

$$f_{0,j} = 0 \quad j \in [1, n] \ (2)$$

$$f_{i,0} = 0 \quad i \in [1, n] \ (3)$$

$$f_{i,j} = \left(f_{i-1,j} + f_{i-1,j-1}\right) \quad i, j \in [1, n] \ (4)$$

2. 实验要求及评分规则

考虑到大家本学期任务繁重，**本次 Lab 中的算法并行化框架的实现为可选项，不计入分数。Lab 的所有分数只根据提交的文档给出。**实现部分我们会照常提供评测接口，方便有兴趣的同学实现后验证程序正确性。

你需要阅读并理解上述算法并行化框架，在文档中回答以下问题：

1. 针对上面两个问题，分别说明应该如何进行任务划分，以通过算法并行化框架解决。只需说明方法，不要求代码实现。
2. 除了上述两个问题之外，算法并行化框架还可以用来解决什么问题？请给出一个问题，并描述如何应用框架解决，无需实现。
3. 虽然不要求大家实现上述的算法并行化框架，但为了让大家对并行化带来的性能提升有所感受，需要大家给出任意某个问题在并行化前后的性能差异（以运行时间或时钟周期）。具体要求包括：1) 请描述该问题；2) 描述并行化方法（不要求使用上述框架、任意并行化方法皆可）；3) 最后给出并行化前后的性能数据进行对比。此小问需要大家编写程序测试性能，但大家可以选择任何代码/问题（尽量使用自己之前写过的代码），只要能表现出并行化对性能的提升效果即可。

3. 扩展阅读: 并行编程常见的坑

提示：本章与实验并不强相关，如果时间紧迫可以放弃这部分内容的阅读。

在上面的框架中我们通过加锁避免data race的发生。但在有些情形下你可能会想要使用一些无锁的设计来避免lock与unlock的性能开销。这时你通常会发现，并行编程并不容易。下面我们会介绍两种常见的无锁并发编程模型，告诉你这些模型的问题在哪里以及如何解决。

自作聪明的编译器

实现线程同步通常有两种方式，一种是前面介绍的睡眠与唤醒，另一种被称作轮询等待。这种方式的思路是：如果线程A需要等待线程B执行某个操作后才能继续执行，那么可以设置A、B共享的flag：线程A轮询等待flag被置上，线程B在执行操作之后将flag置位，从而达到同步的目的。

```
int flag = 0;
// Thread-A
int func() {
    while (!flag);
    // do some thing
}

// Thread-B
{
  // some operation
  flag = true; // 通知A可以继续执行
  // continue to execute
}
```

然而在某些版本的编译器上，上述同步代码可能会造成死循环。这是由于编译器会假设其访问的内存不会被其他核修改，从而导致一些错误的优化。经过x86-gcc 7.1 开启O2后的Thread-A编译结果如下：

```
func:
        movl    flag(%rip), %eax
        testl   %eax, %eax
        jne     .L5
.L4:
        jmp     .L4
.L5:
        ...
```

这段指令序列的含义是：将flag的值读入eax，如果eax中的值不为0则跳转到.L5，否则跳转到.L4。由于.L4的跳转地址是自身，因此会在这里死循环。

你可能已经注意到：生成的可执行文件并没有真正地轮询flag这块内存，而是读取一次后将值写入寄存器，此后的flag都是从寄存器中读取的。这是由于编译器假设flag这个变量只会被当前核修改，那么读取一次已经足够，后续的load指令都被消除，从而导致线程A永远没有机会看到这个flag的更新，造成死循环。

volatile关键字可以提示编译器，不要优化内存读写操作，从而避免上述问题。修正后的代码如下：

```
volatile int flag = 0;
// Thread-A
int func() {
    while (!flag);
    // do some thing
}

// Thread-B
{
  // some operation
  flag = true;
  // continue to execute
}
```

不听话的CPU

在前面介绍的框架中，如果scheduler想要向worker的工作队列添加一个任务,需要在写队列之前上锁以避免data race。但由于队列的生产者（scheduler）和队列的消费者（worker）都只有一个，整个过程其实是可以无锁的。

下面的例子中，我们用task结构体描述一个任务。其中foo和bar用于描述任务信息，flag用来表示任务是否有效。worker的工作队列为一个task数组。

scheduler提交任务的过程为：

1. 调用`find_empty_task`，在工作队列中找到一个flag==false的空闲项
2. 填写任务的相关参数`foo`与`bar`
3. 将`flag`置为true，标记任务有效

worker消费任务的过程为：

1. 调用`find_empty_task`，在工作队列中找到一个flag==true的有效项
2. 提取foo与bar进行计算
3. 将flag置为false，标记该项空闲

```
struct task {
  int foo;
  int bar;
  bool flag;
};

// scheduler
{
  task *tsk = find_empty_task();
  tsk->foo = 0xb;
  tsk->bar = 0xa;
  tsk->flag = true;
}
// worker
{
  task *tsk = find_non_empty_task();
  int ret = calculate(tsk->foo, tsk->bar);
  tsk->flag = false;
  return ret;
}
```

由于scheduler只会写flag==false的数组项，worker只会写flag==true的数组项并且对于同一个工作队列，scheduler和worker都只有一个，理论上程序是没有问题的。

然而由于foo，bar与flag三个字段没有任何数据依赖，CPU在进行读写操作的时候有可能乱序执行。具体来说：虽然wcheduler先将foo和bar设为有效值，最后才将flag字段置为true。但实际的执行顺序可能是flag字段先被置为true，而此时foo和bar为任意值。当worker看到flag字段被置位，误以为任务已经有效，结果使用错误的foo/bar进行了计算。

为了避免上述问题，CPU提供了专用的读写屏障指令。例如：x86中提供了三种fence指令。其中sfence指令保证在该条指令前的store一定比在该条指令后的store更早被执行（即写屏障），lfence指令保证在该条指令前的load一定比在该条指令后的load更早被执行（即读屏障），mfence为读写屏障。

修正后的scheduler如下：通过添加写屏障，保证一定是先写foo/bar，然后才置位flag。

```
#define wmb()asm volatile("sfence" ::: "memory")
```

```
// scheduler
{
  tsk->foo = 0xb;
  tsk->bar = 0xa;
  wmb();
  tsk->flag = true;
}
```

修正后的worker如下：因为只有flag为true后的foo和bar才是确认有效的，这个版本通过添加读屏障保证一定是先读到flag为true后才读foo与bar。

```
#define rmb()asm volatile("lfence" ::: "memory")

// worker
{
  int foo, bar;
  bool ok = false;
  for (int i = 0; i < N && !ok; i ++ ) {
    auto flag = tasks[i]->flag;
    rmb();
    if (flag) {
      foo = tasks[i]->foo;
      bar = tasks[i]->bar;
      ok = true;
    }
  }
  if (!ok) sleep();
  int ret = calculate(foo, bar);
  tsk->flag = false;
  return ret;
}
```