[读书笔记]CSAPP：29[VB]线程级并行

视频地址：

2015CMU 15-213 CSAPP 深入理解计算机系统 课程视频含英文字幕（精校字幕视频见av31289365！！！） 哔哩哔哩 (゜-゜)つロ 干杯~-bilibiliwww.bilibili.com/video/BV1XW411A7fB?p=24

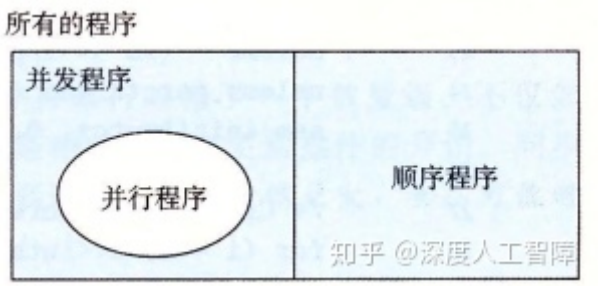
课件地址：

http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/lectures/26-parallelism.pdfwww.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/lectures/26-parallelism.pdf

本章对应于书中的12.6。

- 同步互斥锁的代价很大，要避免进行同步
- 并行程序一般一个核运行一个线程

根据程序逻辑流的数目，可以将程序分为顺序程序和并发程序，其中顺序程序只有一个逻辑流，而并发程序具有多个逻辑流。当在多个处理器中运行并发程序时，就成为了并行程序，速度会更快，因为内核在多个核上并行地调度这些并行线程，而不是在单个核上顺序地调度。



我们以一个简单的例子来看并行程序的例子，用并行程序来计算 $0, 1, \ldots, n-1, n$ 的和。首先，我们可以根据线程数目将这些数字划分成若干个组，每个线程在自己的组中计算结果，再将其放入一个共享全局变量中，需要用互斥锁保护这个变量。

code/conc/psum-mutex.c

```
1   #include "csapp.h"
2   #define MAXTHREADS 32
3
4   void *sum_mutex(void *vargp); /* Thread routine */
5
6   /* Global shared variables */
7   long gsum = 0;           /* Global sum */
8   long nelems_per_thread;  /* Number of elements to sum */
9   sem_t mutex;             /* Mutex to protect global sum */
10
11  int main(int argc, char **argv)
12  {
13      long i, nelems, log_nelems, nthreads, myid[MAXTHREADS];
14      pthread_t tid[MAXTHREADS];
15
16      /* Get input arguments */
17      if (argc != 3) {
18          printf("Usage: %s <nthreads> <log_nelems>\n", argv[0]);
19          exit(0);
20      }
21      nthreads = atoi(argv[1]);
22      log_nelems = atoi(argv[2]);
23      nelems = (1L << log_nelems);
24      nelems_per_thread = nelems / nthreads;
25      sem_init(&mutex, 0, 1);
26
27      /* Create peer threads and wait for them to finish */
28      for (i = 0; i < nthreads; i++) {
29          myid[i] = i;
30          Pthread_create(&tid[i], NULL, sum_mutex, &myid[i]);
31      }
32      for (i = 0; i < nthreads; i++)
33          Pthread_join(tid[i], NULL);
34
35      /* Check final answer */
36      if (gsum != (nelems * (nelems-1))/2)
37          printf("Error: result=%ld\n", gsum);
38
39      exit(0);
40  }
```


code/conc/psum-mutex.c

第30行中我们将每个线程的组号作为参数传递到线程例程中，再让主线程等待所有对等线程运行完毕，再与正确结果比较是否正确。

code/conc/psum-mutex.c

```
/* Thread routine for psum-mutex.c */
void *sum_mutex(void *vargp)
{
    long myid = *((long *)vargp);          /* Extract the thread ID */
    long start = myid * nelems_per_thread; /* Start element index */
    long end = start + nelems_per_thread;  /* End element index */
    long i;

    for (i = start; i < end; i++) {
        P(&mutex);
        gsum += i;
        V(&mutex);
    }
    return NULL;
}
```


code/conc/psum-mutex.c

在线程例程中，首先根据组号计算出线程需要计算的范围，然后对全局变量进行加和时，要注意用互斥锁将其保护起来。我们可以得到这个程序在四核系统上计算 2^{31} 的性能为

	线程数				
版本	1	2	4	8	16
psum-mutex	68	432	719	552	599

单位为秒

可以发现效果很差，其实**主要原因**是由于`P`和`V`对互斥锁的保护。我们可以将每个线程计算的结果保存在全局数组中的特定位置，这样就不用对数组进行保护

code/conc/psum-array.c

```
/* Thread routine for psum-array.c */
void *sum_array(void *vargp)
{
    long myid = *((long *)vargp);          /* Extract the thread ID */
    long start = myid * nelems_per_thread; /* Start element index */
    long end = start + nelems_per_thread;  /* End element index */
    long i;

    for (i = start; i < end; i++) {
        psum[myid] += i;
    }
    return NULL;
}
```

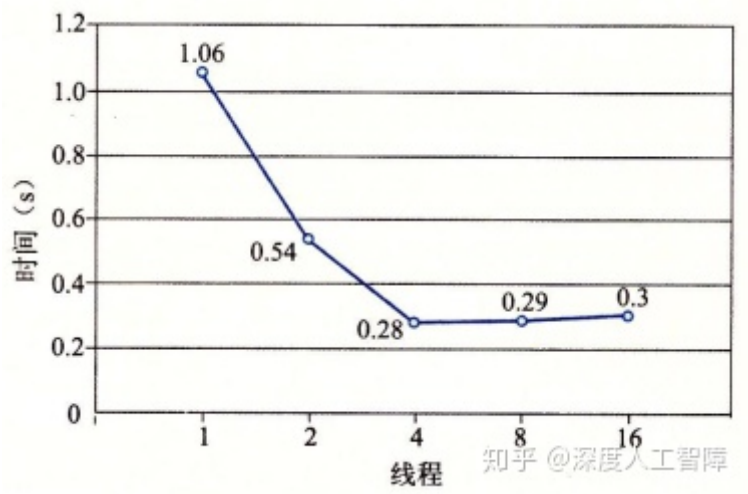

code/conc/psum-array.c

该程序的性能为

	线程数				
版本	1	2	4	8	16
psum-mutex	68.00	432.00	719.00	552.00	599.00
psum-array	7.26	3.64	1.91	1.85	1.84

可以发现去掉互斥锁后性能提升了很多。根据之前优化程序性能介绍的，要避免内存的读写，使用一个局部变量保存循环中计算的结果，在循环计算后再将其保存到数组中，程序性能为

	线程数				
版本	1	2	4	8	16
psum-mutex	68.00	432.00	719.00	552.00	599.00
psum-array	7.26	3.64	1.91	1.85	1.84
psum-local	1.06	0.54	0.28	0.29	0.30



在四核系统上的性能

可以一开始线程数目每增加一个，运行速度就下降一半，而当线程数到达4时，就不变了。这是因为这个程序运行在四核系统中，当线程数大于4时，每个核中就需要内核进行线程上下文切换来对线程进行调度，此时就会增加开销。并行程序一般一个核运行一个线程。

这里介绍几个度量并行程序利用并行性程度的**指标**：

- **加速比（Speedup）**：$S_p = \frac{T_1}{T_p}$

  其中p是核数目，T_p 是并行程序p个核上的运行时间

-
  - **绝对加速比（Absolute Speedup）**：T_1 是顺序程序执行时间。需要编写两套代码，较复杂，但能更真实衡量并行的好处。
  - **相对加速比（Relative Speedup）**：T_1 是并发程序在一个核上的执行时间。由于并发程序需要增加同步开销，会得到比绝对加速比更大的结果。

- **效率（Efficiency）**：$E_p = \frac{S_p}{p}$

  主要用来衡量并行化造成的开销，效率越高，说明程序在有用工作上花费更多时间，在同步和通信上花费较少的时间。

线程（t）	1	2	4	8	16
核（p）	1	2	4	4	4
运行时间（T_p）	1.06	0.54	0.28	0.29	0.30
加速比（S_p）	1	1.9	3.8	3.7	3.5
效率（E_p）	100%	98%	95%	91%	88%

接下来我们以排序为例来介绍并行程序，尝试实现并行版的快速排序。首先看一下快速排序的顺序版本

```
void qsort_serial(data_t *base, size_t nele) {
  if (nele <= 1)
    return;
  if (nele == 2) {
    if (base[0] > base[1])
      swap(base, base+1);
    return;
```

```
  }

  /* Partition returns index of pivot */
  size_t m = partition(base, nele);
  if (m > 1)
    qsort_serial(base, m);
  if (nele-1 > m+1)
    qsort_serial(base+m+1, nele-m-1);
}
```

可以发现我们首先对最左侧的部分进行排序，当左侧子部分排序完成时，才对右侧部分进行排序。最简单的并行方法就是将数据分成左右两部分，用两个线程分别对这两部分进行排序，称为**分而治之并行（Divide-and-Conquer Parallelism）**。此外我们还可以使用生产者-消费者模型。

发布于 2020-04-15