**习题6.1**

不可以，理由：n体问题里所有的粒子都是同时运动的，所以在计算粒子的运动时没有顺序的区别，因此可以据此采用并行的方式进行计算。但是，如果先完成一个粒子的所有计算，那么就会导致其他粒子的计算受到已完成计算的粒子的影响。

**习题6.4**

如下表所示：

|  |  |  |
| --- | --- | --- |
| 串行 | 2个线程 | 4个线程 |
| 0.343s | 0.702s | 0.616s |

在这个例子中，无论线程的数量是多少，调用循环的性能都更差。每次粒子的数据被更新时，线程获得和释放锁所增加的成本，比线程与粒子的子集一起工作减少的成本要高。

**习题6.8**

块划分比循环划分的性能要好，理由：使用块划分时，线程之间独立工作，如果线程访问高速缓存线路L时，其他线程也很可能访问高速缓存线程L。同时，循环分区的每个线程都需要访问每一行的x和y，这就很可能导致大量的错误共享到y。

**习题6.10**

由代码可知，当i=0~31时，线程0执行第一个for循环，所以会产生x的4次缺失和y的4次缺失。同理，线程1执行第二个for循环也有一样的缺失。因为第二个循环取决于chunksize的大小，所以，当chunksize=n/thread\_count=32时，有线程0产生8次缺失，然后它需要的数据已经被缓存下来了。那么chunksize=n/thread\_count，即相等时，第二个循环不会再产生缺失。

**习题6.18**

因为数据集会变大十分庞大，理由：队列需要同时将所有的部分旅行点存储在树的指定水平，而给定水平的旅行点的数量将会快速增长。符合下列公式：

(n-2+1)\*(n-3+1)……(n-k+1)

其中n为旅行点的个数，k为水平数，如果n=10，k=9，那么得到的结果是362880，这个数据不算特别庞大，如果n和k再加大一点点，那么就会产生指数爆炸，数据会变得特别大。

1. 如下所示：

while(queue\_sz(queue)<thread\_count)

{

    tour=Dequeue(queue);

    for each neighbouring city

       if(Feasible(tour,city))

       {

           Add\_city(tour,city);

           Enqueue(tour);

           Remove\_last\_city(tour);

       }

       Free\_tour(tour);

}

其中，queue\_sz(queue)的功能是返回queue中tour的数量，我们就可以迭代直到队列有至少thread\_count的tours，而不是直到队列为空。

1. 如下所示

for(q\_elt=my\_rank;q\_elt<queue\_sz(queue);q\_elt+=thread\_count)

{

    tour=Link\_to(queue,q\_elt);

    Push(stack,tour);

}

Barrier();

if(my\_rank==0)

{

    Free\_queue(queue);

}

当我门从队列的头部移动到尾部时，旅程的长度实际上是增加的。故采用循环分解的方法。

例如，Enqueue将其参数的一个副本放入队列，那么Link\_to可以返回一个指向tour的指针，然后通过Push()将这个tour堆到栈上。Barrier()用于确保每个线程完成其初始访问之前不会释放队列的结构。