8.5 在实践中设计并发代码

当为一个特殊的任务设计并发代码时，需要根据任务本身来考虑之前所提到的问题。为了展示以上的注意事项是如何应用的，我们将看一下在 C++ 标准库中三个标准函数的并行实现。当你遇到问题时，这里的例子可以作为很好的参照。在有较大的并发任务进行辅助下，我们也将实现一些函数。

我主要演示这些实现使用的技术，不过可能这些技术并不是最先进的；更多优秀的实现可以更好的利用硬件并发，不过这些实现可能需要到与并行算法相关的学术文献，或者是多线程的专家库中(比如：Inter的TBB[4])才能看到。

并行版的 `std::for_each` 可以看作为能最直观体现并行概念，就让我们从并行版的 `std::for_each` 开始吧！

8.5.1 并行实现： `std::for_each`

`std::for_each` 的原理很简单：其对某个范围中的元素，依次调用用户提供的函数。并行和串行调用的最大区别就是函数的调用顺序。 `std::for_each` 是对范围中的第一个元素调用用户函数，接着是第二个，以此类推，而在并行实现中对于每个元素的处理顺序就不能保证了，并且它们可能(我们希望如此)被并发的处理。

为了实现这个函数的并行版本，需要对每个线程上处理的元素进行划分。你事先知道元素数量，所以可以处理前对数据进行划分(详见8.1.1节)。假设只有并行任务运行，就可以使用 `std::thread::hardware_concurrency()` 来决定线程的数量。同样，这些元素都能被独立的处理，所以可以使用连续的数据块来避免伪共享(详见8.2.3节)。

这里的算法有点类似于并行版的 `std::accumulate` (详见8.4.1节)，不过比起计算每一个元素的加和，这里对每个元素仅仅使用了一个指定功能的函数。因为不需要返回结果，可以假设这可能会对简化代码，不过想要将异常传递给调用者，就需要使用 `std::packaged_task` 和 `std::future` 机制对线程中的异常进行转移。这里展示一个样本实现。

清单8.7 并行版 `std::for_each`

```
template<typename Iterator,typename Func>
void parallel_for_each(Iterator first,Iterator last,Func f)
{
  unsigned long const length=std::distance(first,last);

  if(!length)
    return;

  unsigned long const min_per_thread=25;
  unsigned long const max_threads=
    (length+min_per_thread-1)/min_per_thread;

  unsigned long const hardware_threads=
    std::thread::hardware_concurrency();

  unsigned long const num_threads=
    std::min(hardware_threads!=0?hardware_threads:2,max_threads);

  unsigned long const block_size=length/num_threads;

  std::vector<std::future<void> > futures(num_threads-1);  // 1
  std::vector<std::thread> threads(num_threads-1);
  join_threads joiner(threads);

  Iterator block_start=first;
  for(unsigned long i=0;i<(num_threads-1);++i)
  {
    Iterator block_end=block_start;
    std::advance(block_end,block_size);
    std::packaged_task<void(void)> task(  // 2
      [=]()
      {
        std::for_each(block_start,block_end,f);
      });
    futures[i]=task.get_future();
    threads[i]=std::thread(std::move(task));  // 3
    block_start=block_end;
  }
  std::for_each(block_start,last,f);
  for(unsigned long i=0;i<(num_threads-1);++i)
  {
    futures[i].get();  // 4
  }
}
```

代码结构与清单8.4的差不多。最重要的不同在于futures向量对 `std::future<void>` 类型①变量进行存储，因为工作线程不会返回值，并且简单的lambda函数会对block_start到block_end上的任务②执行f函数。这是为了避免传入线程的构造函数③。当工作线程不需要返回一个值时，调用futures[i].get()④只是提供检索工作线程异常的方法；如果不想把异常传递出去，就可以省略这一步。

实现并行 `std::accumulate` 的时候，使用 `std::async` 会简化代码；同样，parallel_for_each也可以使用 `std::async` 。实现如下所示。

清单8.8 使用 `std::async` 实现 `std::for_each`

```
template<typename Iterator,typename Func>
void parallel_for_each(Iterator first,Iterator last,Func f)
{
  unsigned long const length=std::distance(first,last);

  if(!length)
    return;

  unsigned long const min_per_thread=25;

  if(length<(2*min_per_thread))
  {
    std::for_each(first,last,f);  // 1
  }
  else
  {
    Iterator const mid_point=first+length/2;
    std::future<void> first_half=  // 2
      std::async(&parallel_for_each<Iterator,Func>,
                 first,mid_point,f);
    parallel_for_each(mid_point,last,f);  // 3
    first_half.get();  // 4
  }
}
```

和基于 `std::async` 的parallel_accumulate(清单8.5)一样，是在运行时对数据进行迭代划分的，而非在执行前划分好，这是因为你不知道你的库需要使用多少个线程。像之前一样，当你将每一级的数据分成两部分，异步执行另外一部分②，剩下的部分就不能再进行划分了，所以直接运行这一部分③；这样就可以直接对 `std::for_each` ①进行使用了。这里再次使用 `std::async` 和 `std::future` 的get()成员函数④来提供对异常的传播。

回到算法，函数需要对每一个元素执行同样的操作(这样的操作有很多种，初学者可能会想到 `std::count` 和 `std::replace`)，一个稍微复杂一些的例子就是使用 `std::find` 。

8.5.2 并行实现： `std::find`

接下来是 `std::find` 算法，因为这是一种不需要对数据元素做任何处理的算法。比如，当第一个元素就满足查找标准，那就没有必要对其他元素进行搜索了。将会看到，算法属性对于性能具有很大的影响，并且对并行实现的设计有着直接的影响。这个算法是一个很特别的例子，数据访问模式都会对代码的设计产生影响(详见8.3.2节)。该类中的另一些算法包括 `std::equal` 和 `std::any_of` 。

当你和妻子或者搭档，在一个纪念盒中找寻一张老照片，当找到这张照片时，就不会再看另外的照片了。不过，你得让其他人知道你已经找到照片了(比如，大喊一声“找到了！”)，这样其他人就会停止搜索了。很多算法的特性就是要对每一个元素进行处理，所以它们没有办法像 `std::find` 一样，一旦找到合适数据就停止执行。因此，你需要设计代码对其进行使用——当得到想要的答案就中断其他任务的执行，所以不能等待线程处理对剩下的元素进行处理。

如果不中断其他线程，那么串行版本的性能可能会超越并行版，因为串行算法可以在找到匹配元素的时候，停止搜索并返回。如果系统能支持四个并发线程，那么每个线程就可以对总数据量的1/4进行检查，并且在我们的实现只需要单核完成的1/4的时间，就能完成对所有元素的查找。如果匹配的元素在第一个1/4块中，串行算法将会返回第一个，因为算法不需要对剩下的元素进行处理了。

一种办法，中断其他线程的一个办法就是使用一个原子变量作为一个标识，在处理过每一个元素后就对这个标识进行检查。如果标识被设置，那么就有线程找到了匹配元素，所以算法就可以停止并返回了。用这种方式来中断线程，就可以将那些没有处理的数据保持原样，并且在更多的情况下，相较于串行方式，性能能提升很多。缺点就是，加载原子变量是一个很慢的操作，会阻碍每个线程的运行。

如何返回值和传播异常呢？现在你有两个选择。你可以使用一个future数组，使用 `std::packaged_task` 来转移值和异常，在主线程上对返回值和异常进行处理；或者使用 `std::promise` 对工作线程上的最终结果直接进行设置。这完全依赖于你想怎么样处理工作线程上的异常。如果想停止第一个异常(即使还没有对所有元素进行处理)，就可以使用 `std::promise` 对异常和最终值进行设置。另外，如果想要让其他工作线程继续查找，可以使用 `std::packaged_task` 来存储所有的异常，当线程没有找到匹配元素时，异常将再次抛出。

这种情况下，我会选择 `std::promise` ，因为其行为和 `std::find` 更为接近。这里需要注意一下搜索的元素是不是在提供的搜索范围内。因此，在所有线程结束前，获取future上的结果。如果被future阻塞住，所要查找的值不在范围内，就会持续的等待下去。实现代码如下。

清单8.9 并行find算法实现

```
template<typename Iterator,typename MatchType>
Iterator parallel_find(Iterator first,Iterator last,MatchType match)
{
  struct find_element  // 1
  {
    void operator()(Iterator begin,Iterator end,
                    MatchType match,
                    std::promise<Iterator>* result,
                    std::atomic<bool>* done_flag)
    {
      try
      {
        for(;(begin!=end) && !done_flag->load();++begin)  // 2
        {
          if(*begin==match)
          {
            result->set_value(begin);  // 3
            done_flag->store(true);  // 4
            return;
          }
        }
      }
      catch(...)  // 5
      {
        try
        {
          result->set_exception(std::current_exception());  // 6
          done_flag->store(true);
        }
        catch(...)  // 7
        {}
      }
    }
  };

  unsigned long const length=std::distance(first,last);

  if(!length)
```

```
    return last;

  unsigned long const min_per_thread=25;
  unsigned long const max_threads=
    (length+min_per_thread-1)/min_per_thread;

  unsigned long const hardware_threads=
    std::thread::hardware_concurrency();

  unsigned long const num_threads=
    std::min(hardware_threads!=0?hardware_threads:2,max_threads);

  unsigned long const block_size=length/num_threads;

  std::promise<Iterator> result;  // 8
  std::atomic<bool> done_flag(false);  // 9
  std::vector<std::thread> threads(num_threads-1);
  {  // 10
    join_threads joiner(threads);

    Iterator block_start=first;
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
      Iterator block_end=block_start;
      std::advance(block_end,block_size);
      threads[i]=std::thread(find_element(),  // 11
                             block_start,block_end,match,
                             &result,&done_flag);
      block_start=block_end;
    }
    find_element()(block_start,last,match,&result,&done_flag);  // 12
  }
  if(!done_flag.load())  //13
  {
    return last;
  }
  return result.get_future().get();  // 14
}
```

清单8.9中的函数主体与之前的例子相似。这次，由find_element类①的函数调用操作实现，来完成查找工作的。循环通过在给定数据块中的元素，检查每一步上的标识②。如果匹配的元素被找到，就将最终的结果设置到promise③当中，并且在返回前对done_flag④进行设置。

如果有一个异常被抛出，那么它就会被通用处理代码⑤捕获，并且在promise⑥尝中试存储前，对done_flag进行设置。如果对应promise已经被设置，设置在promise上的值可能会抛出一个异常，所以这里⑦发生的任何异常，都可以捕获并丢弃。

这意味着，当线程调用find_element查询一个值，或者抛出一个异常时，如果其他线程看到done_flag被设置，那么其他线程将会终止。如果多线程同时找到匹配值或抛出异常，它们将会对promise产生竞争。不过，这是良性的条件竞争；因为，成功的竞争者会作为“第一个”返回线程，因此这个结果可以接受。

回到parallel_find函数本身，其拥有用来停止搜索的promise⑧和标识⑨；随着对范围内的元素的查找⑪，promise和标识会传递到新线程中。主线程也使用find_element来对剩下的元素进行查找⑫。像之前提到的，需要在全部线程结束前，对结果进行检查，因为结果可能是任意位置上的匹配元素。这里将“启动-汇入”代码放在一个块中⑩，所以所有线程都会在找到匹配元素时⑬进行汇入。如果找到匹配元素，就可以调用 `std::future<Iterator>` (来自promise⑭)的成员函数get()来获取返回值或异常。

不过，这里假设你会使用硬件上所有可用的的并发线程，或使用其他机制对线程上的任务进行提前划分。就像之前一样，可以使用 `std::async` ，以及递归数据划分的方式来简化实现(同时使用 `C++` 标准库中提供的自动缩放工具)。使用 `std::async` 的parallel_find实现如下所示。

清单8.10 使用 `std::async` 实现的并行find算法

```
template<typename Iterator,typename MatchType>  // 1
Iterator parallel_find_impl(Iterator first,Iterator last,MatchType match,
                            std::atomic<bool>& done)
{
  try
  {
    unsigned long const length=std::distance(first,last);
    unsigned long const min_per_thread=25;  // 2
    if(length<(2*min_per_thread))  // 3
    {
      for(;(first!=last) && !done.load();++first)  // 4
      {
        if(*first==match)
        {
          done=true;  // 5
          return first;
        }
      }
      return last;  // 6
    }
```

```
    else
    {
      Iterator const mid_point=first+(length/2);  // 7
      std::future<Iterator> async_result=
        std::async(&parallel_find_impl<Iterator,MatchType>,  // 8
                   mid_point,last,match,std::ref(done));
      Iterator const direct_result=
        parallel_find_impl(first,mid_point,match,done);  // 9
      return (direct_result==mid_point)?
        async_result.get():direct_result;  // 10
    }
  }
  catch(...)
  {
    done=true;  // 11
    throw;
  }
}

template<typename Iterator,typename MatchType>
Iterator parallel_find(Iterator first,Iterator last,MatchType match)
{
  std::atomic<bool> done(false);
  return parallel_find_impl(first,last,match,done);  // 12
}
```

如果想要在找到匹配项时结束，就需要在线程之间设置一个标识来表明匹配项已经被找到。因此，需要将这个标识递归的传递。通过函数①的方式来实现是最简单的办法，只需要增加一个参数——一个done标识的引用，这个表示通过程序的主入口点传入⑫。

核心实现和之前的代码一样。通常函数的实现中，会让单个线程处理最少的数据项②；如果数据块大小不足于分成两半，就要让当前线程完成所有的工作了③。实际算法在一个简单的循环当中(给定范围)，直到在循环到指定范围中的最后一个，或找到匹配项，并对标识进行设置④。如果找到匹配项，标识done就会在返回前进行设置⑤。无论是因为已经查找到最后一个，还是因为其他线程对done进行了设置，都会停止查找。如果没有找到，会将最后一个元素last进行返回⑥。

如果给定范围可以进行划分，首先要在 `st::async` 在对第二部分进行查找⑧前，要找数据中点⑦，而且需要使用 `std::ref` 将done以引用的方式传递。同时，可以通过对第一部分直接进行递归查找。两部分都是异步的，并且在原始范围过大时，直接递归查找的部分可能会再细化。

如果直接查找返回的是mid_point，这就意味着没有找到匹配项，所以就要从异步查找中获取结果。如果在另一半中没有匹配项的话，返回的结果就一定是last，这个值的返回就代表了没有找到

匹配的元素⑩。如果“异步”调用被延迟(非真正的异步)，那么实际上这里会运行get()；这种情况下，如果对下半部分的元素搜索成功，那么就不会执行对上半部分元素的搜索了。如果异步查找真实的运行在其他线程上，那么async_result变量的析构函数将会等待该线程完成，所以这里不会有线程泄露。

像之前一样， `std::async` 可以用来提供“异常-安全”和“异常-传播”特性。如果直接递归抛出异常，future的析构函数就能让异步执行的线程提前结束；如果异步调用抛出异常，那么这个异常将会通过对get()成员函数的调用进行传播⑩。使用*try/catch*块只能捕捉在done发生的异常，并且当有异常抛出⑪时，所有线程都能很快的终止运行。不过，不使用*try/catch*的实现依旧没问题，不同的就是要等待所有线程的工作是否完成。

实现中一个重要的特性就是，不能保证所有数据都能被 `std::find` 串行处理。其他并行算法可以借鉴这个特性，因为要让一个算法并行起来这是必须具有的特性。如果有顺序问题，元素就不能并发的处理了。如果每个元素独立，虽然对于parallel_for_each不是很重要，不过对于parallel_find，即使在开始部分已经找到了匹配元素，也有可能返回范围中最后一个元素；如果在知道结果的前提下，这样的结果会让人很惊讶。

OK，现在你已经使用了并行化的 `std::find` 。如在本节开始说的那样，其他相似算法不需要对每一个数据元素进行处理，并且同样的技术可以使用到这些类似的算法上去。我们将在第9章中看到“中断线程”的问题。

为了完成我们的并行“三重奏”，我们将换一个角度来看一下 `std::partial_sum` 。对于这个算法，没有太多的文献可参考，不过让这个算法并行起来是一件很有趣的事。

8.5.3 并行实现： `std::partial_sum`

`std::partial_sum` 会计算给定范围中的每个元素，并用计算后的结果将原始序列中的值替换掉。比如，有一个序列[1，2，3，4，5]，在执行该算法后会成为：[1，3(1+2)，6(1+2+3)，10(1+2+3+4)，15(1+2+3+4+5)]。让这样一个算法并行起来会很有趣，因为这里不能讲任务分块，对每一块进行独立的计算。比如，原始序列中的第一个元素需要加到后面的一个元素中去。

确定某个范围部分和的一种的方式，就是在独立块中计算部分和，然后将第一块中最后的元素的值，与下一块中的所有元素进行相加，依次类推。如果有个序列[1，2，3，4，5，6，7，8，9]，然后将其分为三块，那么在第一次计算后就能得到[{1，3，6}，{4，9，15}，{7，15，24}]。然后将6(第一块的最后一个元素)加到第二个块中，那么就得到[{1，3，6}，{10，15，21}，{7，15，24}]。然后再将第二块的最后一个元素21加到第三块中去，就得到[{1，3，6}，{10，15，21}，{28，36，55}]。

将原始数据分割成块，加上之前块的部分和就能够并行了。如果每个块中的末尾元素都是第一个被更新的，那么块中其他的元素就能被其他线程所更新，同时另一个线程对下一块进行更新，等等。当处理的元素比处理核心的个数多的时候，这样完成工作没问题，因为每一个核芯在每一个阶段都有合适的数据可以进行处理。

如果有很多的处理器(就是要比处理的元素个数多)，那么之前的方式就无法正常工作了。如果还是将工作划分给每个处理器，那么在第一步就没必要去做了。这种情况下，传递结果就意味着让处理器进行等待，这时需要给这些处于等待中的处理器一些工作。所以，可以采用完全不同的方式来处理这个问题。比起将数据块中的最后一个元素的结果向后面的元素块传递，可以对部分结果进行传播：第一次与相邻的元素(距离为1)相加和(和之前一样)，之后和距离为2的元素相加，在后来和距离为4的元素相加，以此类推。比如，初始序列为[1，2，3，4，5，6，7，8，9]，第一次后为[1，3，5，7，9，11，13，15，17]，第二次后为[1，3，6，10，14，18, 22，26，30]，下一次就要隔4个元素了。第三次后[1, 3, 6, 10, 15, 21, 28, 36, 44]，下一次就要隔8个元素了。第四次后[1, 3, 6, 10, 15, 21, 28, 36, 45]，这就是最终的结果。虽然，比起第一种方法多了很多步骤，不过在可并发平台下，这种方法提高了并行的可行性；每个处理器可在每一步中处理一个数据项。

总体来说，当有N个操作时(每步使用一个处理器)第二种方法需要log(N)[底为2]步；在本节中，N就相当于数据链表的长度。比起第一种，每个线程对分配块做N/k个操作，然后在做N/k次结果传递(这里的k是线程的数量)。因此，第一种方法的时间复杂度为O(N)，不过第二种方法的时间复杂度为Q(Nlog(N))。当数据量和处理器数量相近时，第二种方法需要每个处理器上log(N)个操作，第一种方法中每个处理器上执行的操作数会随着k的增加而增多，因为需要对结果进行传递。对于处理单元较少的情况，第一种方法会比较合适；对于大规模并行系统，第二种方法比较合适。

不管怎么样，先将效率问题放一边，让我们来看一些代码。下面清单实现的，就是第一种方法。

清单8.11 使用划分的方式来并行的计算部分和

```
template<typename Iterator>
void parallel_partial_sum(Iterator first,Iterator last)
{
  typedef typename Iterator::value_type value_type;

  struct process_chunk  // 1
  {
    void operator()(Iterator begin,Iterator last,
                    std::future<value_type>* previous_end_value,
                    std::promise<value_type>* end_value)
    {
      try
      {
        Iterator end=last;
```

```
        ++end;
        std::partial_sum(begin,end,begin);  // 2
        if(previous_end_value)  // 3
        {
          value_type& addend=previous_end_value->get();  // 4
          *last+=addend;  // 5
          if(end_value)
          {
            end_value->set_value(*last);  // 6
          }
          std::for_each(begin,last,[addend](value_type& item)  // 7
                        {
                          item+=addend;
                        });
        }
        else if(end_value)
        {
          end_value->set_value(*last);  // 8
        }
      }
      catch(...)  // 9
      {
        if(end_value)
        {
          end_value->set_exception(std::current_exception());  // 10
        }
        else
        {
          throw;  // 11
        }
      }
    }
  };

  unsigned long const length=std::distance(first,last);

  if(!length)
    return last;

  unsigned long const min_per_thread=25;  // 12
  unsigned long const max_threads=
    (length+min_per_thread-1)/min_per_thread;

  unsigned long const hardware_threads=
    std::thread::hardware_concurrency();
```

```
60
61    unsigned long const num_threads=
62      std::min(hardware_threads!=0?hardware_threads:2,max_threads);
63
64    unsigned long const block_size=length/num_threads;
65
66    typedef typename Iterator::value_type value_type;
67
68    std::vector<std::thread> threads(num_threads-1);  // 13
69    std::vector<std::promise<value_type> >
70      end_values(num_threads-1);  // 14
71    std::vector<std::future<value_type> >
72      previous_end_values;  // 15
73    previous_end_values.reserve(num_threads-1);  // 16
74    join_threads joiner(threads);
75
76    Iterator block_start=first;
77    for(unsigned long i=0;i<(num_threads-1);++i)
78    {
79      Iterator block_last=block_start;
80      std::advance(block_last,block_size-1);  // 17
81      threads[i]=std::thread(process_chunk(),  // 18
82                             block_start,block_last,
83                             (i!=0)?&previous_end_values[i-1]:0,
84                             &end_values[i]);
85      block_start=block_last;
86      ++block_start;  // 19
87      previous_end_values.push_back(end_values[i].get_future());  // 20
88    }
89    Iterator final_element=block_start;
90    std::advance(final_element,std::distance(block_start,last)-1);  // 21
91    process_chunk()(block_start,final_element,  // 22
92                    (num_threads>1)?&previous_end_values.back():0,
93                    0);
94  }
```

这个实现中，使用的结构体和之前算法中的一样，将问题进行分块解决，每个线程处理最小的数据块⑫。其中，有一组线程⑬和一组promise⑭，用来存储每块中的最后一个值；并且实现中还有一组future⑮，用来对前一块中的最后一个值进行检索。可以为future⑯做些储备，以避免生成新线程时，再分配内存。

主循环和之前一样，不过这次是让迭代器指向了每个数据块的最后一个元素，而不是作为一个普通值传递到最后⑰，这样就方便向其他块传递当前块的最后一个元素了。实际处理是在process_chunk函数对象中完成的，这个结构体看上去不是很长；当前块的开始和结束迭代器和前

块中最后一个值的future一起，作为参数进行传递，并且promise用来保留当前范围内最后一个值的原始值⑱。

生成新的线程后，就对开始块的ID进行更新，别忘了传递最后一个元素⑲，并且将当前块的最后一个元素存储到future，上面的数据将在循环中再次使用到⑳。

在处理最后一个数据块前，需要获取之前数据块中最后一个元素的迭代器(21)，这样就可以将其作为参数传入process_chunk(22)中了。 `std::partial_sum` 不会返回一个值，所以在最后一个数据块被处理后，就不用再做任何事情了。当所有线程的操作完成时，求部分和的操作也就算完成了。

OK，现在来看一下process_chunk函数对象①。对于整块的处理是始于对 `std::partial_sum` 的调用，包括对于最后一个值的处理②，不过得要知道当前块是否是第一块③。如果当前块不是第一块，就会有一个previous_end_value值从前面的块传过来，所以这里需要等待这个值的产生④。为了将算法最大程度的并行，首先需要对最后一个元素进行更新⑤，这样你就能将这个值传递给下一个数据块(如果有下一个数据块的话)⑥。当完成这个操作，就可以使用 `std::for_each` 和简单的lambda函数⑦对剩余的数据项进行更新。

如果previous_end_value值为空，当前数据块就是第一个数据块，所以只需要为下一个数据块更新end_value⑧(如果有下一个数据块的话——当前数据块可能是唯一的数据块)。

最后，如果有任意一个操作抛出异常，就可以将其捕获⑨，并且存入promise⑩，如果下一个数据块尝试获取前一个数据块的最后一个值④时，异常会再次抛出。处理最后一个数据块时，异常会全部重新抛出⑪，因为抛出动作一定会在主线程上进行。

因为线程间需要同步，这里的代码就不容易使用 `std::async` 重写。任务等待会让线程中途去执行其他的任务，所以所有的任务必须同时执行。

基于块，以传递末尾元素值的方法就介绍到这里，让我们来看一下第二种计算方式。

实现以2的幂级数为距离部分和算法

第二种算法通过增加距离的方式，让更多的处理器充分发挥作用。在这种情况下，没有进一步同步的必要了，因为所有中间结果都直接传递到下一个处理器上去了。不过，在实际中我们很少见到，单个处理器处理对一定数量的元素执行同一条指令，这种方式成为*单指令-多数据流*(SIMD)。因此，代码必须能处理通用情况，并且需要在每步上对线程进行显式同步。

完成这种功能的一种方式是使用*栅栏*(barrier)——一种同步机制：只有所有线程都到达栅栏处，才能进行之后的操作；先到达的线程必须等待未到达的线程。 C++ 11标准库没有直接提供这样的工具，所以你得自行设计一个。

试想游乐场中的过山车。如果有适量的游客在等待，那么过山车管理员就要保证，在过山车启动前，每一个位置都得坐一个游客。栅栏的工作原理也一样：你已经知道了“座位”的数量，线程就是要等待所有“座位”都坐满。当等待线程够数，那么它们可以继续运行；这时，栅栏会重置，并且会让下一拨线程开始托带。通常，会在循环中这样做，当同一个线程再次到达栅栏处，它会再次等待。这种方法是为了让线程同步，所以不会有线程在其他未完成的情况下，就去完成下一个任务。如果有线程提前执行，对于这样一个算法，就是一场灾难，因为提前出发的线程可能会修改要被其他线程使用到的数据，后面线程获取到的数据就不是正确数据了。

下面的代码就简单的实现了一个栅栏。

清单8.12 简单的栅栏类

```
class barrier
{
  unsigned const count;
  std::atomic<unsigned> spaces;
  std::atomic<unsigned> generation;
public:
  explicit barrier(unsigned count_):  // 1
    count(count_),spaces(count),generation(0)
  {}

  void wait()
  {
    unsigned const my_generation=generation;  // 2
    if(!--spaces)  // 3
    {
      spaces=count;  // 4
      ++generation;  // 5
    }
    else
    {
      while(generation==my_generation)  // 6
        std::this_thread::yield();  // 7
    }
  }
};
```

这个实现中，用一定数量的“座位”构造了一个barrier①，这个数量将会存储count变量中。起初，栅栏中的spaces与count数量相当。当有线程都在等待时，spaces的数量就会减少③。当spaces的数量减到0时，spaces的值将会重置为count④，并且generation变量会增加，以向线程发出信号，让这些等待线程能够继续运行⑤。如果spaces没有到达0，那么线程会继续等待。这个实现使

用了一个简单的自旋锁⑥，对generation的检查会在wait()开始的时候进行②。因为generation只会在所有线程都到达栅栏的时候更新⑤，在等待的时候使用yield()⑦就不会让CPU处于忙等待的状态。

这个实现比较“简单”的真实意义：使用自旋等待的情况下，如果让线程等待很长时间就不会很理想，并且如果超过count数量的线程对wait()进行调用，这个实现就没有办法工作了。如果想要很好的处理这样的情况，必须使用一个更加健壮(更加复杂)的实现。我依旧坚持对原子变量操作顺序的一致性，因为这会让事情更加简单，不过有时还是需要放松这样的约束。全局同步对于大规模并行架构来说是消耗巨大的，因为相关处理器会穿梭于存储栅栏状态的缓存行中(可见8.2.2中对乒乓缓存的讨论)，所以需要格外的小心，来确保使用的是最佳同步方法。

不论怎么样，这些都需要你考虑到；需要有固定数量的线程执行同步循环。好吧，大多数情况下线程数量都是固定的。你可能还记得，代码起始部分的几个数据项，只需要几步就能得到其最终值。这就意味着，无论是让所有线程循环处理范围内的所有元素，还是让栅栏来同步线程，都会递减count的值。我会选择后者，因为其能避免线程做不必要的工作，仅仅是等待最终步骤完成。

这意味着你要将count改为一个原子变量，这样在多线程对其进行更新的时候，就不需要添加额外的同步：

```
std::atomic<unsigned> count;
```

初始化保持不变，不过当spaces的值被重置后，你需要显式的对count进行load()操作：

```
spaces=count.load();
```

这就是要对wait()函数的改动；现在需要一个新的成员函数来递减count。这个函数命名为done_waiting()，因为当一个线程完成其工作，并在等待的时候，才能对其进行调用它：

```
1  void done_waiting()
2  {
3    --count;  // 1
4    if(!--spaces)  // 2
5    {
6      spaces=count.load();  // 3
7      ++generation;
8    }
9  }
```

实现中，首先要减少count①，所以下一次spaces将会被重置为一个较小的数。然后，需要递减spaces的值②。如果不做这些操作，有些线程将会持续等待，因为spaces被旧的count初始化，大于期望值。一组当中最后一个线程需要对计数器进行重置，并且递增generation的值③，就像在wait()里面做的那样。最重要的区别：最后一个线程不需要等待。当最后一个线程结束，整个等待也就随之结束！

现在就准备开始写部分和的第二个实现吧。在每一步中，每一个线程都在栅栏出调用wait()，来保证线程所处步骤一致，并且当所有线程都结束，那么最后一个线程会调用done_waiting()来减少count的值。如果使用两个缓存对原始数据进行保存，栅栏也可以提供你所需要的同步。每一步中，线程都会从原始数据或是缓存中读取数据，并且将新值写入对应位置。如果有线程先从原始数据处获取数据，那下一步就从缓存上获取数据(或相反)。这就能保证在读与写都是由独立线程完成，并不存在条件竞争。当线程结束等待循环，就能保证正确的值最终被写入到原始数据当中。下面的代码就是这样的实现。

清单8.13 通过两两更新对的方式实现partial_sum

```
struct barrier
{
  std::atomic<unsigned> count;
  std::atomic<unsigned> spaces;
  std::atomic<unsigned> generation;

  barrier(unsigned count_):
    count(count_),spaces(count_),generation(0)
  {}

  void wait()
  {
    unsigned const gen=generation.load();
    if(!--spaces)
    {
      spaces=count.load();
      ++generation;
    }
    else
    {
      while(generation.load()==gen)
      {
        std::this_thread::yield();
      }
    }
  }

```

```
  void done_waiting()
  {
    --count;
    if(!--spaces)
    {
      spaces=count.load();
      ++generation;
    }
  }
};

template<typename Iterator>
void parallel_partial_sum(Iterator first,Iterator last)
{
  typedef typename Iterator::value_type value_type;

  struct process_element  // 1
  {
    void operator()(Iterator first,Iterator last,
                    std::vector<value_type>& buffer,
                    unsigned i,barrier& b)
    {
      value_type& ith_element=*(first+i);
      bool update_source=false;

      for(unsigned step=0,stride=1;stride<=i;++step,stride*=2)
      {
        value_type const& source=(step%2)?  // 2
          buffer[i]:ith_element;

        value_type& dest=(step%2)?
          ith_element:buffer[i];

        value_type const& addend=(step%2)?  // 3
          buffer[i-stride]:*(first+i-stride);

        dest=source+addend;  // 4
        update_source=!(step%2);
        b.wait();  // 5
      }
      if(update_source)  // 6
      {
        ith_element=buffer[i];
      }
      b.done_waiting();  // 7
```

```
    }
  };

  unsigned long const length=std::distance(first,last);

  if(length<=1)
    return;

  std::vector<value_type> buffer(length);
  barrier b(length);

  std::vector<std::thread> threads(length-1);  // 8
  join_threads joiner(threads);

  Iterator block_start=first;
  for(unsigned long i=0;i<(length-1);++i)
  {
    threads[i]=std::thread(process_element(),first,last,  // 9
                           std::ref(buffer),i,std::ref(b));
  }
  process_element()(first,last,buffer,length-1,b);  // 10
}
```

代码的整体结构应该不用说了。process_element类有函数调用操作可以用来做具体的工作①，就是运行一组线程⑨，并将线程存储到vector中⑧，同样还需要在主线程中对其进行调用⑩。这里与之前最大的区别就是，线程的数量是根据列表中的数据量来定的，而非根据 `std::thread::hardware_concurrency` 。如我之前所说，除非你使用的是一个大规模并行的机器，因为这上面的线程都十分廉价(虽然这样的方式并不是很好)，还能为我们展示了其整体结构。这个结构在有较少线程的时候，每一个线程只能处理源数据中的部分数据，当没有足够的线程支持该结构时，效率要比传递算法低。

不管怎样，主要的工作都是调用process_element的函数操作符来完成的。每一步，都会从原始数据或缓存中获取第i个元素②，并且将获取到的元素加到指定stride的元素中去③，如果从原始数据开始读取的元素，加和后的数需要存储在缓存中④。然后，在开始下一步前，会在栅栏处等待⑤。当stride超出了给定数据的范围，当最终结果已经存在缓存中时，就需要更新原始数据中的数据，同样这也意味着本次加和结束。最后，在调用栅栏中的done_waiting()函数⑦。

注意这个解决方案并不是异常安全的。如果某个线程在process_element执行时抛出一个异常，其就会终止整个应用。这里可以使用一个 `std::promise` 来存储异常，就像在清单8.9中parallel_find的实现，或仅使用一个被互斥量保护的 `std::exception_ptr` 即可。

总结下这三个例子。希望其能保证我们了解8.1、8.2、8.3和8.4节中提到的设计考量，并且证明了这些技术在真实的代码中，需要承担些什么责任。

[4] http://threadingbuildingblocks.org/