2.4 运行时决定线程数量

`std::thread::hardware_concurrency()` 在新版C++标准库中是一个很有用的函数。这个函数将返回能同时并发在一个程序中的线程数量。例如，多核系统中，返回值可以是CPU核芯的数量。返回值也仅仅是一个提示，当系统信息无法获取时，函数也会返回0。但是，这也无法掩盖这个函数对启动线程数量的帮助。

清单2.8实现了一个并行版的 `std::accumulate` 。代码中将整体工作拆分成小任务交给每个线程去做，其中设置最小任务数，是为了避免产生太多的线程。程序可能会在操作数量为0的时候抛出异常。比如， `std::thread` 构造函数无法启动一个执行线程，就会抛出一个异常。在这个算法中讨论异常处理，已经超出现阶段的讨论范围，这个问题我们将在第8章中再来讨论。

清单2.8 原生并行版的 `std::accumulate`

```
template<typename Iterator,typename T>
struct accumulate_block
{
  void operator()(Iterator first,Iterator last,T& result)
  {
    result=std::accumulate(first,last,result);
  }
};

template<typename Iterator,typename T>
T parallel_accumulate(Iterator first,Iterator last,T init)
{
  unsigned long const length=std::distance(first,last);

  if(!length) // 1
    return init;

  unsigned long const min_per_thread=25;
  unsigned long const max_threads=
      (length+min_per_thread-1)/min_per_thread; // 2

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

```

```
  unsigned long const num_threads=  // 3
      std::min(hardware_threads != 0 ? hardware_threads : 2, max_threads);

  unsigned long const block_size=length/num_threads; // 4

  std::vector<T> results(num_threads);
  std::vector<std::thread> threads(num_threads-1);  // 5

  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);  // 6
    threads[i]=std::thread(     // 7
        accumulate_block<Iterator,T>(),
        block_start,block_end,std::ref(results[i]));
    block_start=block_end;  // 8
  }
  accumulate_block<Iterator,T>()(
      block_start,last,results[num_threads-1]); // 9
  std::for_each(threads.begin(),threads.end(),
       std::mem_fn(&std::thread::join));  // 10

  return std::accumulate(results.begin(),results.end(),init); // 11
}
```

函数看起来很长，但不复杂。如果输入的范围为空①，就会得到init的值。反之，如果范围内多于一个元素时，都需要用范围内元素的总数量除以线程(块)中最小任务数，从而确定启动线程的最大数量②，这样能避免无谓的计算资源的浪费。比如，一台32芯的机器上，只有5个数需要计算，却启动了32个线程。

计算量的最大值和硬件支持线程数中，较小的值为启动线程的数量③。因为上下文频繁的切换会降低线程的性能，所以你肯定不想启动的线程数多于硬件支持的线程数量。当 `std::thread::hardware_concurrency()` 返回0，你可以选择一个合适的数作为你的选择；在本例中,我选择了"2"。你也不想在一台单核机器上启动太多的线程，因为这样反而会降低性能，有可能最终让你放弃使用并发。

每个线程中处理的元素数量,是范围中元素的总量除以线程的个数得出的④。对于分配是否得当，我们会在后面讨论。

现在，确定了线程个数，通过创建一个 `std::vector<T>` 容器存放中间结果，并为线程创建一个 `std::vector<std::thread>` 容器⑤。这里需要注意的是，启动的线程数必须比num_threads少1

个，因为在启动之前已经有了一个线程(主线程)。

使用简单的循环来启动线程：block_end迭代器指向当前块的末尾⑥，并启动一个新线程为当前块累加结果⑦。当迭代器指向当前块的末尾时，启动下一个块⑧。

启动所有线程后，⑨中的线程会处理最终块的结果。对于分配不均，因为知道最终块是哪一个，那么这个块中有多少个元素就无所谓了。

当累加最终块的结果后，可以等待 `std::for_each` ⑩创建线程的完成(如同在清单2.7中做的那样)，之后使用 `std::accumulate` 将所有结果进行累加⑪。

结束这个例子之前，需要明确：T类型的加法运算不满足结合律(比如，对于float型或double型，在进行加法操作时，系统很可能会做截断操作)，因为对范围中元素的分组，会导致parallel_accumulate得到的结果可能与 `std::accumulate` 得到的结果不同。同样的，这里对迭代器的要求更加严格：必须都是向前迭代器，而 `std::accumulate` 可以在只传入迭代器的情况下工作。对于创建出results容器，需要保证T有默认构造函数。对于算法并行，通常都要这样的修改；不过，需要根据算法本身的特性，选择不同的并行方式。算法并行会在第8章有更加深入的讨论。需要注意的：因为不能直接从一个线程中返回一个值，所以需要传递results容器的引用到线程中去。另一个办法，通过地址来获取线程执行的结果；第4章中，我们将使用*期望*(futures)完成这种方案。

当线程运行时，所有必要的信息都需要传入到线程中去，包括存储计算结果的位置。不过，并非总需如此：有时候这是识别线程的可行方案，可以传递一个标识数，例如清单2.7中的i。不过，当需要标识的函数在调用栈的深层，同时其他线程也可调用该函数，那么标识数就会变的捉襟见肘。好消息是在设计C++的线程库时，就有预见了这种情况，在之后的实现中就给每个线程附加了唯一标识符。