并行处理技术项目——实验报告

161220049　黄奕诚

一、项目情况概述

更简洁扼要的说明可见 README.md。程序执行说明可见 ReadMe.txt。按照项目实验要求，我使用了 Java 多线程模拟并行处理，尝试了不同的多线程接口，在确保算法正确性和输出正确性的前提下，对其运行时间进行了统计、比较。本次实验完全由个人独立完成。

二、算法伪代码

a. 串行快速排序（算法 1、2）：

Algorithm 1 quicksort(A, lo, hi)

```
1: if lo < hi then
2:     p := partition(A, lo, hi)
3:     quicksort(A, lo, p − 1)
4:     quicksort(A, p + 1, hi)
5: end if
```

Algorithm 2 partition(A, lo, hi)

```
1: pivot := A[hi]
2: i := lo
3: for j := lo to hi − 1 do
4:     if A[j] < pivot then
5:         swap A[i] with A[j]
6:     end if
7:     i := i + 1
8: end for
9: swap A[i] with A[hi]
10: return i
```

b. 串行枚举排序（算法 3）：

Algorithm 3 enumsort(A, lo, hi)

```
1: for i := lo to hi do
2:     k := lo
3:     for j := lo to hi do
4:         if a[i] > a[j] or (a[i] = a[j] and i > j) then
5:             k := k + 1
6:         end if
7:     end for
8:     b[k] := a[i]
9: end for
```

c. 串行归并排序（算法 4、5）：

Algorithm 4 mergesort(A, p, r)

```
1: if p < r then
2:     q := ⌊(p + r)/2⌋
3:     mergesort(A, p, q)
4:     mergesort(A, q + 1, r)
5:     merge(A, p, q, r)
6: end if
```

d. 并行快速排序（算法 6、7）：输入为无序数组 $data[1, n]$，使用的处理器个数为 m，输出为有序数组 $data[1, n]$。

e. 并行枚举排序（算法 8）：输入为无序数组 $a[1, n]$，输出为无序数组 $b[1, n]$。

f. 并行归并排序（算法 9、10）：输入为无序数组 $a[1, n]$，输出为有序数组 $a[1, n]$，处理器个数为 m。

三、Java 实现技术

Java 提供了多种多线程接口来对创建的线程进行调度、执行和通信，我在此项目中利用 ForkJoinPool 及 RecursiveAction 来维护线程池，并规划每一个线程的任务，监听每一个线程的完成情况，并保证线程尽量充分利用四个处理器（我的电脑的配置是双核四处理器）。对于 QuickSort 的并行实

Algorithm 5 merge(A, p, q, r)

1: $n_1 := q - p + 1$
2: $n_2 := r - q$
3: $L[1 \dots n_1 + 1]$ and $R[1 \dots n_2 + 1]$ be arrays
4: **for** $i := 1$ **to** n_1 **do**
5: $\quad L[i] := A[p + i - 1]$
6: **end for**
7: **for** $j := 1$ **to** n_2 **do**
8: $\quad R[j] := A[q + j]$
9: **end for**
10: $L[n_1 + 1] := \infty$
11: $R[n_2 + 1] := \infty$
12: $i := 1$
13: $j := 1$
14: **for** $k := p$ **to** r **do**
15: $\quad$ **if** $L[i] \leq R[j]$ **then**
16: $\quad\quad A[k] := L[i]$, $i := i + 1$
17: $\quad$ **else**
18: $\quad\quad A[k] := R[j]$, $j := j + 1$
19: $\quad$ **end if**
20: **end for**

Algorithm 6 top_module$(data, 1, n, m)$

1: para_quicksort$(data, n, m, 0)$

Algorithm 7 para_quicksort$(data, i, j, m, id)$

1: **if** $m = 0$ **then**
2: $\quad$ call quicksort$(data, i, j)$
3: **else**
4: $\quad r :=$partition$(data, i, j)$
5: $\quad P_{id}$ send $data[r + 1, j]$ to P_{id+1}
6: $\quad$ para_quicksort$(data, i, r - 1, m - 1, id)$
7: $\quad$ para_quicksort$(data, r + 1, j, m - 1, id + 1)$
8: $\quad P_{id+1}$ send $data[r + 1, j]$ back to P_{id}
9: **end if**

Algorithm 8 para_enumsort(a, n)

```
1: P_0 send a[1, n] to P_{1,...n}
2: for all P_i where 1 ≤ i ≤ n para-do
3: for all P_i where 1 ≤ i ≤ n para- do
4:    k := 1
5:    for j := 1 to n do
6:       if a[i] > a[j] or (a[i] = a[j] and i > j) then
7:          k := k + 1
8:       end if
9:    end for
10: end for
11: P_0 collects k and get A[1, n]
```

Algorithm 9 top_module(a, n, m)

```
para_mergesort(a, 1, n, m, 1)
```

Algorithm 10 para_mergesort(a, lo, hi, m, id)

```
1: if lo < hi then
2:    q := ⌊(lo + hi)/2⌋
3:    if id + 1 ≤ m then
4:       let P_{id+1} do para_mergesort(a, lo, q, m, id + 1)
5:       let P_{id+2} do para_mergesort(a, q + 1, hi, m, id + 2)
6:       wait for P_{id+1} and P_{id+2} to finish, then merge(a, lo, q, hi)
7:    else
8:       mergesort(a, lo, q)
9:       mergesort(a, q + 1, hi)
10:      merge(a, lo, q, hi)
11:   end if
12: end if
```

现，我还尝试了 ExecutorService 和 Future 的接口；对于 MergeSort，我也尝试了朴素的 Thread 类的 start、invoke 方法实现。以下是对这些实现方法的举例说明：

a. ForkJoinPool 示例

```
final ForkJoinPool forkJoinPoolQsp = new
    ForkJoinPool(Runtime.getRuntime().availableProcessors());
forkJoinPoolQsp.invoke(new QuickSortParallelTask2(data, 0,
    data.length - 1));
```

这里创建了一个 forkJoinPool 对象，并规定了并行的级别，一般指定为可利用的处理器个数。接着 invoke 方法等待主任务的执行完成，然后主线程才继续向下执行。对于这里的 QuickSortParallelTask2，它是基于 RecursiveAction 接口的一个类，重写了需要的 compute 函数，在每次对象被创建后都会调用 compute 函数。RecursiveAction 的部分内容如下所示：

```
public abstract class RecursiveAction extends ForkJoinTask<Void>{
  protected abstract void compute();
  ...
}
```

以快速排序的并行实现为例，我统计了当前已创建的新线程，在二叉树的实现中，保证创建新线程的数目不超过处理器的数目，节省一个线程在切换不同任务时的开销。当创建的新线程个数达到上限后，调用串行的快速排序算法即可。下面为 ParallelSort 的实现：

```
private void parallelSort(int[] arr, int left, int right){
  List<QuickSortParallelTask2> futures = new Vector<>();
  int l = partition(arr, left, right);
  if (l - left > 1){
    if (count < numOfProcessors){
      count++;
      QuickSortParallelTask2 leftTask = new
          QuickSortParallelTask2(arr, left, l - 1);
      futures.add(leftTask);
    }
```

```
      else{
        sort(arr, left, l - 1);
      }
    }
    if (right - l > 1){
      if (count < numOfProcessors){
        count++;
        QuickSortParallelTask2 rightTask = new
            QuickSortParallelTask2(arr, l + 1, right);
        futures.add(rightTask);
      }
      else{
        sort(arr, l + 1, right);
      }
      if (!futures.isEmpty())
        invokeAll(futures);
    }
}
```

其中 sort 与 partition 皆为串行执行的方法。这里的 invokeAll 保证了全局的同步，若删去，则可能会引发并发 BUG。

b. ExecutorService 及 List<Future> 示例：

```
List<Future> futures = new Vector<>();
ExecutorService executorService =
    Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
QuickSortParallelTask mainQspTask = new QuickSortParallelTask(data,
    0, data.length - 1, executorService, futures);
futures.add(executorService.submit(mainQspTask));
while (!futures.isEmpty()){
  Future topFuture = futures.remove(0);
  try{
    topFuture.get();
  } catch(Exception e){
    e.printStackTrace();
  }
}
executorService.shutdown();
```

这里 ExecutorService 的 submit 方法用于提交 Runnable 的任务，指定线程池的大小，并由 Future 链表管理其执行情况。在创建新线程的过程中，同一个 Future 链表对象会添入更多的任务。然后 Future 的 get 方法会等待该任务执行结束，保证同步。在这里，QuickSortParallelTask 基于接口 Runnable 重写了 run 方法，其它思路与上一种实现方法类似，不再赘述。

c. 朴素的 Thread 的 start/join 方法示例。手动管理了新线程创建的数目，并用 join 方法保证了两个新线程执行结束后才能 merge。示例片段如下：

```
@Override
public void run() { sort(data, begin, end); }
private void sort(int[] arr, int left, int right){
  if (left >= right)
    return;
  int mid = (left + right) / 2;
  if (count < numOfProcessors){
    count += 2;
    Thread leftThread = new Thread(new MergeSortParallelTask(arr,
        left, mid));
    Thread rightThread = new Thread(new MergeSortParallelTask(arr,
        mid + 1, right));
    leftThread.start();
    rightThread.start();
    try {
      leftThread.join();
      rightThread.join();
    } catch(InterruptedException e){
      e.printStackTrace();
    }
    merge(left, right, mid);
  }
  else{
    sort(arr, left, mid);
    sort(arr, mid + 1, right);
    merge(left, right, mid);
  }
}
```

此外，在代码的设计方面，我还在每个算法执行结束后利用 assert 检查数组的排序结果，保证输出的正确性，然后再输出到文件。该项目是用 Intellij IDEA 构建而成的，不过我还手写了一个可以几乎等效编译、运行的 sh 文件，只要 $ bash run.sh 即可编译运行输出结果。

四、性能比较与分析

1. 性能比较

先作如下标记:

排序算法名称	标记
串行快速排序	QS
用 ExecutorService 和 Future List 实现的并行快速排序	$QSP1$
用 ForkJoinPool 和 RecursiveAction 实现的并行快速排序	$QSP2$
串行枚举排序	ES
用 ForkJoinPool 和 RecursiveAction 实现的并行枚举排序	ESP
串行归并排序	MS
用 Thread 方法实现的并行归并排序	$MSP1$
用 ForkJoinPool 和 RecursiveAction 实现的并行归并排序	$MSP2$

在双核四处理器的机器上，对每个算法的实现，在确保正确性的前提下，分别进行 10 次测试，取平均值，可以得出如下运行时间结果对比表——

算法实现	运行时间 (ms)
QS	62.6
$QSP1$	28.7
$QSP2$	30.1
ES	4357.5
ESP	522.5
MS	11.4
$MSP1$	15.0
$MSP2$	23.2

2. 性能分析

从测试结果来看，两种快速排序的并行实现分别比串行实现快了 54% 和 52%，而枚举排序的并行时间比串行实现快了 88%，提速效果显著。而归并排序的并行实现则比串行实现分别慢了 24% 和 51%。

最朴素的想法是，根据我用 Java 完成的并行程序实现，原先单核串行的程序可以变成多处理器并行执行任务，并进行通信和同步，在将可以并行化的串行任务并行处理时，节省了执行时间的开销，但也增加了创建新线程、拷贝及等待线程执行结束的通信开销。

下面对每个具体算法的串行实现和并行实现的性能作出评估：

a. 对于快速排序算法，我采用了比较常规的二叉树并行方法，并将靠近根结点的几个任务交给不同的处理器去执行，将远离根结点的任务串行执行，这样一定程度上节省了创建和管理那些“做很少的任务“的小线程的开销。运行测试结果也在合理范围之内。

b. 对于枚举排序算法，原串行算法每次大循环可以将数组中的一个索引对应的值定位，这显然可以并行化处理。于是我用四个处理器对 n 次循环的任务进行划分调度，最大程度上节省了时间开销，并有了很明显的提速效果。

c. 对于归并排序算法，此次测试中并行实现的耗时比串行时间稍大，可能有以下原因：

   i. 数据量还不够大，并行算法体现不出优势；

   ii. 大部分的运行时间是在用串行实现的 merge 这一个函数中，它没有实现并行处理，所以并行体现不出优势；

   iii. 在 merge 之前，等待两个新线程完成时的同步开销比较大；

   iv. 可能存在的实现级漏洞；

五、优化思路及实验感想

1. 优化思路

考虑到目前并行实现的性能及其瓶颈，我有如下优化思路：

a. 更细化地将串行算法并行化——原先的并行快排并没有改变 partition 本身的串行属性，它的运行时间仍然是 $O(n)$，需要将 partition 也并行化才能得到效率更优的算法。可以模仿第五章讲义中算法 5-3 的思路，在 PRAM-CRCW 上为快排构造二叉树。一般来说该算法构建出的二叉树高度为 $\Theta(\log n)$，所以平均算法复杂度为 $\Theta(\log n)$，在最坏情况下算法复杂度为 $\Theta(n)$；

b. 比起创建多个线程，然后让处理器自己去分工调度，是否可以寻求一种能够更高效、有针对性的调度处理器的方法。例如，对每个线程设置一个优先级，如二叉树中更靠近根结点的任务有更高的优先级，而远离根结点的任务优先级低或串行执行。甚至可以动态地根据每个任务的计算量来灵活调度处理器，更高效地进行并行处理；

2. 实验感想

本实验让我对并行算法及其 Java 实现有了更深的认识，在巩固课堂所学的基础上，提升了程序设计的能力，是设计得比较精良的实验。对该实验，我有几点小小的建议：

a. 不规定输入集，而是要求学生自己随机生成不同规模的输入集，进行测试。然后在验收实验的时候，模仿 OJ，对不公开对测试数据进行测试，看输出结果是否符合要求；

b. 不需要强制要求实验报告中包含伪代码，因为能把程序实现出来并且通过测试，肯定知道了伪代码算法，而且把一拖伪代码排版在实验报告了里显得很冗余；

c. 我觉得不一定要规定用 Java 或者 C#，其实 C++ 等语言也可以写。有些同学不太熟悉那两门语言，还得花时间学，就比较麻烦；

以上即为本实验报告全部内容。