实验 2 高级进程间通信问题

无 16 李根 2021012787

快速排序问题

问题描述

对于有 1,000,000 个乱序数据的数据文件执行快速排序。

实验步骤

（1）首先产生包含 1,000,000 个随机数（数据类型可选整型或者浮点型）的数据文件；
（2）每次数据分割后产生两个新的进程（或线程）处理分割后的数据，每个进程（线程）处理的数据小于 1000 以后不再分割（控制产生的进程在 20 个左右）；
（3）线程（或进程）之间的通信可以选择下述机制之一进行：
1、管道（无名管道或命名管道）
2、消息队列
3、共享内存
（4）通过适当的函数调用创建上述 IPC 对象，通过调用适当的函数调用实现数据的读出与写入；
（5）需要考虑线程（或进程）间的同步；
（6）线程（或进程）运行结束，通过适当的系统调用结束线程（或进程）。

具体实现

对于大量数据的快速排序问题，我们采用多线程的方法，将大量数据分配给了若干个线程进行完成。不妨假定每个线程处理数据为 L 个，有 M 个线程，我们不难看出，经过第一轮快排后，我们得到了 N/L 个有序列表，时间复杂度为 $O(nlogn)$，接下来的操作，我们考虑在相同时间复杂的条件下，我们需要合并有序列表。具体的操作在后面描述。

首先我们先产生 1000000 个随机数，并存入数据文件中。为了方便起见，我采用了 numpy 的相关方法：

```
data = np.random.randint(0, 1000000, size=1000000)
np.savetxt( fname: 'random_numbers.txt', data, fmt='%d')

data = np.loadtxt( fname: 'random_numbers.txt', dtype=int)
```

先定义一个标准的快速排序函数：

```python
def quicksort(data):
    if len(data) <= 1:
        return data
    else:
        pivot = data[len(data) // 2]
        left = [x for x in data if x < pivot]
        middle = [x for x in data if x == pivot]
        right = [x for x in data if x > pivot]
        return quicksort(left) + middle + quicksort(right)
```

这里采用的快速排序兼顾了二分与分治策略，能够将排序的复杂度降到$O(nlogn)$

为了适应本题目的要求，我们需要将数据分割后再交给各新进程进行处理，因此我们对上面的快排进行改造：

```python
def parallel_quicksort(data, threshold=1000):
    if len(data) <= threshold:
        sorted_data = quicksort(data)
        with sorted_parts_lock:
            sorted_parts.append(sorted_data)
    else:
        pivot = data[len(data) // 2]
        left = [x for x in data if x < pivot]
        middle = [x for x in data if x == pivot]
        right = [x for x in data if x > pivot]

        task_queue.put(left)
        task_queue.put(right)

        with sorted_parts_lock:
            sorted_parts.append(middle)
```

在构建上面函数的过程中，我们用到了 sorted_parts_lock 与 task_queue 用于信息同步，其中 sorted_parts_lock 是由 threading.Lock()定义的一个锁，task_queue 是一个 Queue()。

同时定义一个线程工作的函数，用于控制线程调用函数，同时利用 task_queue()来实现了内存共享（还有一个好处是 queue 的 get_nowait()方法没有阻塞）。

```python
def worker():
    while not task_queue.empty():
        try:
            data_chunk = task_queue.get_nowait()
            parallel_quicksort(data_chunk)
        except queue.Empty:
            break
```

剩下的实际运行就很显然了：

```
task_queue.put(data)

threads = []
for _ in range(MAX_THREADS):
    thread = threading.Thread(target=worker)
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()
```

只需要先启动好 MAX_THREADS=20 个线程，再调用 join()函数即可。
此后只需要逐层对得到的有序片段进行归并即可实现多线程快速排序。

运行结果

```
D:\anaconda3\python.exe C:\Users\LiGen\Desktop\os\sort.py
完成排序，结果保存在'sorted_numbers.txt'
```

对文件输出进行观察，不妨输出前 100 个字符：

```
未排序前100个字符：
[ 65828 564516 859631 447556 185227 721735 161310 602541  91539 329388
 500673 472492 688001 303365 806742 289636 675405 846406 998765 563804
 499502 240725 194293 414674 305250 749228 221016 260072 868601 876400
 729575 831886  32562 296360 390900  57667 621810  42494 953969 463621
 935304 425833  33801 587953 581228 174481 771283 476005 638552 164030
 577193 106463  13519 978413 895116 807457 723629 177567 844345 733935
 772672 883987 603330 291030 704162  76135 551418 437768 337265 270375
 697784 322381 843356 557771 749377 153416 587363 856626 152006 717007
 840829   7214 981315 980552 591801 780799 649501 147931 634696 176549
 850767 369074 592549 246875 358262 331615 317795 770122 222806 885775]
排序后前100个字符：
[ 1  2  4  4  6  7  7  7  8  9 15 16 17 17 17 17 18 18 18 19 20 20 23 24
 25 25 25 28 30 31 34 37 38 38 40 40 41 43 44 45 46 46 46 47 48 48 48 49
 50 51 51 55 56 56 57 59 62 63 65 65 66 67 67 69 73 74 74 76 77 77 77 78
 78 78 80 82 84 84 84 84 84 86 88 89 89 89 89 91 91 92 92 94 95 96 96 96
 96 97 97 97]
```

可以看出我们的代码确实成功地完成了多线程快速排序

思考题

1．你采用了你选择的机制而不是另外的两种机制解决该问题，请解释你做出这

种选择的理由。

我使用的机制是共享内存。这是由于我们在进行多线程快排时，本质上是在进行一种分治算法，对于不同线程所处理的数据是互不干扰的，因此只需要用简单的内存共享便可以实现线程间通信而不是借用更特殊的结构(如通道或消息队列)。从性能来看，共享内存的通信方法的信息访存开销较小，这样的通信方法可以有效减少通信时间，提高快速排序效率，相对比消息队列与管道有着一定的优势。

2．你认为另外的两种机制是否同样可以解决该问题？如果可以请给出你的思路；如果不能，请解释理由。

我认为是可以的。具体理由如下：

对于通道，我们先将待排序的数字分成若干子数组，对于每个大小一定的子数组进行排序，然后每个线程通过管道将排序好的子数组发送到下一线程（或许可以定义一个线程的有序链表），然后使用较为高效的归并，同样可以完成类似的任务。

对于消息队列，类似的将算法分为若干部分，对数组的某一个段的排序作为一个任务，让所有任务依次加入队列中。多个线程依次从消息队列中取出任务进行快速排序，然后将结果发送回队列。最后主线程接收排序结果并进行归并，可以完成类似共享内存的任务

总结

通过本实验，我也对多线程分解执行算法有了更清晰的认识。快速排序作为一个非常典型的分治算法，共享内存即可以较为显然地解决线程间通信问题。在实验过程中，我认为一个很重要的点便是良好的归并方法的构建，如果只是用同一个数组去归并一切，时间复杂度甚至高达$O(n^3logn)$，程序甚至需要 5 分钟以上来完成一个简单的任务，因此需要改良。改良后的算法与单线程相比也有时间上的优势

代码如下

```
import numpy as np
import threading
import queue


data = np.random.randint(0, 1000000, size=1000000)
np.savetxt('random_numbers.txt', data, fmt='%d')

data = np.loadtxt('random_numbers.txt', dtype=int)
```

```python
# 定义普通快速排序函数
def quicksort(data):
    if len(data) <= 1:
        return data
    else:
        pivot = data[len(data) // 2]
        left = [x for x in data if x < pivot]
        middle = [x for x in data if x == pivot]
        right = [x for x in data if x > pivot]
        return quicksort(left) + middle + quicksort(right)


# 定义归并两个已排序列表的函数
def merge_sorted_lists(list1, list2):
    merged_list = []
    i, j = 0, 0

    while i < len(list1) and j < len(list2):
        if list1[i] < list2[j]:
            merged_list.append(list1[i])
            i += 1
        else:
            merged_list.append(list2[j])
            j += 1

    while i < len(list1):
        merged_list.append(list1[i])
        i += 1

    while j < len(list2):
        merged_list.append(list2[j])
        j += 1

    return merged_list


# 定义任务队列和结果列表
task_queue = queue.Queue()
sorted_parts = []
sorted_parts_lock = threading.Lock()
MAX_THREADS = 20


# 定义多线程快速排序函数
```

```
def parallel_quicksort(data, threshold=1000):
    if len(data) <= threshold:
        sorted_data = quicksort(data)
        with sorted_parts_lock:
            sorted_parts.append(sorted_data)
    else:
        pivot = data[len(data) // 2]
        left = [x for x in data if x < pivot]
        middle = [x for x in data if x == pivot]
        right = [x for x in data if x > pivot]

        task_queue.put(left)
        task_queue.put(right)

        with sorted_parts_lock:
            sorted_parts.append(middle)


def worker():
    while not task_queue.empty():
        try:
            data_chunk = task_queue.get_nowait()
            parallel_quicksort(data_chunk)
        except queue.Empty:
            break


task_queue.put(data)

threads = []
for _ in range(MAX_THREADS):
    thread = threading.Thread(target=worker)
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()


def merge_all_parts(parts):
    while len(parts) > 1:
        new_parts = []
        for i in range(0, len(parts), 2):
            if i + 1 < len(parts):
```

```
                merged = merge_sorted_lists(parts[i], parts[i +
1])
                new_parts.append(merged)
            else:
                new_parts.append(parts[i])
        parts = new_parts
    return parts[0] if parts else []


# 归并排序结果
sorted_data = merge_all_parts(sorted_parts)

# 将排序后的数据保存到文件中
np.savetxt('sorted_numbers.txt', sorted_data, fmt='%d')

print("完成排序，结果保存在'sorted_numbers.txt'")
```