并行快速排序的OMP实现

21307140069 田沐钊

在本项目中，我们选择用C++与omp库实现多线程的并行快速排序算法，对结果进行验证，并尝试多种不同的线程数和数据量，分析其加速效果。

代码实现

以下是本项目具体的代码实现，线程数和数据量的设置均在主函数中完成。

```
#include<stdio.h>
#include<omp.h>
#include <iostream>
#include<time.h>
#include<stdlib.h>
using namespace std;


int Partition(int* arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++)
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSort(int* data, int start, int end)  //并行快排
{
    if (start < end) {
        int pos = Partition(data, start, end);
        #pragma omp parallel sections    //设置并行区域
        {
            #pragma omp section          //该区域对前部分数据进行排序
            quickSort(data, start, pos - 1);
            #pragma omp section          //该区域对后部分数据进行排序
            quickSort(data, pos + 1, end);
        }
    }
}

int main(int argc, char* argv[])
{
    int n = atoi(argv[2]), i;   //线程数
```

```
    int size = atoi(argv[1]);   //数据大小
    int* num = (int*)malloc(sizeof(int) * size);

    srand(time(NULL) + rand());   //生成随机数组
    for (i = 0; i < size; i++)
        num[i] = rand();
    omp_set_num_threads(n);   //设置线程数
    double starttime = omp_get_wtime();
    quickSort(num, 0, size - 1);   //并行快排
    double endtime = omp_get_wtime();

    printf(" %d %d %lf\n", size, n, endtime - starttime);
    return 0;
}
```

下面主要对快速排序函数进行讲解。

1. **函数签名**:
   - 该函数接受一个引用类型的 `vector<int>` 数组 `arr`,以及两个整型参数 `left` 和 `right`,表示要排序的数组范围。
2. **递归条件**:
   - 函数首先检查 `left` 是否小于 `right`,如果不是,说明数组已经划分到只有一个或零个元素,无需排序,直接返回。
3. **选择基准点**:
   - 函数选择数组中间元素 `arr[(left + right) / 2]` 作为基准点 `pivot`。
4. **分区操作**:
   - 函数使用两个指针 `i` 和 `j`,分别从左右两端开始遍历数组。
   - 当 `i` 指向的元素小于基准点时,`i` 向右移动;当 `j` 指向的元素大于基准点时, `j` 向左移动。
   - 当 `i` 小于等于 `j` 时,交换 `i` 和 `j` 指向的元素,并同时移动 `i` 和 `j`。
   - 这个过程会将数组划分成两个子数组,左边的元素都小于基准点,右边的元素都大于基准点。
5. **并行递归调用**:
   - 在分区操作完成后,函数使用 `#pragma omp parallel sections` 指令,启动两个并行的工作线程。
   - 第一个线程递归地对左子数组 `[left, j]` 进行快速排序。
   - 第二个线程递归地对右子数组 `[i, right]` 进行快速排序。
   - 这样可以充分利用多核CPU,并行地对两个子数组进行排序,提高算法的执行效率。

这里计算加速比时，出于便捷考虑，直接将线程为1的运行时间作为串行时间。并且在计算运行时间时，选择每种条件运行五次程序取平均，这样得到的数据会更加可靠。

实验结果与分析

对代码进行编译即运行，排序结果均显示正确。下面分析不同数据量和线程数的加速比。

数据量为 10000

线程数	1	2	4	8	16
运行时间（ms）	1.48	0.89	1.01	0.88	1.12
加速比	1.0	1.66	1.46	1.68	1.32

数据量为 100000

线程数	1	2	4	8	16
运行时间（ms）	17.2	9.19	7.85	9.00	9.10
加速比	1.0	1.87	2.19	1.91	1.89

可以看到，在每种数据量下，随着线程数的增多，加速效果均先有显著性的提升，之后又略微减弱，我们分析原因如下：

1. 并行计算的收益递减:
    - 在并行计算中,随着加入的线程数量增加,每个线程可以分担的工作量会逐渐减少。
    - 这是因为随着线程数的增加,需要在线程之间进行更多的同步和协调,从而会增加额外的开销。
    - 当线程数过多时,这些额外开销可能会超过并行计算带来的收益,从而导致性能反而下降。
2. 资源竞争和上下文切换:
    - 每个线程都需要占用系统资源,如CPU、内存、IO等。
    - 当线程数过多时,这些资源会产生激烈的竞争,导致频繁的上下文切换,从而降低了整体性能。
    - 过多的线程会给操作系统调度带来巨大压力,增加了调度开销。
3. 缓存局部性的降低:
    - 当线程数增加时,每个线程可以独占的缓存空间会减少。
    - 这会导致缓存命中率下降,内存访问延迟增加,从而拖慢整体性能。
4. 并行开销的增加:
    - 随着线程数的增加,在线程间进行同步、通信和协调的开销也会相应增加。
    - 例如,需要更多的加锁、屏障同步等操作,这些都会带来额外的时间开销。

并且，随着数据量的增大,加速比的数值有所提升，加速效果减弱的情况也会有所缓解，我们分析原因如下：

1. 计算密集型任务:
    - 快速排序算法是一种计算密集型任务,其执行时间主要取决于CPU的计算能力。
    - 当数据量较小时,算法的执行时间较短,其中与并行化相关的开销(如线程创建、同步等)会占较大比例,从而导致加速比下降。

- 但是随着数据量的增大,算法的执行时间也会显著增加,此时算法的计算密集型特点就会更加突出,并行化带来的收益也会相对更大。

2. 并行计算效率的提高:
   - 当数据量较大时,每个子任务的计算量也会相应增大。
   - 这意味着每个工作线程可以独立完成的工作量也会增加,从而减少了线程间同步和通信的开销。
   - 相对于算法本身的计算时间,这些并行开销就会相对较小,因此加速比的下降会不那么明显。
3. 内存访问模式的优化:
   - 快速排序算法涉及大量的内存访问,当数据量较小时,内存访问模式可能不够优化。
   - 但随着数据量增大,算法会更多地利用CPU缓存,减少内存访问开销,从而提高整体的并行计算效率。

结论

1. 当线程数小于处理器数时，随着线程数的增加，加速比也增加；线程数超过处理器数时，线程之间切换的开销导致随着线程数的增加，加速比反而下降。
2. 决定并行效果（线程数与加速比之间的比值）的并不是数据量，而是可并行部分的计算量；数据量对并行效果的影响是因为其影响了可并行部分的计算量