并行计算实验二报告
——排序算法的并行及优化

PB20111701 叶升宇 **PB20111689 蓝俊玮**

说明

本此实验由 PB20111701 叶升宇完成。

实验题目

排序算法的并行及优化。

实验环境

- 操作系统：Windows 11 22H2
- IDE 及工具链：CLion + MinGW64 + LLVM + clang + Ninja
- OpenMP 库：OpenMP 201811
- 处理器： Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz 2.59 GHz

归并排序并行

原理介绍

首先针对经典排序算法，归并排序，考虑并行如下：

```
#define TASK_SIZE 100
void parallelMergeSort(int arr[], int low, int high) {
    int n = high - low + 1;
    if (n < 2) {
      return;
    }
    int mid = (low + high) / 2;

    #pragma omp task default(none) shared(arr, low, mid) if (n > TASK_SIZE)
    parallelMergeSort(arr, low, mid);

    #pragma omp task default(none) shared(arr, mid, high) if (n > TASK_SIZE)
    parallelMergeSort(arr, mid + 1, high);

    #pragma omp taskwait
    merge(arr, low, mid, high);
}

void parallelMergeSortEntry(int arr[], int low, int high) {
    #pragma omp parallel default (none) shared(arr, low, high)
    {
        #pragma omp single
        parallelMergeSort(arr, low, high);
    }
}
```

归并排序的核心思想是分治, 将排序区间划分划分成 **不相交** 的等长的两段，分别排序后再归并，使得整体有序，而针对两个子区间的排序则是 **递归** 进行的。

因此我们并行的思路就是, 针对两个不相交的子区间的排序是可以同时进行的，即对其并行。由于归并排序是递归的，因此考虑使用 `task` 结构来控制这种较为复杂的并行结构。同时，只有在两个子区间都有序后才能进行归并，因此在 `merge` 之前用 `taskwait` 结构包含一个隐式的任务调度点, 使得执行 `merge` 时已经子区间有序，来保证程序的正确性。

补充

在上面的程序中还有几处细节值的注意：

1. 在并行归并排序的入口处显示地表明只有一个任务，同时则会也是为了和测试排序程序的接口保持一致；
2. 对子区间的并行只有在其区间长度达到一定程度才进行，这里用 `TASK_SIZE` 来控制, 下面针对这一点也有分析；
3. 线程数是在单独的测试排序程序中通过 `omp_set_num_threads(THREADS_NUM);` 来设置, 具体可以看程序的源代码。

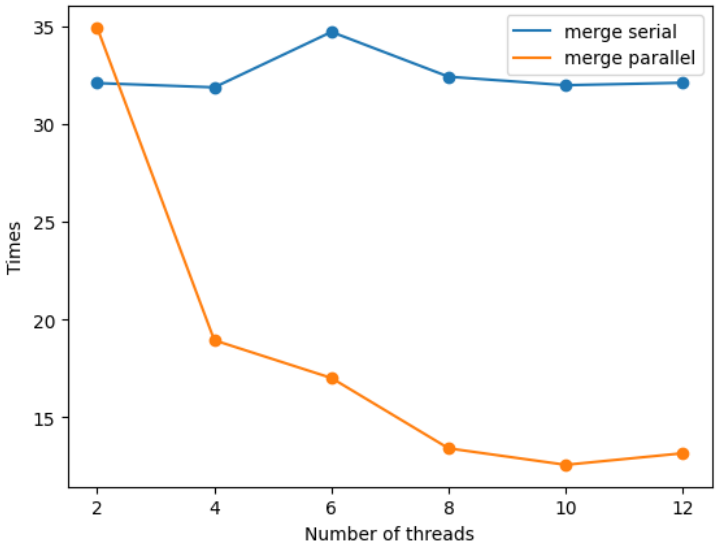
性能测试

说明

以下所有测试的数据量均为 `SIZ = 150000000`，这是因为太小的话，一次运行就几秒钟，每次测试的误差影响会很大，而太大的话，一是无法分配出过大的数组, 而是运行时间过长, 因此折中固定为目前 150000000 的大小，串行排序的运行时间为 32s 左右，比较合适。

由于电脑为 6 核 12 线程, 以下针对不同线程数进行了测试, 每次都设置串行程序对相同的数组进行排序作为对照, 结果如下：

num threads	2	4	6	8	10	12
merge serial	32.0976	31.8797	34.7177	32.4248	31.9894	32.1153
merge parallel	34.9321	18.9337	17.0003	13.4003	12.5569	13.1498
acceleration ratio	0.9189	1.6838	2.0422	2.4197	2.5476	2.4423



通过上面的结果可以发现，最好的加速比在 2.5476，这是在 8 线程的情况下，其实结果不太理想，并且两个线程的并行甚至没有取得效果, 分析后有以下几点结论：

- 由于归并排序是递归的，属于比较复杂的并行，因此如果线程太少，比如 2 个线程，确实有可能起不到效果，线程调度的开销应该不小，与并行的效果相抵消；
- 随着线程数的增加，并行效果逐步有所体现，**加速比** 逐步增加, 不过增加的幅度比较有限，并且整体上的 **并行效率** 其实不高，说明没有充分利用多线程；
- 特别地，开满 12 线程后，加速比相比 10 线程还有所下降，考虑到计算机上肯定还有其他必要的程序要运行，不可能全部线程都用来排序，因此实际上应该是没有用满的，而且还有线程调度的额外开销，因此加速比下降也属于正常现象。

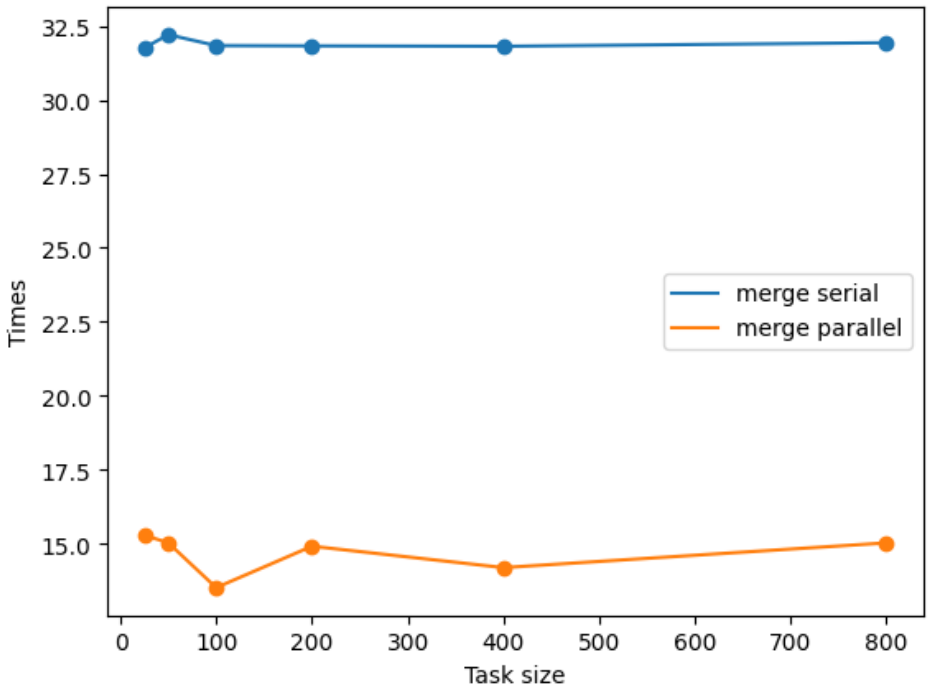
TASK_SIZE 探究

下面还针对 TASK_SIZE，即子区间达到什么程度才对其并行，进行了探究。

考虑到如果 TASK_SIZE 太小，那么对子区间排序的开销可能会和分配、调度线程的开销差不多，这样就得不偿失了，还不如直接就串行执行；而如果太大，又没有充分利用多线程并行的优势, 因此需要找一个合适的大小。

下面是对不同 TASK_SIZE 的测试结果(固定在 8 个线程上运行)：

TASK_SIZE	25	50	100	200	400	800
merge serial	31.7625	32.2121	31.8397	31.8305	31.8176	31.9376
merge parallel	15.2853	15.0268	13.5111	14.9121	14.1888	15.0250
acceleration ratio	2.0780	2.1435	2.3566	2.1345	2.2424	2.1256



可以发现，TASK_SIZE 过小和过大的加速比和中间的 TASK_SIZE 加速比相比要小一些，通过上面的结果来看，100 是比较合适的值。

快速排序并行

说明

先前只是简单地针对归并排序的子区间排序加以 task 并行，效果并不是特别好，下面参考快速排序的思路重新设计了并行排序算法，实际上也即 PSRS 算法（Parallel Sorting by Regular Sampling）。

原理介绍

快速排序的思想是，先选取主元 pivot，然后划分成小于和大于两个区间，接着分别去这两个区间递归排序。与归并排序不同的是，它是先划分再排序而不是先排序再归并。

将该思想用到并行排序上，假设待排序数组长 n，并行机上有 p 个处理器：

1. 采样：将数组均匀地划分为 p 段，去每一段都采样得到若干样本，合并为采样数组；
2. 主元：对采样数组进行 **排序**，然后选取共 p - 1 个 p-分位数，作为划分的主元；
3. 划分：由上一步的 p - 1 个主元，将数组划分为 p 段，每一段取值范围在相应两主元之间；
4. 排序：p 个处理器分别对这 p 段同时并行排序后，总数组也有序。

```
void computeSamples(const int arr[], int samples[], int pivots[]) {
    const int step = (int) (SIZE / (SAMPLE_SIZE * THREADS_NUM));
    #pragma omp parallel num_threads(THREADS_NUM) default(none) shared(arr, samples)
    {
        int tid = omp_get_thread_num();
        int offset = tid * SAMPLE_SIZE * step;
        int index = tid * SAMPLE_SIZE;
        for (int i = 0; i < SAMPLE_SIZE; ++i) {
            samples[index + i] = arr[offset + (i * step)];
        }
    }
    quickSort(samples, 0, THREADS_NUM * SAMPLE_SIZE - 1);
    for (int i = 0; i < THREADS_NUM - 1; ++i) {
        pivots[i] = samples[(i + 1) * SAMPLE_SIZE];
    }
}

void partitionData(int arr[], int pivots[], int partitions[], int start, int end) {
    for (int i = 1; i <= THREADS_NUM; ++i) {
        partitions[i] = end;
    }
    partitions[0] = start;

    int index = 1;
    for (int i = 0; i < THREADS_NUM - 1; ++i) {
        int lower = partitions[i], upper = partitions[i + 1], pivot = pivots[i];
        while (lower <= upper) {
            while (arr[lower] < pivot) {
                ++lower;
            }
            while (arr[upper] >= pivot) {
                --upper;
            }
            if (lower <= upper) {
                swap(arr[lower++], arr[upper--]);
            }
```

```
        }
        partitions[index++] = lower;
    }
}

void parallelQuickSort(int arr[], int low, int high) {
    int samples[SAMPLE_SIZE * THREADS_NUM];
    int partitions[THREADS_NUM + 1];
    int pivots[THREADS_NUM - 1];

    computeSamples(arr, samples, pivots);
    partitionData(arr, pivots, partitions, low, high);

    #pragma omp parallel default(none) shared(arr, partitions, high)
    {
        int tid = omp_get_thread_num();
        int left = partitions[tid] ;
        int right = (tid == THREADS_NUM - 1) ? high : partitions[tid + 1] - 1;
        quickSort(arr, left, right);
    }
}
```

核心的采样、划分以及并行排序算法如上所示，下面简要分析复杂度：

- 采样的大小预先指定，可以认为是常数时间，因此对采样数组的排序也是常数时间（因此在采样里的那个 parallel 实际上没什么必要，对实际结果影响可以忽略不计）；
- 划分的时间复杂度应该是 $O(p*n)=O(n)$，因为相当于进行了 p - 1 次 partition;
- 最后每个处理器排序的时间复杂度为 $O\left(\frac{n}{p}\log\frac{n}{p}\right)=O\left(\frac{n}{p}\log n\right)$

所以，实际上，整体的时间复杂度近似为 $O\left(\frac{n}{p}\log n\right)$，与串行的 $O(n\log n)$ 相比，理论上加速比应该接近 p。

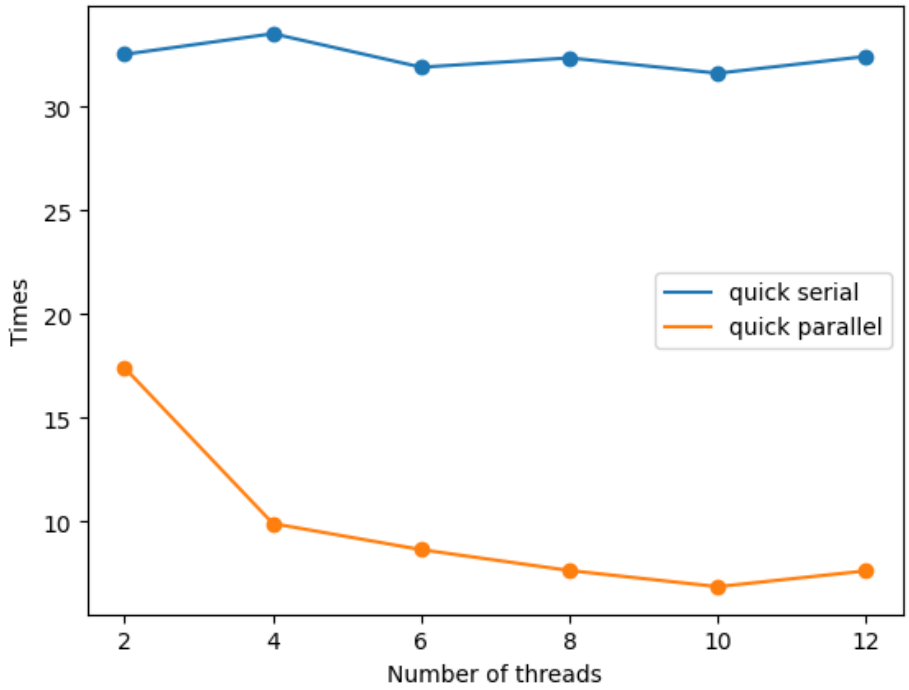
说明

实际上，快速排序的快慢取决于划分是否均匀，同样的，这里并行的排序快慢取决于那 p 个区间划分是否均匀，因为总时间是由 **最慢** 的那个线程，一般来说也就是最长的那个区间决定，因此采样的量不能过小，要保证相对划分出来的区间尽可能均匀。

性能测试

与归并排序相同，下面同样针对不同的线程进行了测试：

num threads	2	4	6	8	10	12
quick serial	32.5225	33.5187	31.9018	32.3487	31.6100	32.4221
quick parallel	17.4009	9.89134	8.6304	7.62502	6.84606	7.60903
acceleration ratio	1.8690	3.3887	3.6964	4.2424	4.6173	4.2610



从上面的结果来看，相对归并的并行有了明显的优化，最好的加速比达到了 4.6173，同时在 2/4 线程的时候加速比能达到 1.8690/3.3887，说明效率有 0.9345/0.8417，表明对线程利用相当充分！可见优化后设计的并行排序效果比之前有 **显著提升**。

说明

不过也遇到了和并行归并排序一样的问题，随着线程数增加，加速比增长还是比较有限，想真正吃满是不现实的，考虑到是 6 核 12 线程，能有这样的效果已经是非常不错了。

实验总结

本次实验并不困难，实际上也没有用到太高深的 OpenMP 的技巧或特性，关键还在与并行算法的设计与组织，以达到充分利用多线程的效果。