姓名：李其融

学号：21307110403

使用OpenMP编程实现并行快速排序算法

实验要求

1.使用使用QT或者OpenMP编程实现并行快速排序算法 2.测试程序在1K、5K、10K、100K等不同数据量以及在不同线程数情况下的加速比 3.可以使用其他语言的以及其他线程库 4.但是不可以直接调用库函数或者开源项目提供的直接可以完成并行快速排序的接口

实现思路

1. 先实现串行快速排序算法，快速排序的算法步骤如下：

   > 1．选择一个基准元素。
   >
   > 2．通过一趟排序将序列划分为两个子序列：遍历整个序列，将小于基准元素的元素放在左边，大于基准元素的元素放在右边，基准元素放在两个子序列的中间位置。这个过程称为分区（Partition）操作。
   >
   > 3．对左右两个子序列分别递归地应用快速排序算法。
   >
   > 4．重复上述步骤，直到每个子序列只包含一个元素，或为空。

2. 发现在串行算法中，左右两个子序列分别递归地应用快速排序算法，互不干扰。因此可以使用`#pragma omp task untied`，并行地完成对左右两个子列排序的任务。

3. 因为并行计算涉及线程创建、同步和销毁的开销，所以，如果子列长度很小，这些开销可能会超过并行计算带来的性能提升，从而导致并行版本比串行版本慢。因此，需要设置threshold，当子列长度小于threshold，直接串行完成排序。

实现过程

生成随机数列

根据输入的数列长度，生成随机数列`arr1`和`arr2`，这两个数列是一样的，分别用于串行算法和并行算法。

```
int len = atoi(argv[1]);
int *arr1 = new int[len];
int *arr2 = new int[len];
srand(time(0));
for (int i = 0; i < len; i++)
{
    arr1[i] = rand() % (len * 10);
    arr2[i] = arr1[i];
}
```

分别调用串行和并行算法并输出计时结果

把相应的参数传入`serial_quicksort()`和`parallel_quicksort()`并在调用函数的前后计时，最后输出排序用时。与串行不同的是，调用并行算法之前，使用`#pragma omp parallel num_threads(4)`开启4个线程，然后使用`#pragma omp single`只让一个线程调用并行快排的函数，否则会造成排序混乱。

```
// Serial
clock_t start1, end1;
start1 = clock();
serial_quicksort(arr1, 0, len - 1);
end1 = clock();
```

```
    cout << "Runtime(Serial): " << double(end1 - start1) / CLOCKS_PER_SEC << "s" << endl;

    // Parallel
    clock_t start2, end2;
#pragma omp parallel num_threads(4)
    {
#pragma omp single
        {
            start2 = clock();
            parallel_quicksort(arr2, 0, len - 1);
            end2 = clock();
        }
    }
    cout << "Runtime(Parallel): " << double(end2 - start2) / CLOCKS_PER_SEC << "s" << endl;
```

`partition()`的具体实现

对于输入的数列`arr`，选取中间的数字作为基准值`pivot`，`i`指向数列始端，`j`指向数列尾端。

满足`i <= j`时，循环执行：

1. 如果`arr[i]`小于`pivot`，则这个元素不需要移动，`i`向后移动；如果`arr[i]`大于等于`pivot`，则这个元素需要放到右边子列，`i`不移动。
2. 如果`arr[j]`大于`pivot`，则这个元素不需要移动，`j`向前移动；如果`arr[i]`小于等于`pivot`，则这个元素需要放到左边子列，`j`不移动。
3. 如果满足`i <= j`，则可以交换`arr[i]`和`arr[j]`，然后`i`和`j`分别向后向前移动

返回右边子列的始端`i`。

最后，`arr[i]`到`arr[high]`都比`pivot`大，`arr[low]`到`arr[j]`都比`pivot`小。

```
int partition(int *arr, int low, int high)
{
    int i = low;
    int j = high;
    int pivot = arr[(low + high) / 2]; // 取中间的数字作为基准值
    while (i <= j)
    {
        while (arr[i] < pivot) i++;
        while (arr[j] > pivot) j--;
        if (i <= j)
        {
            std::swap(arr[i], arr[j]);
            i++;
            j--;
        }
    }
    return i;
}
```

`serial_quicksort()`的具体实现

调用`partition()`，得到右边子列始端的下标`new_low`，则左边子列的尾端的下标为`new_high = new_low - 1`。如果子列的长度大于1（尾端大于始端），则递归调用`serial_quicksort()`处理子列。

```
void serial_quicksort(int *arr, int low, int high)
{
    int new_low = partition(arr, low, high);
    int new_high = new_low - 1;
    if (low < new_high) serial_quicksort(arr, low, new_high);
```

```
    if (new_low < high) serial_quicksort(arr, new_low, high);
}
```

`parallel_quicksort()`的具体实现

当数列的长度（`high - low`）小于`threshold`时，具体实现和`serial_quicksort()`相同。

当数列的长度（`high - low`）大于等于`threshold`时，通过`#pragma omp task untied`动态创建线程，分别递归调用`parallel_quicksort()`处理左右子列

```
void parallel_quicksort(int *arr, int low, int high, int threshold = 1000)
{
    int new_low = partition(arr, low, high);
    int new_high = new_low - 1;
    // 未排序的数列长度小于阈值时，不采用并行
    if (high - low < threshold)
    {
        if (low < new_high) parallel_quicksort(arr, low, new_high);
        if (new_low < high) parallel_quicksort(arr, new_low, high);
    }
    else
    {
#pragma omp task untied
        parallel_quicksort(arr, low, new_high);
#pragma omp task untied
        parallel_quicksort(arr, new_low, high);
    }
}
```

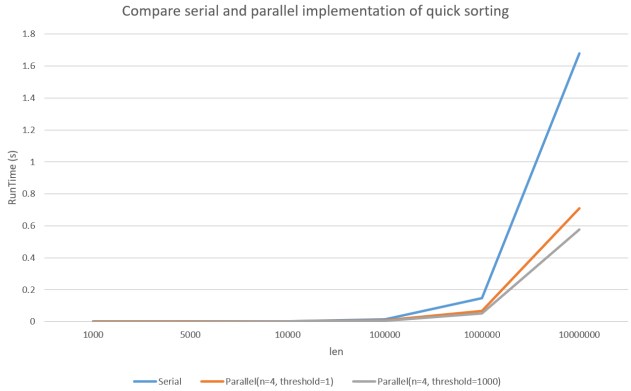
测试结果与分析

固定线程数n=4、阈值threshold=1，改变数列长度`len`：1K、5K、10K、100K、1M、10M，测量串行和并行进行快速排序各自需要的时间，每组实验各运行5次求平均值，最后计算加速比。

len	1000	5000	10000	100000	1000000	10000000
serial(s)	0.0000900842	0.00053315	0.001096992	0.01286616	0.14671	1.677978
parallel(s)	0.001086682	0.001939958	0.00215213	0.007335882	0.06690654	0.709586
speedup ratio	0.082898401	0.274825744	0.509723855	1.753866815	2.192775176	2.364728165

固定线程数n=4、阈值threshold=1000，改变数列长度`len`：1K、5K、10K、100K、1M、10M，测量串行和并行进行快速排序各自需要的时间，每组实验各运行5次求平均值，最后计算加速比。

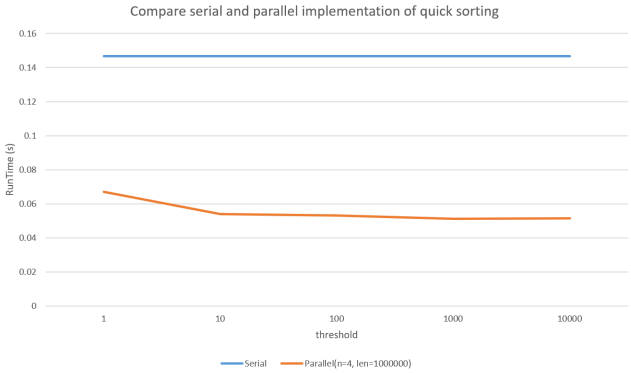
len	1000	5000	10000	100000	1000000	10000000
serial(s)	9.00842E-05	0.00053315	0.001096992	0.01286616	0.14671	1.677978
parallel(s)	0.000229248	0.00037094	0.000556655	0.004746094	0.0513382	0.5777556
speedup ratio	0.392954542	1.437295519	1.970686321	2.710894474	2.857735565	2.904304173



可以看出，数据量小，由于管理线程的开销，并行略慢于串行，但当数据量增大，并行的效果好于串行。

固定数列长度len=1000000、线程数n=4，改变阈值`threshold`，测量串行和并行进行快速排序各自需要的时间，每组实验各运行5次求平均值，最后计算加速比。

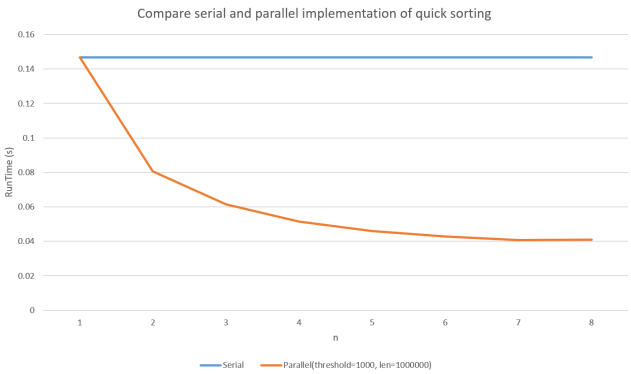
threshold	**1**	**10**	**100**	**1000**	**10000**
serial(s)	0.146711	0.146711	0.146711	0.146711	0.146711
parallel(s)	0.06690654	0.05409312	0.0530619	0.0513382	0.05156902
speedup ratio	2.192775176	2.712193344	2.764902878	2.857735565	2.844944504



可以看出，数据量较大时，调整阈值会略微影响加速比。阈值过小会减少并行的部分，过大会增加管理线程的开销，图中结果显示`threshold=1000`较为合适。

固定数列长度为len=1000000、阈值threshold=1000，改变线程数`n`，测量串行和并行进行快速排序各自需要的时间，每组实验各运行5次求平均值，最后计算加速比。

n	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
serial(s)	0.146711	0.146711	0.146711	0.146711	0.146711	0.146711	0.146711	0.146711
parallel(s)	0.1466536	0.08062488	0.06146592	0.0513382	0.04586296	0.04283022	0.04069732	0.04089794
speedup ratio	1.000391399	1.819674026	2.386867389	2.857735565	3.198899504	3.425408508	3.604930251	3.587246693



可以看出，数据量较大时，调整线程数会显著影响加速比。只有1个线程的时候，几乎和串行的时间相同；线程数增大，并行度增大，需要的时间减少，加速比增大；当线程数过大，任务粒度过细，并行开销占比大，需要的时间增多，加速比减小。

综合所有实验结果，并行算法实现快速排序比串行的方法可以快2-3倍，提高了排序的速度。