并行计算Lab2实验报告

方驰正PB21000163

一、实验目的

使用openmp编写并行程序，加速排序算法。

二、实验过程

在本次实验中，我们分别编写了串行和并行版本的冒泡排序和归并排序，并对它们分别进行了测试，计算了加速比。

1. 冒泡排序

我们分别实现了串行和并行的冒泡排序，具体代码如下：

```
for (int i = 0; i < n; i++)
#pragma omp parallel for
    for (int j = i & 1; j < n - 1; j += 2)
        if (a[j] > a[j + 1]) std::swap(a[j], a[j + 1]);
```

串行版本仅仅是去掉了 `#pragma omp parallel for` ，这里不再赘述。
这里，我们使用的方法是奇偶排序，即先对奇数位和偶数位进行排序，然后再对奇数位和偶数位进行排序，如此往复，直到排序完成。

1.1正确性证明

考虑归纳法，最大的元素经过这n轮排序后一定会在最后一位（即正确的位置），再考虑次大的元素，它最多只会在遇到最大值时停下一轮，从而它也会在第n-1位，以此类推，所有元素都会在正确的位置。

1.2加速比计算

容易得到

$$S_p = \frac{T_1}{T_p} = \frac{n^2}{n^2/p} = p$$

这里，我们分别测量了线程数不同时，在$n = 100000$的情况下的运行时间，结果如下：

线程数	时间(s)	加速比
1	6.0094	1
2	2.8782	2.0895
4	1.7205	3.4947
8	1.6855	3.5637
16	1.6567	3.6257

在这个实验中，加速比随着线程数的增加而增加，但是增加的幅度逐渐减小。这是因为线程数增加时，线程间的同步开销、通信开销以及线程管理开销逐渐增大，从而导致加速比增加的幅度逐渐减小。

2. 归并排序

我们首先实现了串行版本的归并排序，具体代码不再赘述。

然后，我们实现了并行版本的归并排序，具体代码如下：

```
int find(int l, int r, int x) {
    int res = r + 1;
    while (l <= r) {
```

```
        int mid = (l + r) >> 1;
        if (a[mid] < x)
            l = mid + 1;
        else
            r = mid - 1, res = mid;
    }
    return res;
}

void Merge(int l1, int r1, int l2, int r2, int lb) {
    // merge a[l1~r1] a[l2~r2] to b[lb~];
    if (r1 - l1 < r2 - l2) {
        std::swap(l1, l2);
        std::swap(r1, r2);
    }
    if (l1 > r1) return;
    if (l2 > r2) {
        memcpy(b + lb, a + l1, sizeof(int) * (r1 - l1 + 1));
        return;
    }

    int mid1 = (l1 + r1) >> 1;
    int mid2 = find(l2, r2, a[mid1]);

    int midb = lb + mid1 - l1 + mid2 - l2;
    b[midb] = a[mid1];
#pragma omp parallel sections
    {
#pragma omp section
        Merge(l1, mid1 - 1, l2, mid2 - 1, lb);
#pragma omp section
        Merge(mid1 + 1, r1, mid2, r2, midb + 1);
    }
}

void Sort(int l, int r) {
    if (l == r) return;
    int mid = (l + r) >> 1;
#pragma omp parallel sections
    {
#pragma omp section
        Sort(l, mid);

#pragma omp section
        Sort(mid + 1, r);
    }
    Merge(l, mid, mid + 1, r, l);
    memcpy(a + l, b + l, sizeof(int) * (r - l + 1));
}
```

这里，我们同样使用了分治的思想，将数组分成两部分，然后分别对两部分进行排序，最后再将两部分合并。

在合并的过程中，若直接使用归并排序的方法，会导致线程间的负载不均衡，因此我们使用分治的思想进行合并，即先找到两部分的中位数，然后分别递归，最后再将两部分合并。以此来保证线程间的负载均衡。

由此，我们可以得到:

$$W(n) = O(n \log n)$$
$$D(n) = O(\log^2 n)$$
$$S_{\inf} = O(\frac{n}{\log n})$$

2.1加速比计算

我们分别测量了线程数不同时，在$n = 2000000$的情况下的运行时间，结果如下：

线程数	时间(s)	加速比
1	0.734967	1
2	1.017372	0.7221
4	0.981844	0.7487
8	0.988531	0.7431
16	0.993399	0.7389

可以看到，虽然加速比随着线程数的增加而增加，但是加速比并未超过1。这是因为，并行版本的归并排序中，递归调用层数较多，导致线程创建销毁的开销较大，从而导致运行速度不如串行版本。

2.2优化

我们考虑将归并排序改为非递归版本，以此来减少线程创建销毁的开销。具体代码如下：

```
void Merge(int l1, int r1, int l2, int r2) {
    int i = l1, j = l2, k = l1;
    while (i <= r1 && j <= r2)
        if (a[i] <= a[j])
            b[k++] = a[i++];
        else
            b[k++] = a[j++];
    while (i <= r1) b[k++] = a[i++];
    while (j <= r2) b[k++] = a[j++];
    for (int i = l1; i <= r2; i++) a[i] = b[i];
}

void Sort(int l, int r) {
#pragma omp parallel for
    for (int i = l; i < r; i += 2)
        if (a[i] > a[i + 1]) std::swap(a[i], a[i + 1]);

    for (int len = 2; len <= (r - l + 1); len <<= 1) {
#pragma omp parallel for
        for (int i = l; i <= r - len; i += 2 * len)
            Merge(i, i + len - 1, i + len,
                  std::min(i + 2 * len - 1, r));
    }
}
```

这里，我们首先对相邻的两个元素进行排序，然后对长度为2的子数组进行合并，再对长度为4的子数组进行合并，以此类推，直到合并整个数组。

其理论加速比为：

$$S_p = \frac{T_1}{T_p} = \frac{n \log n}{n \log n / p} = p$$

同样的，我们分别测量了线程数不同时，在$n = 2000000$的情况下的运行时间，结果如下：

线程数	时间(s)	加速比
1	0.734967	1
2	0.647527	1.1347
4	0.623139	1.1795
8	0.703093	1.0437
16	0.653140	1.1257

可以看到，通过将归并排序改为非递归版本，我们成功提高了并行版本的运行速度，使得加速比超过了1。

三、实验总结

在本次实验中，我分别实现了串行和并行版本的冒泡排序和归并排序，并对它们进行了测试，计算了加速比。通过本次实验，我学会了如何使用openmp编写并行程序，加速排序算法。同时，也学会了如何改进自己的并行程序，提高程序的运行速度。