并行计算Lab3实验报告

方驰正PB21000163

一、实验目的

使用openmp编写并行程序，加速矩阵乘法。

二、实验过程

在这里，我们分别编写了串行版本的矩阵乘法以及不同版本的并行矩阵乘法。

1. 串行版本

```
Matrix operator*(const Matrix &_) const {
    Matrix res(n, _.m);
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= _.m; j++)
            for (int k = 1; k <= m; k++)
                res.a[i][j] += a[i][k] * _.a[k][j];
    return res;
}
```

串行版本的矩阵乘法直接使用三层循环实现。

2. 行并行

```
Matrix operator*(const Matrix &_) const {
    Matrix res(n, _.m);
#pragma omp parallel for
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= _.m; j++)
            for (int k = 1; k <= m; k++)
                res.a[i][j] += a[i][k] * _.a[k][j];
    return res;
}
```

如上所示，我们并行了最外层的循环，即对矩阵的每一行进行并行计算。

相比于并行另外两层循环，这种并行方式的效果更好，因为并行的粒度更大，减少了线程间的通信和同步的开销。

同时，如果并行第三层循环，由于每个线程都会访问 `res.a[i][j]` ，会导致线程间的竞争，效果与串行版本相差无几。

这种方法的理论加速比为p，其中p为线程数。

3.分块并行

```
Matrix operator*(const Matrix &_) const {
    Matrix res(n, _.m);
    if (n <= 10 || m <= 10) {
#pragma omp parallel for schedule(dynamic)
        for (int k = 1; k <= m; k++)
            for (int i = 1; i <= n; i++)
                for (int j = 1; j <= _.m; j++)
                    res.a[i][j] += a[i][k] * _.a[k][j];
    } else {
        int m1 = n >> 1, m2 = _.m >> 1;
```

```
        auto sa = this->split_by_ind(m1);
        auto sb = _.split_by_row(m2);

#pragma omp parallel sections
        {
#pragma omp section
            res.copy(sa.first * sb.first, 1, 1);
#pragma omp section
            res.copy(sa.first * sb.second, 1, m2 + 1);
#pragma omp section
            res.copy(sa.second * sb.first, m1 + 1, 1);
#pragma omp section
            res.copy(sa.second * sb.second, m1 + 1, m2 + 1);
        }
    }
    return res;
}
```

如上所示，我们将矩阵A按行分为A_1和A_2，将矩阵B按列分为B_1和B_2，然后分别计算A_1B_1、A_1B_2、A_2B_1、A_2B_2，最后合并得到结果。即:

$$A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}, B = \begin{bmatrix} B_1 & B_2 \end{bmatrix}$$

$$C = AB = \begin{bmatrix} A_1B_1 & A_1B_2 \\ A_2B_1 & A_2B_2 \end{bmatrix}$$

则

$$W(n,m) = 4W(n,m/2) + O(nm), W(n,n) = O(n^3)$$

$$D(n,m) = D(n,m/2) + O(nm), D(n,n) = O(n^2)$$

因此理论最大加速比为n。

但是，这种方法多次对矩阵进行拆分和合并，都涉及到了内存的拷贝，将会造成较大的开销。因此，我们修改了这种方法，将值传递改为只传递下标。具体如下所示:

4.只传递下标的分块并行

```
struct SubMat {
    int sx, sy, ex, ey;
    unsigned int **a;
};

void Mult(SubMat a, SubMat b, SubMat c) {
    // a * b into c;
    if (a.ex - a.sx + 1 <= 10) {
        for (int k = a.sy; k <= a.ey; k++)
            for (int i = a.sx; i <= a.ex; i++) {
                int x = i - a.sx + c.sx;
                for (int j = b.sy; j <= b.ey; j++) {
                    int y = j - b.sy + c.sy;
                    c.a[x][y] += a.a[i][k] * b.a[k][j];
                }
            }
        return;
    }

    // split a,b into 2 mats.

    int axm = (a.sx + a.ex) >> 1;
    int bym = (b.sy + b.ey) >> 1;
    SubMat a1(a.sx, a.sy, axm, a.ey, a.a);
    SubMat a2(axm + 1, a.sy, a.ex, a.ey, a.a);
```

```
    SubMat b1(b.sx, b.sy, b.ex, bym, b.a);
    SubMat b2(b.sx, bym + 1, b.ex, b.ey, b.a);

    SubMat c11(c.sx, c.sy, c.sx + axm - a.sx,
               c.sy + bym - b.sy, c.a);
    SubMat c12(c.sx, c.sy + bym - b.sy + 1,
               c.sx + axm - a.sx, c.ey, c.a);
    SubMat c21(c.sx + axm - a.sx + 1, c.sy, c.ex,
               c.sy + bym - b.sy, c.a);
    SubMat c22(c.sx + axm - a.sx + 1, c.sy + bym - b.sy + 1,
               c.ex, c.ey, c.a);

#pragma omp parallel sections
    {
#pragma omp section
        Mult(a1, b1, c11);
#pragma omp section
        Mult(a1, b2, c12);
#pragma omp section
        Mult(a2, b1, c21);
#pragma omp section
        Mult(a2, b2, c22);
    }
}
```

如上所示，我们只传递了下标，而不是传递矩阵，减少了内存的拷贝，从而大大减小了时间开销。可以在加速比计算部分中看到这种方法的效果。

5.加速比计算

我们分别对以上三种方法进行了一次2000×2000的矩阵乘法，并记录了时间。如下所示：

5.1 行并行

线程数	时间(s)	加速比
1	47.081813	1
2	25.631217	1.84
4	12.011230	3.92
8	5.983183	7.87
16	4.622329	10.19

5.2 分块并行

线程数	时间(s)	加速比
1	47.081813	1
2	17.977252	2.62
4	9.317965	5.05
8	9.508511	4.95
16	9.532316	4.94

5.3 只传递下标的分块并行

线程数	时间(s)	加速比
1	47.081813	1
2	1.689669	27.85
4	0.858368	54.88
8	0.848744	55.47
16	0.897218	52.52

可以看到，只传递下标的分块并行方法的加速比最高，达到了55.47，远远高于其他两种方法。

三、实验总结

在这次实验中，我们学习了openmp的使用方法，并且实现了三种不同的并行矩阵乘法方法。通过实验，我们发现只传递下标的分块并行方法的效果最好，加速比最高，达到了55.47。这种方法减少了内存的拷贝，减小了时间开销，是一种非常有效的并行方法。