中山大学数据科学与计算机院本科生实验报告

（2020 学年秋季学期）

课程名称：**高性能计算程序设计**　　任课教师：**黄聃**　　**批改人：**

年级+班级	2018 级五班	专业（方向）	计算机科学
学号	18340126	姓名	罗仁良
Email	luorliang@mail2.sysu.edu.cn	完成日期	2020.9.20

1. 实验目的(20 分)

1) 利用 C/C++语言实现通用矩阵乘法，输出矩阵和矩阵的计算时间。
2) 基于算法分析实现对矩阵乘法的优化，采用 Strassen 算法作为实验详细分析的例子，详细分析比较计算时间以及优化效果。
3) 大规模矩阵计算的优化方式（稀疏矩阵乘法优化，大规模矩阵防止内存溢出）。

2. 实验过程和核心代码(40 分)

1) 通用矩阵乘法

- 数据结构

采用 Matrix 类对矩阵相关的数据结构进行封装；

```
class Matrix{
public:

    int ** data;            //数据
    int row;                //行
    int col;                //列

    Matrix();

    Matrix(int row, int col);

    Matrix(const Matrix& A);

// 随机数生成矩阵数值
    void ProduceMat(const unsigned int & seed);

// 拷贝原矩阵的部分（子矩阵 b）到矩阵 B 中
```

```
        void copy(const Matrix &B, const Submartrix &b);

        Matrix operator + (const Matrix &A);

        Matrix operator - (const Matrix &A);

        Matrix operator *(const Matrix& A);

        Matrix& operator=(const Matrix& A);
    // 打印矩阵
        void printMatrix();
    // 资源回收
        ~Matrix();
};
```

- 通用矩阵乘法算法比较简单，直接采用运算符重载实现。

```
    Matrix Matrix::operator*(const Matrix &A)
    {
        //两个矩阵不可相乘，返回空矩阵；
        if (col != A.row)
            return Matrix(0, 0);

        Matrix C(row, A.col);
        for (int i = 0; i < row; i++)
            for (int j = 0; j < A.col; j++)
            {
                C.data[i][j] = 0;
                for (int k = 0; k < col; k++)
                    C.data[i][j] += data[i][k] * A.data[k][j];
            }
        return C;
}
```

2) Strassen 算法

Strassen 算法是一种分而治之的思想，将矩阵分成几个小的矩阵，分别对小矩阵进行运算处理。若采用简单的分治递归，最终的复杂的还是$O(n^3)$，还会带来函数递归调用的额外开销，导致运算效率下降。

先分析简单的分治算法计算$C = A \cdot B$的过程：

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

$$\mathrm{B}=\begin{pmatrix}B_{11} & B_{12}\\ B_{21} & B_{22}\end{pmatrix}$$

$$\mathrm{C}=\begin{pmatrix}C_{11} & C_{12}\\ C_{21} & C_{22}\end{pmatrix}$$

根据分块矩阵乘法，

$$\mathrm{C}_{11}=A_{11}\cdot B_{11}+A_{12}\cdot B_{21}$$
$$\mathrm{C}_{12}=A_{11}\cdot B_{12}+A_{12}\cdot B_{22}$$
$$\mathrm{C}_{21}=A_{21}\cdot B_{11}+A_{22}\cdot B_{21}$$
$$\mathrm{C}_{22}=A_{21}\cdot B_{12}+A_{22}\cdot B_{22}$$

设算法计算的时间为$\mathrm{T(n)}$，由上可得递推公式，$\mathrm{T(n)}=8\cdot T\left(\frac{n}{2}\right)+4$ （8 次乘法，4 次加法），算法复杂度仍然为$\mathrm{O(n^3)}$。Strassen 算法的优化之处在于使用额外的加法运算，复杂为$\mathrm{O(n^2)}$，减少一次乘法运算，最终使得时间复杂度为$\mathrm{O(n^{\log_2 7})}$ 。

仍然采取如上的矩阵划分形式，构造如下等式

$$\mathrm{S}_1=A_{11}\cdot(B_{12}-B_{22})$$
$$\mathrm{S}_2=(\mathrm{A}_{11}+\mathrm{A}_{12})\cdot B_{22}$$
$$\mathrm{S}_3=(\mathrm{A}_{21}+\mathrm{A}_{22})\cdot \mathrm{B}_{11}$$
$$\mathrm{S}_4=\mathrm{A}_{22}\cdot(B_{21}-B_{11})$$
$$\mathrm{S}_5=(A_{11}+A_{22})\cdot(B_{11}+B_{22})$$
$$\mathrm{S}_6=(\mathrm{A}_{12}-\mathrm{A}_{22})\cdot(B_{21}+B_{22})$$
$$\mathrm{S}_7=(A_{11}-A_{21})\cdot(B_{11}+B_{12})$$

由此可以计算出，

$$\mathrm{C}_{11}=S_5+S_4-S_2+S_6$$
$$\mathrm{C}_{12}=S_1+S_2$$
$$\mathrm{C}_{21}=\mathrm{S}_3+\mathrm{S}_4$$
$$\mathrm{C}_{22}=\mathrm{S}_5+\mathrm{S}_1-\mathrm{S}_3-\mathrm{S}_7$$

则可得递推公式，$\mathrm{T(n)}=7\cdot\mathrm{T}\left(\frac{\mathrm{n}}{2}\right)+\mathrm{O(n^2)}$。

注：假设计算的矩阵规模都为$\mathbf{n\times n}$

- **实现代码**

Strassen 算法实现过程中有许多子矩阵的加减运算，为减少创建新矩阵的额外开销，实现了直接在访问原矩阵的数据进行加减运算。

数据结构 Submatrix 用于定位子矩阵在原矩阵中的区域

```
struct Submartrix //方阵
{       //子矩阵首元素在父矩阵中的位置
        int row;
        int col;
    //矩阵大小
        int size;
        Submartrix(int row, int col, int size) : row(row), col(col), size(size) {}
};
```

子矩阵运算

```
Matrix sub_sum(const Matrix &A, const Matrix &B,const  Submartrix &a, const Submartrix &b) // 子矩阵加法

Matrix sub_minus(const Matrix &A, const Matrix &B,const  Submartrix &a, const Submartrix &b) // 子矩阵减法

Matrix sub_mat(const Matrix &A, const Submartrix &a) // 子矩阵拷贝
```

- 实现核心

```
Matrix strassen(const Matrix &A, const Matrix &B, const Submartrix &a,const Submartrix &b)
{
    // 划分门槛，防止矩阵划分过小
    if (a.size <= 64)
    {
        Matrix C(a.size,a.size);
        for (int i = 0; i < a.size; i++)
        {
            for (int j = 0; j < a.size; j++)
            {
                int sum = 0;
                for (int k = 0; k < a.size; k++)
                {
                    sum += A.data[a.row + i][a.col + k] * B.data[b.row + k][b.col + j];
                }
                C.data[i][j] = sum;
            }
        }
        return C;
    }
// 划分子矩阵
    int sub_size = a.size / 2;
```

```
        Submartrix a11(a.row, a.col, sub_size);
        Submartrix a12(a.row, a.col + sub_size, sub_size);
        Submartrix a21(a.row + sub_size, a.col, sub_size);
        Submartrix a22(a.row + sub_size, a.col + sub_size, sub_size);
        Submartrix b11(b.row, b.col, sub_size);
        Submartrix b12(b.row, b.col + sub_size, sub_size);
        Submartrix b21(b.row + sub_size, b.col, sub_size);
        Submartrix b22(b.row + sub_size, b.col + sub_size, sub_size);
        Matrix S1(sub_size,sub_size), S2(sub_size,sub_size), S3(sub_size,sub_size),
S4(sub_size,sub_size), S5(sub_size,sub_size);
        Matrix S6(sub_size,sub_size), S7(sub_size,sub_size);
        Submartrix D(0,0,sub_size);
    //计算构造式子
        S1 = strassen(sub_mat(A,a11),sub_minus(B,B,b12,b22),D,D);
        S2 = strassen(sub_sum(A,A,a11,a12),sub_mat(B,b22),D,D);
        S3 = strassen(sub_sum(A,A,a21,a22),sub_mat(B,b11),D,D);
        S4 = strassen(sub_mat(A,a22),sub_minus(B,B,b21,b11),D,D);
        S5 = strassen(sub_sum(A,A,a11,a22),sub_sum(B,B,b11,b22),D,D);
        S6 = strassen(sub_minus(A,A,a12,a22),sub_sum(B,B,b21,b22),D,D);
        S7 = strassen(sub_minus(A,A,a11,a21),sub_sum(B,B,b11,b12),D,D);
        Matrix C11(sub_size,sub_size), C12(sub_size,sub_size),
C21(sub_size,sub_size), C22(sub_size,sub_size);
        C11 = S5 + S4 - S2 + S6;
        C12 = S1 + S2;
        C21 = S3 + S4;
        C22 = S5 + S1 - S3 - S7;
        Matrix C(sub_size * 2,sub_size * 2);
    // 将子矩阵合并
        C11.copy(C, Submartrix(0, 0, sub_size));
        C12.copy(C, Submartrix(0, sub_size, sub_size));
        C21.copy(C, Submartrix(sub_size, 0, sub_size));
        C22.copy(C, Submartrix(sub_size, sub_size, sub_size));
        return C;
}
```

- 基于软件的优化方式

基于软件的优化方式的核心思想是人工控制计算的顺序，减少 CPU 的访存操作从而加快运算的速度。

循环拆分的过程中，选择循环过程中会反复访问的数据作为内层循环计算，使得数据可一直接保存在寄存器或者缓存中，加快访存的速度。

内存重排的最终目的是利用缓存块，合理分布数据空间，减少 cache miss 发生的次数，加快访存的速度。一般情况下采用二维数组存储的矩阵在内存的物理空间上是矩阵行连续存储的，结合告诉缓存行存取的特性，将矩阵做个“转置”存储，访问时运算发生的 cache miss 次数更少，可以提高运算速度。

- 大规模矩阵乘法

矩阵压缩（稀疏矩阵存储）

1) 三元组方式（i，j，value） 只存储非零元素的位置和值

2) CSR 存储（行压缩）

可靠性保证

在有限的内存空间计算大规模矩阵可能出现内存溢出的情况。为防止这种情况的出现，可以根据实际的内存情况，将矩阵进行分块运算，在内存不足的情况下，没参与计算的子矩阵放置在外存中，需要参与计算时，重新载入内存。

3. 实验结果(30 分)

矩阵规模	通用算法（单位：ms）	Strassen 算法(单位:ms)
512	705	463
1024	9135	3266
2048	89837	23001

注：只输出了矩阵结果的一部分和时间

```
input m, n , k >> 1024 1024 1024
normal C:
20277 16498 23717 20817 16402 22991 21538 16312 22265 22276
24383 20283 16480 23651 20823 16390 22925 21544 16300 22196
16572 24411 20327 16476 23679 20863 16386 22953 21584 16290
19739 16566 24381 20279 16470 23655 20815 16380 22929 21536
25230 19745 16548 24315 20285 16458 23589 20821 16368 22860
16640 25258 19789 16544 24343 20325 16454 23617 20861 16358
19227 16680 25314 19767 16584 24395 20303 16494 23669 20830
26182 19233 16662 25248 19773 16572 24329 20309 16482 23600
16901 26210 19277 16658 25276 19813 16568 24357 20349 16472
18715 16941 26266 19255 16698 25328 19791 16608 24409 20318
strassen C:
20277 16498 23717 20817 16402 22991 21538 16312 22265 22276
24383 20283 16480 23651 20823 16390 22925 21544 16300 22196
16572 24411 20327 16476 23679 20863 16386 22953 21584 16290
19739 16566 24381 20279 16470 23655 20815 16380 22929 21536
25230 19745 16548 24315 20285 16458 23589 20821 16368 22860
16640 25258 19789 16544 24343 20325 16454 23617 20861 16358
19227 16680 25314 19767 16584 24395 20303 16494 23669 20830
26182 19233 16662 25248 19773 16572 24329 20309 16482 23600
16901 26210 19277 16658 25276 19813 16568 24357 20349 16472
18715 16941 26266 19255 16698 25328 19791 16608 24409 20318
time : 9135.000000 ms
time : 3266.000000 ms
```

从表中结果可以看到，尽管 Strasseen 仍是$O(n^{\log_2 7})$，当n足够大时，优化的效果逐渐明显。

4. 实验感想(10 分)

一般矩阵乘法运算的时间复杂度是较高的$O(n^3)$，经过 Strassen 算法优化之后的$O(n^{\log_2 7})$的时间复杂度仍然比较高。想要进一步通过算法优化是苦难的，所以我们转换角度，从减少访存时间合理管理数据在内存中的布局，从而提高运算速度（提高 CPU 利用率)。在单核处理器（运算核心较少）的情况下，矩阵规模一旦较大，运算时间就会大幅增加。想要进一步提高计算的性能，需要硬件上的提升，多处理器资源等，实现并行处理，从而提高计算速度。

//可以写写过程中遇到的问题，你是怎么解决的。以及可以写你对此次实验的一些理解……