实验报告

Strassen矩阵乘法核心算法实现

核心思想

Strassen算法优化矩阵乘法利用了分治的思想，将2个2^n阶方阵分成8个2^{n-1}阶矩阵进行运算

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

之后通过8个子矩阵加、减、乘等运算计算出7个中间矩阵

$$\begin{aligned} M_1 &= A_{11} \times (B_{12} - B_{22}) \\ M_2 &= (A_{11} + A_{12}) \times B_{22} \\ M_3 &= (A_{21} + A_{22}) \times B_{11} \\ M_4 &= A_{22} \times (B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ M_6 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) \\ M_7 &= (A_{11} - A_{21}) \times (B_{11} + B_{12}) \end{aligned}$$

之后通过7个中间矩阵计算出结果矩阵的4个子矩阵

$$\begin{aligned} C_{11} &= M_5 + M_4 - M_2 + M_6 \\ C_{12} &= M_1 + M_2 \\ C_{21} &= M_3 + M_4 \\ C_{22} &= M_1 + M_5 + M_3 + M_7 \end{aligned}$$

总共算法时间复杂度递推式为

$$T(n) = 7T(\frac{n}{2}) + O(n^2)$$

根据主定理可得算法总时间复杂度为$O(n^{log_2 7})$。算法主要思想是将分块乘法的需要8次乘法，通过加减法的组合减少到7次。所付出的代价是算法运行时间的常数较大，在娇小的数据规模中表现得不如朴素的矩阵乘法

代码实现

```
auto A11 = makeUp(A, size, 0, 0);
auto A12 = makeUp(A, size, 0, mid);
auto A21 = makeUp(A, size, mid, 0);
auto A22 = makeUp(A, size, mid, mid);
auto B11 = makeUp(B, size, 0, 0);
auto B12 = makeUp(B, size, 0, mid);
auto B21 = makeUp(B, size, mid, 0);
auto B22 = makeUp(B, size, mid, mid);

strassenMul(M1, A11, sub(B12, B22), depth + 1);
strassenMul(M2, add(A11, A12), B22, depth + 1);
```

```
strassenMul(M3, add(A21, A22), B11, depth + 1);
strassenMul(M4, A22, sub(B21, B11), depth + 1);
strassenMul(M5, add(A11, A22), add(B11, B22), depth + 1);
strassenMul(M6, sub(A12, A22), add(B21, B22), depth + 1);
strassenMul(M7, sub(A11, A21), add(B11, B12), depth + 1);

auto C11 = add(add(M5, M4), sub(M6, M2));
auto C12 = add(M1, M2);
auto C21 = add(M3, M4);
auto C22 = sub(add(M5, M1), add(M3, M7));

merge(C, C11, 0, 0);
merge(C, C12, 0, mid);
merge(C, C21, mid, 0);
merge(C, C22, mid, mid);
```

优化思路

优化1

优化描述

Strassen算法优化矩阵乘法只适用于阶数为2^n的方阵，局限性较大。在计算一般矩阵乘法时，我们需要找到不小于矩阵边长最大值的2的指数，将两个矩阵缺失的地方进行补零。

代码实现

```
int len = std::max(n, std::max(m, p));
        for (int cur = 1; ; cur *= 2)
            if (len <= cur) {
                size = cur;
                break;
            }
```

优化2

优化描述

我们注意到Strassen算法常数较大，在矩阵阶数较小时我们选择朴素矩阵乘法直接计算而非递归继续计算，我们设置矩阵阶数为32而非2作为递归基。

代码实现

```
if (size <= MIN) {
    C = mul(A, B, size, size, size);
    return;
}
```

优化3

优化描述

我们注意到当矩阵阶数较大时，矩阵中补零的数量较多，导致大量无效计算，我们对每一个矩阵打一个标记，当tag为true的时候代表矩阵为全零矩阵，当矩阵乘法中有一个为全零矩阵时，直接返回全零矩阵。

代码实现

```
if (A.selfCheck() | B.selfCheck()) {
    C = Matrix(size);
    return;
}
```

优化4

优化描述

我们发现Strassen算法需要多次动态分配内存并释放，我们将二维数组压缩为一维数组并重载()运算符简化数组索引，以减少程序向堆中申请内存与释放的次数。

代码实现

```
struct Matrix {
    int *matrix = nullptr;
    int size;

    inline explicit Matrix(int _size) : size(_size) {
        matrix = new int[size * size];
        std::fill(matrix, matrix + size * size, 0);
    }

    inline ~Matrix() {
        delete[] matrix;
    }

    inline Matrix &operator=(const Matrix &M) {
        size = M.size;
        matrix = new int[size * size];
        memcpy(matrix, M.matrix, size * size * sizeof(int));
        return *this;
    }

    inline int &operator()(int x, int y) {
        return matrix[x * size + y];
    }

    inline const int &operator()(int x, int y) const {
        return matrix[x * size + y];
    }
```

```
    [[nodiscard]] inline bool selfCheck() const {
        bool flag = true;
        for (int i = 0; i < size * size; i++)
            flag &= !matrix[i];
        return flag;
    }
};
```

优化5

优化描述

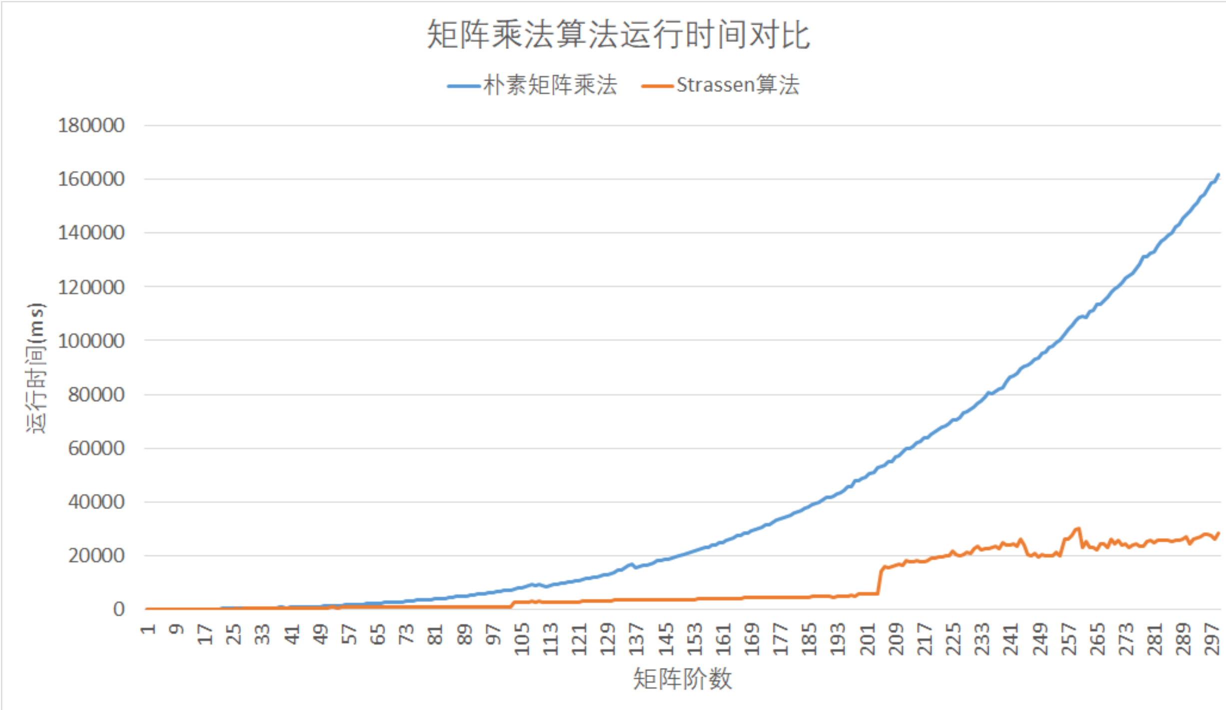
我们与朴素的矩阵乘法对比，发现Strassen算法也可以利用并发编程进行优化。同时因为Strassen为递归实现，导致子线程的数量过多，我们限制子线程生成的个数，即限制递归深度来选择是否生成子线程。

代码实现

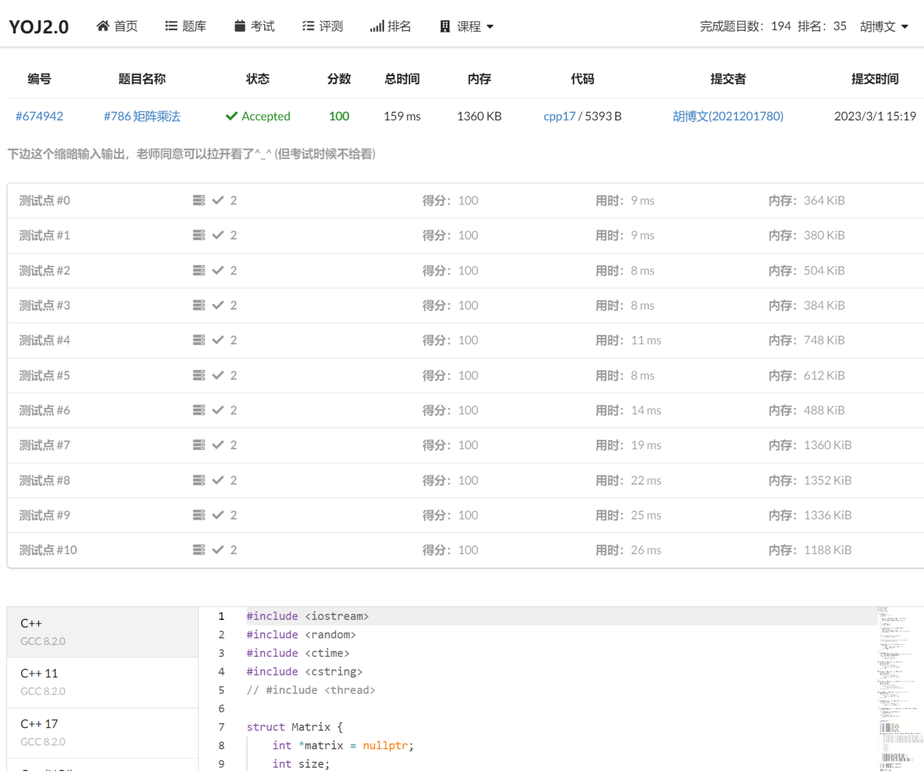
```
if (depth <= 2) {
    std::thread t1([&]() { strassenMul(M1, A11, sub(B12, B22), depth + 1); });
    std::thread t2([&]() { strassenMul(M2, add(A11, A12), B22, depth + 1); });
    std::thread t3([&]() { strassenMul(M3, add(A21, A22), B11, depth + 1); });
    std::thread t4([&]() { strassenMul(M4, A22, sub(B21, B11), depth + 1); });
    std::thread t5([&]() { strassenMul(M5, add(A11, A22), add(B11, B22), depth + 1);
});
    std::thread t6([&]() { strassenMul(M6, sub(A12, A22), add(B21, B22), depth + 1);
});
    std::thread t7([&]() { strassenMul(M7, sub(A11, A21), add(B11, B12), depth + 1);
});

    t1.join();
    t2.join();
    t3.join();
    t4.join();
    t5.join();
    t6.join();
    t7.join();
}
```

运行时间比较

我们通过测试不同算法下不同阶数下矩阵乘法运行时间，绘制以下折线图：发现在矩阵阶数较小时，因为Strassen算法的常数较大，与朴素矩阵乘法的差距较小；当矩阵阶数提升时，二者差距显著。



YOJ通过截图



YOJ2.0 首页 题库 考试 评测 排名 课程 完成题目数：194 排名：35 胡博文

编号	题目名称	状态	分数	总时间	内存	代码	提交者	提交时间
#674942	#786 矩阵乘法	Accepted	100	159 ms	1360 KB	cpp17 / 5393 B	胡博文(2021201780)	2023/3/1 15:19

下边这个缩略输入输出，老师同意可以拉开看了^_^ (但考试时候不给看)

测试点 #0	✓ 2	得分：100	用时：9 ms	内存：364 KiB
测试点 #1	✓ 2	得分：100	用时：9 ms	内存：380 KiB
测试点 #2	✓ 2	得分：100	用时：8 ms	内存：504 KiB
测试点 #3	✓ 2	得分：100	用时：8 ms	内存：384 KiB
测试点 #4	✓ 2	得分：100	用时：11 ms	内存：748 KiB
测试点 #5	✓ 2	得分：100	用时：8 ms	内存：612 KiB
测试点 #6	✓ 2	得分：100	用时：14 ms	内存：488 KiB
测试点 #7	✓ 2	得分：100	用时：19 ms	内存：1360 KiB
测试点 #8	✓ 2	得分：100	用时：22 ms	内存：1352 KiB
测试点 #9	✓ 2	得分：100	用时：25 ms	内存：1336 KiB
测试点 #10	✓ 2	得分：100	用时：26 ms	内存：1188 KiB

C++ GCC 8.2.0
C++ 11 GCC 8.2.0
C++ 17 GCC 8.2.0

```
#include <iostream>
#include <random>
#include <ctime>
#include <cstring>
// #include <thread>

struct Matrix {
    int *matrix = nullptr;
    int size;
```