

**本科实验报告**

**课程名称： C语言程序设计A**

**实验名称： 用C语言实现Strassen矩阵乘法算法**

**姓名： 刘芷辰**

**学院： 计算机科学与技术学院**

**系别： 计算机科学与技术**

**专业： 计算机**

**年级： 大一**

**学号： PB21111728**

**任课教师： 吴锋**

**2021年12月11日**

* **1.实验目的：**

（1）了解Strassen矩阵算法的实现过程

（2）用C语言实现Strassen算法和传统矩阵乘法

* **2.实验内容：**

（1）编程实现Strassen矩阵相乘算法

（2）测试不同矩阵规模时Strassen算法的性能

（3）比较Strassen算法和传统的0（）算法的实际计算性能

* **3. 算法：**

（1） 传统矩阵乘法:

AB=C A,B,C均为n阶方阵

（因为Strassen算法只适用于2的n次幂阶数的方阵，为了比较，这里的传统算法也只进行方阵的相乘）

C第i行第j列的元素的值等于A的第i行元素与B的第j列元素分别相乘并相加

用伪代码表示为

for(k=0;k<n;k++)

C[i][j]=C[i][j]+A[i][k]\*B[k][j]

（2）Strassen算法：

AB=C

A,B,C均为n阶方阵(n为2的次幂)

Strassen算法采用分治策略，即将矩阵都划分为四个的矩阵，即

A= B= C=

那么有：

=

=

=

=

引入新矩阵：

M1=

M2=

M3=

M4=

M5=

M6=

M7=

便得到

因此采用递归，可以将2的n阶次幂的矩阵逐步转化为2阶的矩阵乘法

由于需要测试的阶数较大，为了便于输入，另写一随机生成矩阵的程序便于从文件读入数据

* **4.程序清单：**

/\*传统矩阵相乘源程序\*/

#include <stdio.h>

#include <string.h>

#include<stdlib.h>

#include <time.h>

clock\_t start, end; //计时

#define buildmatrix(a,n,m)\

int \*\*a=(int\*\*)malloc(sizeof(int \*)\*n);\

for(int t=0;t<n;t++)\

{\

a[t]=(int \*)malloc(sizeof(int)\*m);\

memset(a[t],0.0,sizeof(\*a[t]));\

}

#define N 32 //确定阶数，编译前修改

typedef int count; //计数器

count i, j, k;

//定义输入函数

void input(int n,int \*\*p)

{

FILE \*fp;

fp=fopen("putout.txt","r");

if(fp==NULL)

{

printf("FILE CANNOT OPEN!");

exit(0);

}

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

fscanf(fp,"%d",&p[i][j]);

}

}

fclose(fp);

}

//定义输出函数

void output(int n,int \*\*p)

{

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

printf("%4d ", p[i][j]);

}

printf("\n");

}

}

//定义矩阵相乘函数

void matrix\_multiply(int n,int \*\*X,int \*\*Y,int \*\*Z)

{

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

Z[i][j] = 0;

for (k = 0; k < n; k++)

{

Z[i][j] = Z[i][j] + X[i][k] \* Y[k][j];

}

}

}

}

int main()

{

buildmatrix(A,N,N);

buildmatrix(B,N,N);

buildmatrix(C,N,N);

input(N, A);

printf("\n");

input(N, B);

start = clock();

matrix\_multiply(N, A, B, C);

end = clock();

output(N, C);

printf("time=%lf\n", (double)(end - start) / CLK\_TCK);

return 0;

}

/\*strassen矩阵乘法源程序\*/

#include <string.h>

#include<stdlib.h>

#include <time.h>

#include <stdio.h>

#define N 32

clock\_t start, end; //计时器

//定义宏，来实现对多个二维数组动态空间的开辟，同时命名

#define buildmatrix(a,n,m)\

int \*\*a=(int\*\*)malloc(sizeof(int \*)\*n);\

for(int t=0;t<n;t++)\

{\

a[t]=(int \*)malloc(sizeof(int)\*m);\

memset(a[t],0.0,sizeof(\*a[t]));\

}

//确定矩阵规模，编译前修改

#define limN 32 //确定采用传统算法的最大阶数，编译前修改

typedef int count; //计数器

count i, j, k;

//定义输入函数

void input(int n, int \*\*p)

{

//从文件中调用矩阵

FILE \*fp;

fp=fopen("putout.txt","r");

if(fp==NULL)

{

printf("FILE CANNOT OPEN!");

exit(0);

}

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

fscanf(fp,"%d",&p[i][j]);

}

}

fclose(fp);

}

//定义输出函数

void output(int n, int \*\*p)

{

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

printf("%4d ", p[i][j]);

}

printf("\n");

}

}

//定义矩阵相加函数

void matrix\_add(int n, int \*\*X, int \*\*Y, int \*\*Z)

{

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

Z[i][j] = X[i][j] + Y[i][j];

}

}

}

//定义矩阵相减函数

void matrix\_sub(int n, int \*\*X, int \*\*Y, int \*\*Z)

{

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

Z[i][j] = X[i][j]- Y[i][j];

}

}

}

//定义传统矩阵相乘函数

void matrix\_multiply(int \*\*A, int \*\*B, int \*\*C)

{

for (i = 0; i < limN ; i++)

{

for (j = 0; j < limN ; j++)

{

C[i][j] = 0;

for (k = 0; k < limN; k++)

{

C[i][j] = C[i][j] + A[i][k] \* B[k][j];

}

}

}

}

//定义strassen算法函数

void strassen(int n, int \*\*A, int \*\*B, int \*\*C)

{

//在矩阵规模小于一定阶数的时候，采用传统的矩阵乘法

if (n <= limN)

{

matrix\_multiply(A, B, C);

return;

}

//二维数组的命名以及动态空间的开辟

buildmatrix(A11,n/2,n/2);

buildmatrix(A12,n/2,n/2);

buildmatrix(A21,n/2,n/2);

buildmatrix(A22,n/2,n/2);

buildmatrix(B11,n/2,n/2);

buildmatrix(B12,n/2,n/2);

buildmatrix(B21,n/2,n/2);

buildmatrix(B22,n/2,n/2);

buildmatrix(C11,n/2,n/2);

buildmatrix(C12,n/2,n/2);

buildmatrix(C21,n/2,n/2);

buildmatrix(C22,n/2,n/2);

buildmatrix(M1,n/2,n/2);

buildmatrix(M2,n/2,n/2);

buildmatrix(M3,n/2,n/2);

buildmatrix(M4,n/2,n/2);

buildmatrix(M5,n/2,n/2);

buildmatrix(M6,n/2,n/2);

buildmatrix(M7,n/2,n/2);

buildmatrix(AA,n/2,n/2);

buildmatrix(BB,n/2,n/2);

buildmatrix(MM1,n/2,n/2);

buildmatrix(MM2,n/2,n/2);

//分治策略

for (i = 0; i < n / 2; i++)

{

for (j = 0; j < n / 2; j++)

{

A11[i][j] = A[i][j];

A12[i][j] = A[i][j + n / 2];

A21[i][j] = A[i + n / 2][j];

A22[i][j] = A[i + n / 2][j + n / 2];

B11[i][j] = B[i][j];

B12[i][j] = B[i][j + n / 2];

B21[i][j] = B[i + n / 2][j];

B22[i][j] = B[i + n / 2][j + n / 2];

}

}

//递归

matrix\_sub(n / 2, B12, B22, BB);

strassen(n / 2, A11, BB, M1);

matrix\_add(n / 2, A11, A12, AA);

strassen(n / 2, AA, B22, M2);

matrix\_add(n / 2, A21, A22, AA);

strassen(n / 2, AA, B11, M3);

matrix\_sub(n / 2, B21, B11, BB);

strassen(n / 2, A22, BB, M4);

matrix\_add(n / 2, A11, A22, AA);

matrix\_add(n / 2, B11, B22, BB);

strassen(n / 2, AA, BB, M5);

matrix\_sub(n / 2, A12, A22, AA);

matrix\_add(n / 2, B21, B22, BB);

strassen(n / 2, AA, BB, M6);

matrix\_sub(n / 2, A11, A21, AA);

matrix\_add(n / 2, B11, B12, BB);

strassen(n / 2, AA, BB, M7);

matrix\_add(n / 2, M5, M4, MM1);

matrix\_sub(n / 2, M2, M6, MM2);

matrix\_sub(n / 2, MM1, MM2, C11);

matrix\_add(n / 2, M1, M2, C12);

matrix\_add(n / 2, M3, M4, C21);

matrix\_add(n / 2, M5, M1, MM1);

matrix\_add(n / 2, M3, M7, MM2);

matrix\_sub(n / 2, MM1, MM2, C22);

for (i = 0; i < n / 2; i++)

{

for (j = 0; j < n / 2; j++)

{

C[i][j] = C11[i][j];

C[i][j + n / 2] = C12[i][j];

C[i + n / 2][j] = C21[i][j];

C[i + n / 2][j + n / 2] = C22[i][j];

}

}

//释放malloc的空间

for(i = 0; i < n/2; i++)

{

free(A11[i]);

free(B11[i]); free(C11[i]); free(M1[i]); free(M5[i]);

free(A12[i]); free(B12[i]); free(C12[i]);

free(M2[i]); free(M6[i]);

free(A21[i]); free(B21[i]); free(C21[i]);

free(M3[i]); free(M7[i]);

free(A22[i]); free(B22[i]); free(C22[i]); free(M4[i]);

free(AA[i]); free(BB[i]); free(MM1[i]); free(MM2[i]);

}

free(A11); free(B11); free(C11); free(M1); free(M5);

free(A12); free(B12); free(C12); free(M2); free(M6);

free(A21); free(B21); free(C21); free(M3); free(M7);

free(A22); free(B22); free(C22); free(M4);

free(AA); free(BB); free(MM1); free(MM2);

}

int main()

{

buildmatrix(A,N,N);

buildmatrix(B,N,N);

buildmatrix(C,N,N);

input(N, A);

input(N, B);

start = clock();

strassen(N, A, B, C);

end = clock();

output(N, C);

printf("time=%lf\n", (double)(end - start) / CLK\_TCK);

return 0;

}

/\*生成随机矩阵源程序\*/

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define N 200000

typedef int count;

count i, j;

int main()

{

FILE \*fp;

fp = fopen("putout.txt", "w");

if (fp == NULL)

{

printf("FILE CANNOT OPEN!");

exit(0);

}

int n;

int matrix[N];

scanf("%d", &n);

srand((unsigned)time(NULL));

for (i = 0; i < n \* n; i++)

{

matrix[i] = (int)(rand() % 10);

}

for (i = 0; i < n \* n; i++)

{

fprintf(fp, "%d ", matrix[i]);

if((i+1)%n==0)

{

fprintf(fp,"\n");

}

}

fclose(fp);

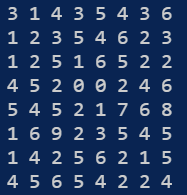
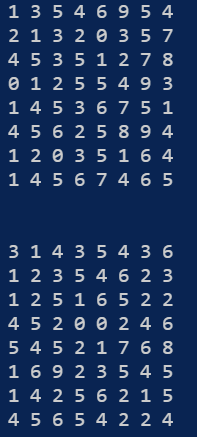
return 0;

}

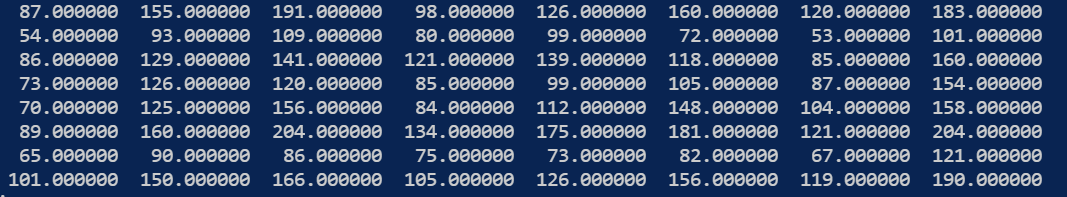
* **5.运行结果：**

***测试strassen算法正确性：***

8阶：输入数据：



输出数据（在初代double时所测）：



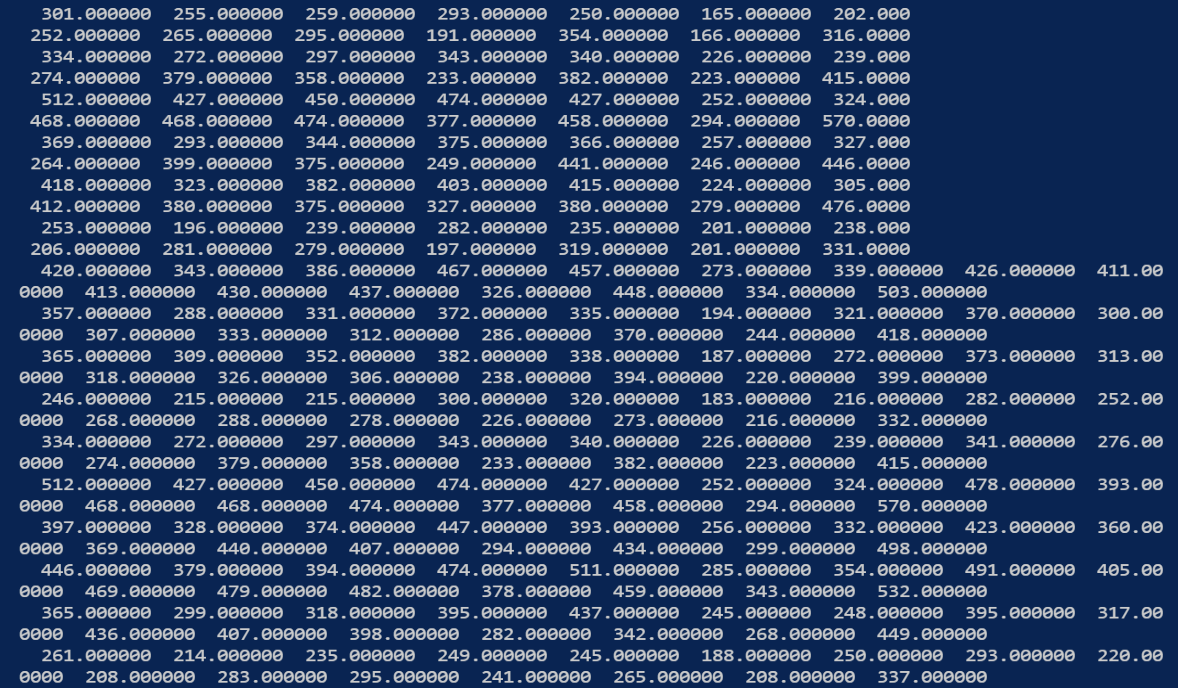
✔

16阶：

输入数据：



输出数据（在初代double时所测）：



✔

更高阶的由于输出数据过多，不方便展示，但经检验，均为正确

***测试两种算法的性能***

***（忽略输出所测,选取合适的limN进行测试，比如比测试阶数低两个2的次幂）：***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 阶数 | 传统算法 | | strassen算法 | |
| 三次测量值（单位/秒） | 平均值（单位/秒） | 三次测量值（单位/秒） | 平均值（单位/秒） |
| 8 | <0.001(小于clock最小精度) | <0.001(小于clock最小精度) | <0.001(小于clock最小精度) | <0.001(小于clock最小精度) |
| 16 | <0.001(小于clock最小精度) | <0.001(小于clock最小精度) | <0.001(小于clock最小精度) | <0.001(小于clock最小精度) |
| 32 | <0.001(小于clock最小精度) | <0.001(小于clock最小精度) | <0.001(小于clock最小精度) | <0.001(小于clock最小精度) |
| 64 | 0.001 | 0.001 | 0.002 | 0.001666667 |
| 0.001 | 0.001 |
| 0.001 | 0.002 |
| 128 | 0.009 | 0.009333333 | 0.012 | 0.011666667 |
| 0.01 | 0.011 |
| 0.009 | 0.012 |
| 256 | 0.084 | 0.080333333 | 0.027 | 0.026333333 |
| 0.08 | 0.026 |
| 0.077 | 0.026 |
| 512 | 0.671 | 0.682 | 0.413 | 0.416666667 |
| 0.671 | 0.412 |
| 0.704 | 0.425 |
| 1024 | 7.113 | 7.771 | 2.368 | 2.359333333 |
| 7.384 | 2.346 |
| 8.816 | 2.364 |

折线图比较（横轴表示阶数，纵轴表示时间）：

柱形图比较（横轴表示时间，纵轴表示阶数）：

***结果说明与分析：***

（1）从图表中可以看出，随阶数增加，两种算法所需时间都在增加，但在阶数较低时，strassen算法并未展现出优越性，而随着阶数的增加，strassen算法的优越性逐渐明显起来

（2）大概是从64阶开始，strassen算法性能开始超越传统算法

（3）经查资料，发现strassen算法的时间复杂度是O(),小于传统算法的O(),是由于strassen算法比传统算法少一次乘法，虽然加法增多，但在高阶时，乘法更少带来的时间节省就非常大

* **6.调试分析与体会**

（1）在有多层循环时，需要注意是否将某些量重新初始化

（2）对事先不能确定范围的值，可以采用宏定义的方式，在编译前进行修改

（3） 为了使某些变量的意义更明确，可以采用typedef的方式，如将int typedef为count，再用count去定义，可以明确看出变量是用来计数的，可读性更高

（4）定义函数时参数不能设同一个，在后续使用中会按照函数定义中的参数来执行

（5）往后进行的实验由于测试数据较大，手动输入过于繁琐，所以需要写一个自动生成矩阵的程序，从而得到进一步练习

（6）在最开始的程序中，在超过64阶时出现爆栈的情况，于是采用malloc进行动态空间分配，在堆上开辟空间，从而避免爆栈的情况

（7）由于利用malloc开辟二维数组空间需要的代码较多，在大量开辟时十分麻烦，所以采用宏定义的方式，在定义的同时开辟动态空间，大大节省了代码量，但在此过程中，遇到段错误，经查是因为memset的方式随着malloc的开辟，也从初始化二维数组变到初始化指针，因此需要就修改为初始化数据

（8）虽然开辟了动态空间，但是由于大量的递归开辟，导致运行时间过长，甚至出现死机的情况，于是将一次递归过后的空间free掉，同时将double改为int，加快了运行速度

（9）虽然速度有提升，但还是与理论值偏差过大，考虑到在低阶是strassen算法没有优越性，反而会因为大量地递归和调用函数导致运行时间长得令人难以接受，于是设置一个limN，编译前进行修改，数据是在合适的limN下测量的，低于limN的阶数采用传统的矩阵乘法进行计算，高阶的用strassen递归至limN再采用传统算法，从而大大加快了运行速度，符合预期

（10）通过本次实验，学会了使用计时函数，文件操作，分治策略等，并了解了Strassen算法与传统算法的性能优劣