并行与分布式作业

“作业名字”

第 2 次作业

姓名：TRY

班级：18 级计科 7 班

学号：

一、 问题描述

1. 问题 1：

分别采用不同的算法（非分布式算法）例如一般算法、分治算法和 Strassen 算法等计算计算矩阵两个 300x300 的矩阵乘积，并通过 Perf 工具分别观察 cache miss、CPI、mem-loads 等性能指标。

2. 问题 2：理论分析题

3. Consider a memory system with a level 1 cache of 32 KB and DRAM of 512 MB with the processor operating at 1 GHz. The latency to L1 cache is one cycle and the latency to DRAM is 100 cycles. In each memory cycle, the processor fetches four words (cache line size is four words). What is the peak achievable performance of a dot product of two vectors? Note: Where necessary, assume an optimal cache placement policy.

```
/* dot product loop */
for (i = 0; i < dim; i++)
        dot_prod += a[i] * b[i];
```

3. 问题 3：理论分析

4. Now consider the problem of multiplying a dense matrix with a vector using a two-loop dot-product formulation. The matrix is of dimension 4*K* x 4*K*. (Each row of the matrix takes 16 KB of storage.) What is the peak achievable performance of this technique using a two- loop dot-product based matrix-vector product?

```
/* matrix-vector product loop */
for (i = 0; i < dim; i++)
        for (j = 0; i < dim; j++)
                c[i] += a[i][j] * b[j];
```

二、 解决方案

1. 问题 1：

本题要利用不同的算法计算 300*300 的矩阵乘积。然而，在经过演算分析后发现，如果矩阵 n×n 的大小 n 不是 2^n 的话，无法正常进行有效的分治或 Strassen。所以在此题中，笔者将题中的 n=300 改成了 256=2^8 去进行观察。

其次，在设计本题中，笔者通过网上资料学习了解了计算矩阵乘积的 3 中方法：一般算法、分治算法、Strassen 算法。

（1）一般法

根据笔者在线性代数中学习的知识，两个矩阵的乘法仅当第一个矩阵 B 的列数和另一个矩阵 A 的行数相等时才能定义。如 A 是 m×n 矩阵和 B 是 n×p 矩阵，它们的乘积 AB 是一个 m×p 矩阵，它的一个元素其中 1 ≤ i ≤ m，1 ≤ j ≤ p。（代码过于简单，此处省略）

Matrix multiplication. Given two n-by-n matrices A and B, compute C = AB.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

Brute force. $\Theta(n^3)$ arithmetic operations.

（2）分治法

C=A * B ，假设三个矩阵均为 n×n，n 为 2 的幂。可以对其分解为 4 个 n/2×n/2 的子矩阵分别递归求解：

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \\ C_{12} &= (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \\ C_{21} &= (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \\ C_{22} &= (A_{21} \times B_{12}) + (A_{22} \times B_{22}) \end{aligned}$$

$$T(n) = \underbrace{8T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, form submatrices}} \Rightarrow T(n) = \Theta(n^3)$$

然而，在按照此思路编写分治代码的时候，发现如果将矩阵一直分治到大小为 1 再返回的话，时间会大大多于一般算法的运行时间！！根据笔者的研究发现，当 n 较小时，分治递归的开销会大于分治算法的运算优势，导致总运行时间大于一般循环算法的时间。所以，笔者设定了一个界限。当 n<界限时，使用普通法计算矩阵，而不继续分治递归。改进后算法的计算优势明显，时间大幅下降（从 327s 降到了 20s 左右）。之后，笔者还针对不同大小的界限进行试验，并发现最优的界限值在 32～128 之间（此处笔者采用的是 64）。同理，下面的 Strassen 算法也进行了同样的处理，也在小于某个下界的时候运用一般循环算法进行计算。

在分治算法中，笔者利用了传下标的方式，减小了矩阵的多次复制赋值的开销。同时，我利用了 vector< vector<int> >的方式储存矩阵，同时定义了结构体 matrix 来进行储存。相对来说时间开销会较运用 int**的时间大。

下面是，分治算法中一些重点函数的代码：

```
//r,c表示行、列起始位置，rn,cn为行、列终止位置。本函数主要进行矩阵+运算。
void SetSubMatrix(int r, int c, int rn, int cn, const matrix &A, const matrix &B)
{
    for (int i = c; i < cn; ++i)
    {
        for (int j = r; j < rn; j++)
        {
            data[i][j] = A.data[i - c][j - r] + B.data[i - c][j - r];
        }
    }
}

//ar,ac是A矩阵左上角元素的行与列，n表示该矩阵的大小
static matrix SquareMultiplyRecursive(matrix &A, matrix &B, int ar, int ac, int br, int bc, int n)
{
    matrix C(n, n);

    if (n <= 64)
        C = matrix::common_matrix_product(A, B, ar, ac, br, bc, n);
    else
    {
        C.SetSubMatrix(0, 0, n / 2, n / 2,
            SquareMultiplyRecursive(A, B, ar, ac, br, bc, n / 2),
            SquareMultiplyRecursive(A, B, ar, ac + (n / 2), br + (n / 2), bc, n / 2));

        C.SetSubMatrix(0, n / 2, n / 2, n,
            SquareMultiplyRecursive(A, B, ar, ac, br, bc + (n / 2), n / 2),
            SquareMultiplyRecursive(A, B, ar, ac + (n / 2), br + (n / 2), bc + (n / 2), n / 2));

        C.SetSubMatrix(n / 2, 0, n, n / 2,
            SquareMultiplyRecursive(A, B, ar + n / 2, ac, br, bc, n / 2),
            SquareMultiplyRecursive(A, B, ar + (n / 2), ac + (n / 2), br + (n / 2), bc, n / 2));

        C.SetSubMatrix(n / 2, n / 2, n, n,
            SquareMultiplyRecursive(A, B, ar + (n / 2), ac, br, bc + (n / 2), n / 2),
            SquareMultiplyRecursive(A, B, ar + (n / 2), ac + (n / 2), br + (n / 2), bc + (n / 2), n / 2));

    }
    return C;
}
```

（3）Strassen 算法：

由上面的分析可以知道，分治算法仍然存在了 8 次乘法运算，故虽然进行了分治，其时间复杂度仍然与一般循环算法的时间复杂度一样，都是$O(n^3)$。随着历史的发展，数学家们也尝试通过减少矩阵乘法中乘法运算的次数来降低时间复杂度。终于，在 1969 年德国数学家 Strassen 将乘法的计算次数降低到了 7 次，时间复杂度也降到了$O(n^{2.81})$。

Key idea. multiply 2-by-2 block matrices with only 7 multiplications.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} P_1 &= A_{11} \times (B_{12} - B_{22}) \\ P_2 &= (A_{11} + A_{12}) \times B_{22} \\ P_3 &= (A_{21} + A_{22}) \times B_{11} \\ P_4 &= A_{22} \times (B_{21} - B_{11}) \\ P_5 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ P_6 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) \\ P_7 &= (A_{11} - A_{21}) \times (B_{11} + B_{12}) \end{aligned}$$

$$\begin{aligned} C_{11} &= P_5 + P_4 - P_2 + P_6 \\ C_{12} &= P_1 + P_2 \\ C_{21} &= P_3 + P_4 \\ C_{22} &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

因此，在编写代码的时候，笔者按照此思路计算矩阵的乘法。同样，针对时间开销的问题，引入了一个下界，当 n<下界时，运用一般循环算法来进行计算，有效降低了运行时间。同时，在 Strassen 中，我利用了类的思想进行处理，并且利用 int**来对矩阵进行储存。时间较上面的 vector 要少。

下面是 Strassen 算法中的重要函数：

```
class Strassen_class {
public:
    void ADD(T** MatrixA, T** MatrixB, T** MatrixResult, int MatrixSize);//矩阵相加
    void SUB(T** MatrixA, T** MatrixB, T** MatrixResult, int MatrixSize);//矩阵相减
    void MUL(T** MatrixA, T** MatrixB, T** MatrixResult, int MatrixSize);//一般算法实现
    void FillMatrix(T** MatrixA, T** MatrixB, int length);//A,B矩阵赋值
    void PrintMatrix(T **MatrixA, int MatrixSize);//打印矩阵
    void Strassen(int N, T **MatrixA, T **MatrixB, T **MatrixC);//Strassen算法实现
};


void Strassen_class<T>::Strassen(int N, T **MatrixA, T **MatrixB, T **MatrixC)
{

    int HalfSize = N / 2;
    int newSize = N / 2;

    if (N <= 64)    //分治门槛，小于这个值时不再递归，而是采用普通矩阵乘法
    {
        MUL(MatrixA, MatrixB, MatrixC, N);
    }
```

```
//现在计算M1~M7矩阵
//M1[][]
ADD(A11, A22, AResult, HalfSize);
ADD(B11, B22, BResult, HalfSize);                //p5=(a+d)*(e+h)
Strassen(HalfSize, AResult, BResult, M1); //now that we need to multiply this , we use the strassen itself .
//M2[][]
ADD(A21, A22, AResult, HalfSize);              //M2=(A21+A22)B11    p3=(c+d)*e
Strassen(HalfSize, AResult, B11, M2);       //Mul(AResult,B11,M2);
//M3[][]
SUB(B12, B22, BResult, HalfSize);              //M3=A11(B12-B22)    p1=a*(f-h)
Strassen(HalfSize, A11, BResult, M3);       //Mul(A11,BResult,M3);
//M4[][]
SUB(B21, B11, BResult, HalfSize);           //M4=A22(B21-B11)     p4=d*(g-e)
Strassen(HalfSize, A22, BResult, M4);       //Mul(A22,BResult,M4);
//M5[][]
ADD(A11, A12, AResult, HalfSize);           //M5=(A11+A12)B22    p2=(a+b)*h
Strassen(HalfSize, AResult, B22, M5);       //Mul(AResult,B22,M5);
//M6[][]
SUB(A21, A11, AResult, HalfSize);
ADD(B11, B12, BResult, HalfSize);             //M6=(A21-A11)(B11+B12)    p7=(c-a)(e+f)
Strassen(HalfSize, AResult, BResult, M6);    //Mul(AResult,BResult,M6);
//M7[][]
SUB(A12, A22, AResult, HalfSize);
ADD(B21, B22, BResult, HalfSize);             //M7=(A12-A22)(B21+B22)     p6=(b-d)*(g+h)
Strassen(HalfSize, AResult, BResult, M7);     //Mul(AResult,BResult,M7);
```

```
//C11 = M1 + M4 - M5 + M7;
ADD(M1, M4, AResult, HalfSize);
SUB(M7, M5, BResult, HalfSize);
ADD(AResult, BResult, C11, HalfSize);

//C12 = M3 + M5;
ADD(M3, M5, C12, HalfSize);

//C21 = M2 + M4;
ADD(M2, M4, C21, HalfSize);

//C22 = M1 + M3 - M2 + M6;
ADD(M1, M3, AResult, HalfSize);
SUB(M6, M2, BResult, HalfSize);
ADD(AResult, BResult, C22, HalfSize);
```

2. 问题 2

```
/* dot product loop */
for (i = 0; i < dim; i++)
        dot_prod += a[i] * b[i];
```

由题目可知，cache line=4 words,而在 32 位系统中，1 word=4 bytes，所以 1 cache line 可以储存 4 个 int。而 cache 大小为 32KB,可算得共有 2K 行。而在第一次循环 cache miss 后，会从内存 DRAM 中装入 a[i]~a[i+3]的 int，和 b[i]~b[i+3]的 int，之后 3 次循环都会 hit。而且，根据 PPT 中“latency”的定义，DRAM latency 已经包含了从 DRAM 到 CPU 处理器的时间，也就是包括了从 DRAM 取到 cache 再到 CPU 的时间。所以总时间是：第 1 次循环中从 DRAM 中取数的延迟+第 2、3、4 次循环中从 cache 中取数的延迟=2*100 + 3*2*1=206 cycles。而以 4 次循环为一个单位，每 4 次循环中进行了 4 次乘法操作和 4 次加法操作，所以一共有 8 次浮点操作。所以 peak achievable performance of a dot product of two vectors 是 $8 \div \left(206 \times 10^{-9}\right) = 38.8 MFLOPS$.

同理，如果是在 64 位系统中，1 word=8 bytes，所以 1 cache line 可以储存 8 个 int。所以将 8 个循环作为一个单位，则总时间是第 1 次循环中从 DRAM 中取数的延迟+第 2~8 次循环中从 cache 中取数的延迟=2*100+7*2=214 cycles。而以 8 个循环作为一个单位，每 8 次循环中进行了 8 次乘法操作和 8 次加法操作，所以一共有 16 次浮点操作。所以 peak achievable performance of a dot product of two vectors 是

$16 \div \left(214 \times 10^{-9}\right) = 74.77 MFLOPS$.

3. 问题 3

4. Now consider the problem of multiplying a dense matrix with a vector using a two-loop dot-product formulation. The matrix is of dimension 4*K* x 4*K*. (Each row of the matrix takes 16 KB of storage.) What is the peak achievable performance of this technique using a two- loop dot-product based matrix-vector product?

```
/* matrix-vector product loop */
for (i = 0; i < dim; i++)
        for (j = 0; i < dim; j++)
                c[i] += a[i][j] * b[j];
```

在本题中，计算的是矩阵与向量相乘。在 32 位系统中，由于此处给定矩阵的大小是 4K*4K，可以发现，在内存循环的 4K 次中，用的都是同一个 c[i]，所以 c[i]的延迟可以忽略不计。而由上题可知，cache 一共有 2K 行。故在第一次外层循环中，4K 次内层循环执行完后，cache 中有一半的 cache 行储存着 a[0][0]~a[0][4K-1]，另一半的 cache 行储存着 b[0]~b[4K-1]。（因为 1K 行可以储存 1K*4 words=4K 个 int，刚好等于 b 向量的大小）所以在之后的外层循环中，b 向量已经储存在 cache 中，且不会被替换出去。只是 a 矩阵的不同行被替换进 cache 中。所以，第一次外层循环 b 向量从 DRAM 中造成的取数延迟，在 4K 次外层循环的执行中，同样可忽略不计。所以，只需统计 a 矩阵从 DRAM 取数中造成的延迟和从 cache 中取数造成的延迟。

因此，可以将 4 次内层循环看成一个单位，总时间是：第 1 次循环中 a 矩阵从 DRAM 中取数造成的延迟+第 2、3、4 次循环中 a 矩阵从 cache 中取数造成的延迟+第 1、2、3、4 次循环中 b 向量从 cache 中取数造成的延迟=100+3+4=107 cycles。而 4 次内层循环中，共有 4 次乘法操作和 4 次加法操作，共 8 次浮点数操作。所以 peak achievable performance of this technique using a two-loop dot-product based matrix-vector product 是

$8 \div \left(107 \times 10^{-9}\right) = 74.77 MFLOPS$.

同理，如果在 64 位系统中，cache 中有 1K 行。因此，将 8 次内层循环看成一个单位，总时间是：第 1 次循环中 a 矩阵从 DRAM 中取数造成的延迟+第 2~8 次循环中 a 矩阵从 cache 中取数造成的延迟+第 1~8 次循环中 b 向量从 cache 中取数造成的延迟=100+7+8=115 cycles。而 8 次内层循环中，共有 8 次乘法操作和 8 次加法操作，共 16 次浮点数操作。所以 peak achievable performance of this technique using a two-loop dot-product based matrix-vector product 是 $16 \div \left(115 \times 10^{-9}\right) = 139.13 MFLOPS$.

三、 实验结果

1. 问题 1

*注：由于分治算法和 Strassen 算法分别利用了 vector 和 int**去储存矩阵，故两种方法都编写了相应的一般循环算法去进行比较。*

（1）一般算法与分治算法的比较

下面是分治算法的 CPI(即下图中的 instructions)、cache_misses 和 mem-stores.

```
shelly@shelly-virtual-machine:~$ sudo perf stat ./c.out
[sudo] shelly 的密码:
分治算法:0.191921s

 Performance counter stats for './c.out':

        194.861147      task-clock (msec)         #    0.991 CPUs utilized
                16      context-switches          #    0.082 K/sec
                 0      cpu-migrations            #    0.000 K/sec
               405      page-faults               #    0.002 M/sec
       734,945,588      cycles                    #    3.772 GHz
     2,189,699,709      instructions              #    2.98  insn per cycle
       244,423,729      branches                  # 1254.348 M/sec
           297,307      branch-misses             #    0.12% of all branches

       0.196542765 seconds time elapsed
```

```
shelly@shelly-virtual-machine:~$ sudo perf stat -e cache-misses,mem-stores ./c.out
分治算法:0.19111s

 Performance counter stats for './c.out':

           192,176      cache-misses
       448,942,468      mem-stores

       0.195848882 seconds time elapsed
```

下面是一般循环算法的 CPI(即下图中的 instructions)、cache_misses 和 mem-stores.

```
shelly@shelly-virtual-machine:~$ gedit c.c
shelly@shelly-virtual-machine:~$ g++ c.c -o c.out
shelly@shelly-virtual-machine:~$ ./c.out
一般算法:0.200733s
shelly@shelly-virtual-machine:~$ sudo perf stat ./c.out
一般算法:0.190037s

 Performance counter stats for './c.out':

        192.914356      task-clock (msec)         #    0.991 CPUs utilized
                18      context-switches          #    0.093 K/sec
                 0      cpu-migrations            #    0.000 K/sec
               306      page-faults               #    0.002 M/sec
       726,601,614      cycles                    #    3.766 GHz
     2,149,875,475      instructions              #    2.96  insn per cycle
       238,559,751      branches                  # 1236.610 M/sec
            90,940      branch-misses             #    0.04% of all branches

       0.194627011 seconds time elapsed
```

```
shelly@shelly-virtual-machine:~$ sudo perf stat -e cache-misses,mem-stores ./c.out
一般算法:0.191037s

 Performance counter stats for './c.out':

           193,012      cache-misses
       439,682,349      mem-stores

       0.195842601 seconds time elapsed
```

经过以上的比较可知，两个方法的运行时间和 CPI 都相近，这也与理论上推得的“时间复杂度相同”相符合。而一般算法的 cache-misses 要略大于分治算法的 cache-misses，mem-stores 指标则略小于。

（2）一般算法和 strassen 算法的比较：

下面是 Strasssen 算法的 CPI(即下图中的 instructions)、cache_misses 和 mem-stores.

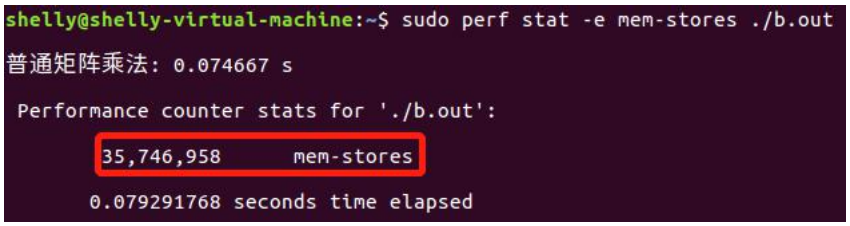
```
shelly@shelly-virtual-machine:~$ sudo perf stat ./b.out

Strassen算法:0.057119 s

 Performance counter stats for './b.out':

         61.556287      task-clock (msec)         #    0.975 CPUs utilized
                11      context-switches          #    0.179 K/sec
                 0      cpu-migrations            #    0.000 K/sec
             1,105      page-faults               #    0.018 M/sec
       226,511,997      cycles                    #    3.680 GHz
       729,517,424      instructions              #    3.22  insn per cycle
        32,521,136      branches                  #  528.315 M/sec
            37,737      branch-misses             #    0.12% of all branches

       0.063122395 seconds time elapsed

shelly@shelly-virtual-machine:~$ sudo perf stat -e cache-misses ./b.out

Strassen算法:0.056252 s

 Performance counter stats for './b.out':

           347,512      cache-misses

       0.059872165 seconds time elapsed
```

```
shelly@shelly-virtual-machine:~$ sudo perf stat -e mem-stores ./b.out

Strassen算法:0.057626 s

 Performance counter stats for './b.out':

        32,122,347      mem-stores

       0.061735338 seconds time elapsed
```

下面是一般循环算法的CPI(即下图中的instructions)、cache_misses和 mem-stores.

```
普通矩阵乘法: 0.070989 s

 Performance counter stats for './b.out':

         74.125306      task-clock (msec)         #    0.971 CPUs utilized
                 9      context-switches          #    0.121 K/sec
                 0      cpu-migrations            #    0.000 K/sec
               302      page-faults               #    0.004 M/sec
       285,253,461      cycles                    #    3.848 GHz
       871,005,179      instructions              #    3.05  insn per cycle
        36,522,671      branches                  #  492.715 M/sec
            88,782      branch-misses             #    0.24% of all branches

       0.076351367 seconds time elapsed
```

```
普通矩阵乘法: 0.072766 s

 Performance counter stats for './b.out':

           170,925      cache-misses

       0.077823373 seconds time elapsed
```



经过上面的比较可知，Strassen 算法的运行时间和 CPI 都要明显优于一般算法。但是，Strassen 算法的 cache-misses 要明显大于一般算法。笔者认为，原因在于 Strassen 算法要在不断创建空间，占用了不少空间开销，所以 cache-misses 会变大。而 mem-stores 则要优于一般算法。而 mem-stores 则表示存回内存的次数，也就是说，cache-miss 之后要存回内存的次数。所以，Strassen 的 mem-stores 要小于一般算法的 mem-stores，这也与实验相符。

2. **问题 2 和问题 3**

结果详见“解决办法”板块。

四、 遇到的问题及解决方法

本次是并行的第二次作业，我遇到了不少困难，但也通过不断查询资料和与同学讨论，大部分获得了解决。

其中，第一个困难就是编写分治算法和 Strassen 算法的代码。一开始，我对如何传递参数可以减小开销这个问题纠结了很久。后来，经过查资料分析，发现传递下标可以有效解决开销问题。

第二个困难，在于 perf 工具的运用。一开始，我在 ubuntu 系统的下载上遇到了困难。后来，经过群里大佬的提示，发现对 ubuntu 进行换源可以有效解决问题。当我以为终于可以运用上 perf 的时候，我发现想要观察的 3 个指标竟然都是“not-supported”？？后来才发现是自己忘记开“硬件虚拟化”的按钮。当这些准备工作都完成后，我发现 mem-loads 指标还是 not-supported，后来群里有大佬说可以用 mem-stores 指标代替。

第三个困难，是在后面两题的理论计算中。一开始，我苦恼于为什么第一题没有给矩阵的大小。后来发现并不需要，因为整体计算可以划分为以 4 次循环为一单位进行重复计算。所以，只需分析“一个单位”就可以得到理论值。