并行与分布式计算作业

第二次作业

姓名：施天予

班级：计科 2 班

学号：19335174

一、问题描述

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_load等性能指标, 找出特征或者规律。
2. 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. Now consider the problem of multiplying a dense matrix with a vector using a two-loop dot-product formulation. The matrix is of dimension 4K x 4K. (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; j < dim; j++)
        c[i] += a[i][j] * b[j];
```

二、解决方案

一般算法

最直接的算法，即用一个矩阵的每一行的每个元素分别乘另一个矩阵每一列的每个元素，各个乘积相加得到结果矩阵的一个元素。使用三重循环实现，时间复杂度为 $O(n^3)$。

代码如下：

```cpp
#include<bits/stdc++.h>
using namespace std;

#define N 256

void multiply(int **A, int **B, int **C) {
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            for (int k = 0; k < N; ++k)
                C[i][j] += A[i][k] * B[k][j];
}

int main() {
    int **A = new int*[N];
    int **B = new int*[N];
    int **C = new int*[N];
    for(int i = 0; i < N; ++i) {
        A[i] = new int[N];
        B[i] = new int[N];
        C[i] = new int[N];
        memset(A[i], 0, sizeof(int)*N);
        memset(B[i], 0, sizeof(int)*N);
        memset(C[i], 0, sizeof(int)*N);
    }
    multiply(A, B, C);
    return 0;
}
```

分治算法

假定三个矩阵均为 $n \times n$ 矩阵，其中n为2的幂。在每个分解步骤中，$n \times n$ 矩阵都被划分为4个 $n/2 \times n/2$ 的子矩阵。

三个矩阵可以写成下面的格式：

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & A_{22} \end{bmatrix}$$

因此可以将公式$C = A \cdot B$改写为

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

其中

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21}$$
$$C_{12} = A_{11} \times B_{12} + A_{12} \times B_{22}$$
$$C_{21} = A_{21} \times B_{11} + A_{22} \times B_{21}$$
$$C_{22} = A_{21} \times B_{12} + A_{22} \times B_{22}$$

这样将矩阵不断分解为小矩阵进行计算，最后合并为一个矩阵。

因为分治算法矩阵的单位长度n应该是2的幂才能正确划分，所以我将n设为256，具体代码如下：

```
#include<bits/stdc++.h>
using namespace std;

#define N 256
int n = N;

void help(int *s, int *one, int *two, int *three, int *four) {
    int left = s[0];
    int right = s[1];
    int top = s[2];
    int bottom = s[3];

    one[0] = left;
    one[1] = (left + right) / 2;
    one[2] = top;
    one[3] = (top + bottom) / 2;

    two[0] = (left + right + 1) / 2;
    two[1] = right;
    two[2] = top;
    two[3] = (top + bottom) / 2;

    three[0] = left;
    three[1] = (left + right) / 2;
    three[2] = (top + bottom + 1) / 2;
    three[3] = bottom;

    four[0] = (left + right + 1) / 2;
    four[1] = right;
    four[2] = (top + bottom + 1) / 2;
    four[3] = bottom;
}

void divide(int **A, int **B, int **C, int *A_size,int *B_size) {
    if (n == 2) {
        C[A_size[2]][B_size[0]] = A[A_size[2]]
[A_size[0]]*B[B_size[2]][B_size[0]]+A[A_size[2]]
[A_size[1]]*B[B_size[3]][B_size[0]]+C[A_size[2]][B_size[0]];
```

```
        C[A_size[2]][B_size[1]] = A[A_size[2]][A_size[0]]*B[B_size[2]][B_size[1]]+A[A_size[2]][A_size[1]]*B[B_size[3]][B_size[1]]+C[A_size[2]][B_size[1]];
        C[A_size[3]][B_size[0]] = A[A_size[3]][A_size[0]]*B[B_size[2]][B_size[0]]+A[A_size[3]][A_size[1]]*B[B_size[3]][B_size[0]]+C[A_size[3]][B_size[0]];
        C[A_size[3]][B_size[1]] = A[A_size[3]][A_size[0]]*B[B_size[2]][B_size[1]]+A[A_size[3]][A_size[1]]*B[B_size[3]][B_size[1]]+C[A_size[3]][B_size[1]];
    }
    else {
        n /= 2;
        int A_one[4], A_two[4], A_three[4], A_four[4];
        int B_one[4], B_two[4], B_three[4], B_four[4];

        help(A_size, A_one, A_two, A_three, A_four);
        help(B_size, B_one, B_two, B_three, B_four);

        divide(A, B, C, A_one, B_one);
        divide(A, B, C, A_two, B_three);
        divide(A, B, C, A_one, B_two);
        divide(A, B, C, A_two, B_four);
        divide(A, B, C, A_three, B_one);
        divide(A, B, C, A_four, B_three);
        divide(A, B, C, A_three, B_two);
        divide(A, B, C, A_four, B_four);
    }
}

int main(){
    int **A = new int*[N];
    int **B = new int*[N];
    int **C = new int*[N];
    for(int i = 0; i < N; ++i) {
        A[i] = new int[N];
        B[i] = new int[N];
        C[i] = new int[N];
        memset(A[i], 0, sizeof(int)*N);
        memset(B[i], 0, sizeof(int)*N);
        memset(C[i], 0, sizeof(int)*N);
    }
    int a_shape[4] = {0, N-1, 0, N-1};
    int b_shape[4] = {0, N-1, 0, N-1};
    divide(A, B, C, a_shape, b_shape);
    return 0;
}

```

Strassen算法

Strassen算法的核心思想是令递归树稍微不那么茂盛一点儿，即只递归7次而不是8次 $n/2 \times n/2$ 矩阵的乘法。减少一次矩阵乘法带来的代价可能是额外几次矩阵的加法，但只是常数次。

Strassen算法包含4个步骤

1. 将输入矩阵A、B和输出矩阵C分解为$n/2 \times n/2$ 的子矩阵，采用下标计算方法。
2. 创建10个 $n/2 \times n/2$ 的矩阵S_1，S_2，…，S_{10}，每个矩阵保存步骤1中创建的两个子矩阵的和或差。
3. 用步骤1中创建的子矩阵和步骤2中创建的10个矩阵，递归地计算7个矩阵P_1，P_2，…，P_7。每个矩阵P_i都是 $n/2 \times n/2$ 的。
4. 通过P_i矩阵的不同组合进行加减运算，计算出结果矩阵C的子矩阵C_{11}，C_{12}，C_{21}，C_{22}。

在步骤2中，创建如下10个矩阵：

$$
\begin{aligned}
S_1 &= B_{12} - B_{22} \\
S_2 &= A_{11} + A_{12} \\
S_3 &= A_{21} + A_{22} \\
S_4 &= B_{21} - B_{11} \\
S_5 &= A_{11} + A_{22} \\
S_6 &= B_{11} + B_{22} \\
S_7 &= A_{12} - A_{22} \\
S_8 &= B_{21} + B_{22} \\
S_9 &= A_{11} - A_{21} \\
S_{10} &= B_{11} + B_{12}
\end{aligned}
$$

在步骤3中，递归地计算7次 $n/2 \times n/2$ 矩阵的乘法，如下所示：

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 \\
P_2 &= S_2 \cdot B_{22} \\
P_3 &= S_3 \cdot B_{11} \\
P_4 &= A_{22} \cdot S_4 \\
P_5 &= S_5 \cdot S_6 \\
P_6 &= S_7 \cdot S_8 \\
P_7 &= S_9 \cdot S_{10}
\end{aligned}
$$

步骤4计算出C的4个子矩阵：

$$C_{11} = P_5 + P_4 - P_2 + P_6$$
$$C_{12} = P_1 + P_2$$
$$C_{21} = P_3 + P_4$$
$$C_{22} = P_5 + P_1 - P_3 - P_7$$

最后将这四个矩阵合并就是最终结果。

Strassen 算法在低维度上效率非常低，因为算法本身需要创建大量的临时矩阵，较好的解决办法是在维度较低时直接使用一般算法而不是循环递归调用，所以我在 $n \leq 16$ 时直接使用一般算法，提高效率。具体代码如下：

```
#include<bits/stdc++.h>
using namespace std;

#define N 256
int n = N;

void ADD(int **A, int **B, int **C, int n) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            C[i][j] =  A[i][j] + B[i][j];
}
void SUB(int **A, int **B, int **C, int n) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            C[i][j] =  A[i][j] - B[i][j];
}
void MUL( int **A, int **B, int **C, int n) {
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            for (int k = 0; k < n; ++k)
                C[i][j] += A[i][k] * B[k][j];
}

void strassen(int **A, int **B, int **C, int n) {
    int half = n / 2;
    if (n <= 4) {
        MUL(A, B, C, n);
        return;
    }
    int **A11 = new int*[half];int **A12 = new int*[half];
    int **A21 = new int*[half];int **A22 = new int*[half];
    int **B11 = new int*[half];int **B12 = new int*[half];
    int **B21 = new int*[half];int **B22 = new int*[half];
    int **C11 = new int*[half];int **C12 = new int*[half];
    int **C21 = new int*[half];int **C22 = new int*[half];

    int **S1 = new int*[half];int **S2 = new int*[half];
```

```
    int **S3 = new int*[half];int **S4 = new int*[half];
    int **S5 = new int*[half];int **S6 = new int*[half];
    int **S7 = new int*[half];int **S8 = new int*[half];
    int **S9 = new int*[half];int **S10 = new int*[half];

    int **P1 = new int*[half];int **P2 = new int*[half];
    int **P3 = new int*[half];int **P4 = new int*[half];
    int **P5 = new int*[half];int **P6 = new int*[half];
    int **P7 = new int*[half];

    for (int i = 0; i < half; i++) {
        A11[i] = new int[half];A12[i] = new int[half];
        A21[i] = new int[half];A22[i] = new int[half];
        B11[i] = new int[half];B12[i] = new int[half];
        B21[i] = new int[half];B22[i] = new int[half];
        C11[i] = new int[half];C12[i] = new int[half];
        C21[i] = new int[half];C22[i] = new int[half];

        S1[i] = new int[half];S2[i] = new int[half];
        S3[i] = new int[half];S4[i] = new int[half];
        S5[i] = new int[half];S6[i] = new int[half];
        S7[i] = new int[half];S8[i] = new int[half];
        S9[i] = new int[half];S10[i] = new int[half];

        P1[i] = new int[half];P2[i] = new int[half];
        P3[i] = new int[half];P4[i] = new int[half];
        P5[i] = new int[half];P6[i] = new int[half];
        P7[i] = new int[half];
    }
    for (int i = 0; i < half; ++i) {
        for (int j = 0; j < half; ++j) {
            A11[i][j] = A[i][j];
            A12[i][j] = A[i][j+half];
            A21[i][j] = A[i+half][j];
            A22[i][j] = A[i+half][j+half];
            B11[i][j] = B[i][j];
            B12[i][j] = B[i][j+half];
            B21[i][j] = B[i+half][j];
            B22[i][j] = B[i+half][j+half];
        }
    }
    SUB(B12, B22, S1, half);
    ADD(A11, A12, S2, half);
    ADD(A21, A22, S3, half);
    SUB(B21, B11, S4, half);
    ADD(A11, A22, S5, half);
    ADD(B11, B22, S6, half);
    SUB(A12, A22, S7, half);
    ADD(B21, B22, S8, half);
```

```
    SUB(A11, A21, S9, half);
    ADD(B11, B12, S10, half);

    strassen(A11, S1, P1, half);
    strassen(S2, B22, P2, half);
    strassen(S3, B11, P3, half);
    strassen(A22, S4, P4, half);
    strassen(S5, S6, P5, half);
    strassen(S7, S8, P6, half);
    strassen(S9, S10, P7, half);

    ADD(P4, P5, C11, half);SUB(C11, P2, C11, half);ADD(C11, P6, C11, half);
    ADD(P1, P2, C12, half);
    ADD(P3, P4, C21, half);
    ADD(P1, P5, C22, half);SUB(C22, P3, C22, half);SUB(C22, P7, C22, half);

    for (int i = 0; i < half; ++i) {
        for (int j = 0; j < half; ++j) {
            C[i][j] = C11[i][j];
            C[i][j+half] = C12[i][j];
            C[i+half][j] = C21[i][j];
            C[i+half][j+half] = C22[i][j];
        }
    }
    for (int i = 0; i < half; ++i) {
        delete[] A11[i];delete[] A12[i];delete[] A21[i];delete[] A22[i];
        delete[] B11[i];delete[] B12[i];delete[] B21[i];delete[] B22[i];
        delete[] C11[i];delete[] C12[i];delete[] C21[i];delete[] C22[i];
        delete[] S1[i];delete[] S2[i];delete[] S3[i];delete[] S4[i];delete[] S5[i];
        delete[] S6[i];delete[] S7[i];delete[] S8[i];delete[] S9[i];delete[] S10[i];
        delete[] P1[i];delete[] P2[i];delete[] P3[i];delete[] P4[i];
        delete[] P5[i];delete[] P6[i];delete[] P7[i];
    }
    delete[] A11;delete[] A12;delete[] A21;delete[] A22;
    delete[] B11;delete[] B12;delete[] B21;delete[] B22;
    delete[] C11;delete[] C12;delete[] C21;delete[] C22;
    delete[] S1;delete[] S2;delete[] S3;delete[] S4;delete[] S5;
    delete[] S6;delete[] S7;delete[] S8;delete[] S9;delete[] S10;
    delete[] P1;delete[] P2;delete[] P3;delete[] P4;
    delete[] P5;delete[] P6;delete[] P7;
}
```

```
128
129 int main(){
130     int **A = new int*[N];
131     int **B = new int*[N];
132     int **C = new int*[N];
133     for(int i = 0; i < N; ++i) {
134         A[i] = new int[N];
135         B[i] = new int[N];
136         C[i] = new int[N];
137         memset(A[i], 0, sizeof(int)*N);
138         memset(B[i], 0, sizeof(int)*N);
139         memset(C[i], 0, sizeof(int)*N);
140     }
141     strassen(A, B, C, n);
142     return 0;
143 }
```

三、实验结果

问题1

在实验过程中，我发现我的 Ubuntu 下的 perf 的 perf list 中没有mem-loads这个指标，所以我使用了 L1-dcache-loads 指标作为代替，它的含义为一级数据缓存读取次数。这个指标会映射到 MEM_UOPS_RETIRED.ALL_LOADS，并返回正确的装载次数。

在终端输入相应的命令，可以看到如下结果：

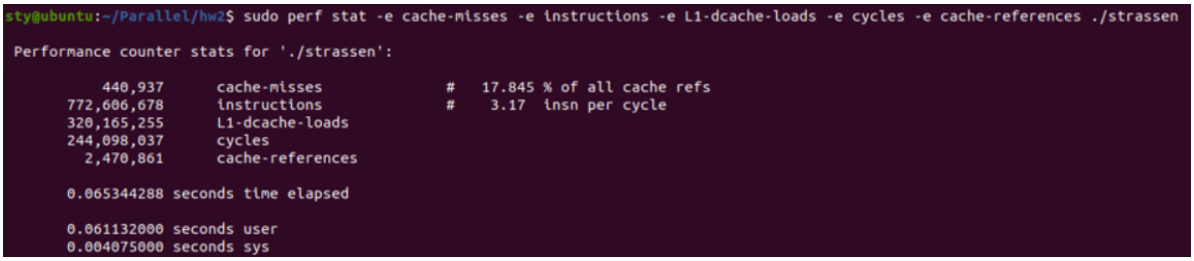
一般算法

```
sty@ubuntu:~/Parallel/hw2$ sudo perf stat -e cache-misses -e instructions -e L1-dcache-loads -e cycles -e cache-references ./simple

Performance counter stats for './simple':

         171,114      cache-misses              #    9.178 % of all cache refs
     843,953,976      instructions              #    2.83  insn per cycle
     354,040,112      L1-dcache-loads
     297,777,555      cycles
       1,864,441      cache-references

       0.091595038 seconds time elapsed

       0.082847000 seconds user
       0.007890000 seconds sys
```

分治算法

```
sty@ubuntu:~/Parallel/hw2$ sudo perf stat -e cache-misses -e instructions -e L1-dcache-loads -e cycles -e cache-references ./divide

Performance counter stats for './divide':

          87,237      cache-misses              #   32.481 % of all cache refs
       4,592,847      instructions              #    0.55  insn per cycle
       1,207,694      L1-dcache-loads
       8,318,938      cycles
         268,579      cache-references

       0.004956109 seconds time elapsed

       0.004743000 seconds user
       0.000000000 seconds sys
```

Strassen算法



结果分析

- cache-misses：一般算法的cache miss rate最小，为9.178%，分治算法的cache miss rate最大，为32.481%。可能是因为分治算法在递归时对不同数据块频繁访问，导致 cache的大量miss。
- CPI：根据三种算法的 instructions per cycle，可以算出CPI从小到大依次是Strassen算法、一般算法、分治算法。instructions从小到大依次是分治算法、Strassen算法、一般算法。这可能是因为Strassen算法较为复杂，指令并行度高，并减少了一定的矩阵乘法和加法运算次数。而分治算法虽然也有较高的并行度，但是它的 cache miss rate 较高，使得访存周期较多，进而使得整体的 CPI 增多，并且其减少了较多的矩阵加法和乘法运算次数，使得 instructions 数最小。
- L1-dcache-loads：Strassen算法和一般算法差不多，分治算法最小。Strassen算法创建了大量中间矩阵，所以其负载也会较大。
- 运行时间：从快到慢依次是分治算法、Strassen算法、一般算法。理论上Strassen算法应该是时间复杂度最小的，但是由于矩阵的规模较小，所以无法体现Strassen算法的优势，反而计算起来较为复杂。

问题2

处理器执行一次循环体 `C[i][j] += A[i][k] * B[k][j]` 会执行两次浮点计算。由于访问 Cache 的时间远小于访问 Dram 的时间，所以可以忽略不计。而因为 cache line size 是四个字，所以可以存储4个浮点数，又因为 a 和 b 存在不同的内存区域，所以会对 a 和 b 各产生一次cache miss，即每4次循环会出现2次 cache miss。因此：

处理器周期：$\frac{1}{1GHz} = 1ns$

每4个循环延迟时间：$1ns * 100 * 2 = 200ns$

每4个循环浮点计算次数：$4 * 2 = 8$

浮点计算速度：$\frac{8}{200ns} = 40MFLOPS$

问题3

考虑最大速率情况，b 数组全部已缓冲在 cache 中（b 数组占用 16KB 而 cache 总大小为 32KB），每次循环 b 数组元素读取都将 cache 命中，而 c 数组元素的读取次数较少可忽略不计，因此只需要考虑 a 数组元素的读取延迟。

每4个循环延迟时间：$1ns * 100 = 100ns$

每4个循环浮点计算次数：$4 * 2 = 8$

浮点计算速度：$\frac{8}{100ns} = 80MFLOPS$

四、遇到的问题及解决方法

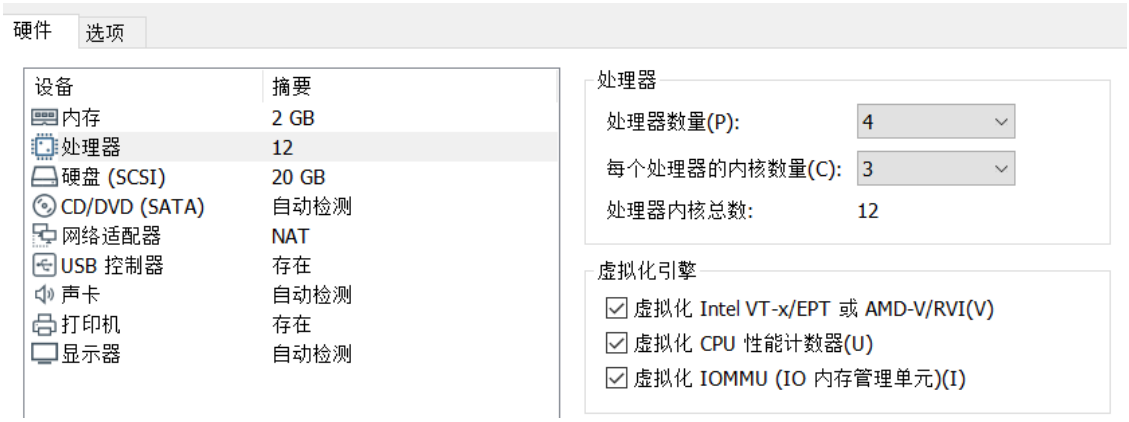
1. 使用分治算法时矩阵的单维长度必须是2的幂，否则无法划分

   解决方法：将矩阵单维长度设为256（300并不重要）

2. 使用perf 观察 cahce-misses 等指标时发现< not supported >

```
Performance counter stats for './simple':

  <not supported>      cache-misses
```

   解决方法：使用虚拟机的虚拟化引擎功能



3. perf list中没有mem-loads指标

   解决方法：不知道由于什么原因，我在perf list中没有找到mem-loads指标，只好使用 L1-dcache-loads 指标代替，可以取得类似的效果。