并行与分布式作业

第二次作业

姓名：　何泽

班级：　18 级计科（超算方向）

学号：　18340052

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
2. Consider a memory system with a level 1 cache of 32KB 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 processes 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 16KB 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

Ⅰ、问题一

1. 算法描述

①一般算法

最直白、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。具体实现即三重循环。

②分治算法

不断地将矩阵分为四个矩阵分别计算，直到变为单个元素.

- 三个矩阵可以写成下面的格式：

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

- 那么相关的计算可以写成

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

- 同理A_{11}等一些子矩阵也可以写成相关的子矩阵，就这样将矩阵不断分解为小矩阵进行计算，最后归并为一个矩阵。
  同时，这种方法要求矩阵的行列必须为2的n次方。

③Strassen算法

Strassen算法同样是使用分治的思想解决问题，只不过，不同的是当矩阵的阶很大时就会采取一个递推式进行计算相关递推式。

计算如下表达式：

$$S_1 = B_{12} - B_{22}$$
$$S_2 = A_{11} + A_{12}$$
$$S_3 = A_{21} + A_{22}$$
$$S_4 = B_{21} - B_{11}$$
$$S_5 = A_{11} + A_{22}$$
$$S_6 = B_{11} + B_{22}$$
$$S_7 = A_{12} - A_{22}$$
$$S_8 = B_{21} + B_{22}$$
$$S_9 = A_{11} - A_{21}$$
$$S_10 = B_{11} + B_{12}$$

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

那么最终的矩阵结果为：

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

其中A11，A12，A21，A22和B11，B12，B21，B22分别为两个乘数A和B矩阵的四个子矩阵。C11，C12，C21，C22为最终的结果C矩阵的四个子矩阵。

那么只需要 将这四个矩阵合并就是最终结果。

2. 算法实现

> **为了之后测试三种算法的公平性，我准备将三种算法全都写到一个类里，在测试用不同算法计算时，只需要调用类里的不同函数即可。**

① 类的定义与构造函数实现

其中，构造函数就是传入行数、列数并随机赋值

```cpp
class Matrix{

private:
    int row;
    int col;
    vector<vector<int>> data;

public:
    Matrix(int row, int col) :data(row),row(row),col(col){
        for (int i = 0; i < row; i++){
            data[i].resize(col);
        }
        srand(time(0));
        for (int i = 0; i < row; i++){
            for (int j = 0; j < col; j++){
                data[i][j] = rand();
            }
        }
    }

    Matrix(int row1, int col1, int row2, int col2, const Matrix& a) :row(row2 - row1), col(col2 - col1),data(row){
        for (int i = 0; i < row; i++){
            data[i].resize(col);
        }

        for (int i = 0; i < row; i++){
            for (int j = 0; j < col; j++){
                data[i][j] = a.get(col1 + i, row1 + j);
            }
        }
    }

    ~Matrix(){
        this->row = 0;
        this->col = 0;
    }

    Matrix(const Matrix& a){
        *this = a;
   }

    Matrix* cal1(const Matrix&);
    Matrix* cal2(const Matrix&);
    Matrix* cal3(const Matrix&);

    Matrix* operator+(const Matrix&);
    Matrix* operator-(const Matrix&);
    Matrix* operator*(const Matrix&);
    static Matrix* add(const Matrix*, const Matrix*);
    static Matrix* sub(const Matrix*, const Matrix*);
    static Matrix* plus(const Matrix*, const Matrix*,const Matrix*,const Matrix*);

    vector<int> operator[](const int);
    int get(const int,const int) const;
    void set(const int, const int, int);
```

```
56
57      bool isSimilar(const Matrix& x);
58      void clear(int);
59
60  };
```

② 除计算乘积函数之外其他函数的实现

- 与另一个矩阵加减乘
  定义：

```
1   Matrix* operator+(const Matrix&);
2   Matrix* operator-(const Matrix&);
3   Matrix* operator*(const Matrix&);
```

实现：

```
1   Matrix* Matrix::operator+(const Matrix& a)
2   {
3       if (!isSimilar(a)){
4           return nullptr;
5       }
6
7       Matrix *ptr = new Matrix(a.row, a.col);
8       for (int i = 0; i < row; i++){
9           for (int j = 0; j < col; j++){
10              ptr->set(i, j, this->get(i, j) + a.get(i, j));
11          }
12      }
13
14      return ptr;
15  }
16
17  Matrix* Matrix::operator-(const Matrix& a)
18  {
19      if (!isSimilar(a)){
20          return nullptr;
21      }
22
23      Matrix *ptr = new Matrix(a.row, a.col);
24      for (int i = 0; i < row; i++){
25          for (int j = 0; j < col; j++){
26              ptr->set(i, j, this->get(i, j) - a.get(i, j));
27          }
28      }
29
30      return ptr;
31  }
```

由于不同方法乘法方法不同，乘法函数后面再说

- 两矩阵相加减

```
1   Matrix* Matrix::add(const Matrix* p1, const Matrix* p2)
2   {
3       if (!(p1->col == p2->col && p1->row == p2->row)){
4           return nullptr;
5       }
```

```
6
7       Matrix *ptr = new Matrix(p1->row, p1->col);
8       for (int i = 0; i < p1->row; i++){
9           for (int j = 0; j < p1->col; j++){
10              ptr->set(i, j, (p1->get(i, j) + p2->get(i, j)));
11          }
12      }
13
14      return ptr;
15  }
16
17
18  Matrix* Matrix::sub(const Matrix* p1, const Matrix* p2)
19  {
20      if (!(p1->col == p2->col && p1->row == p2->row)){
21          return nullptr;
22      }
23
24      Matrix *ptr = new Matrix(p1->row, p1->col);
25      for (int i = 0; i < p1->row; i++){
26          for (int j = 0; j < p1->col; j++){
27              ptr->set(i, j, (p1->get(i, j) - p2->get(i, j)));
28          }
29      }
30
31      return ptr;
32  }
```

- 将四个矩阵合并为一个矩阵

```
1   Matrix* Matrix::plus(const Matrix* p1, const Matrix* p2,const Matrix*
    p3, const Matrix* p4)
2   {
3       //不符合可以进行合并的条件
4       if (!(p1->row == p2->row && p2->col == p4->col && p4->row == p3->row
    && p1->col == p3->col)){
5           return nullptr;
6       }
7
8       Matrix* ptr = new Matrix(p1->row + p3->row, p2->col + p1->col);
9       ptr->clear(0);
10
11      for (int i = 0; i < p1->row; i++){
12          for (int j = 0; j < p1->col; j++){
13              ptr->set(i, j, p1->get(i, j));
14          }
15      }
16
17      for (int i = 0; i < p2->row; i++){
18          for (int j = 0; j < p2->col; j++){
19              ptr->set(i, j + p1->col, p2->get(i, j));
20          }
21      }
22
23      for (int i = 0; i < p3->row; i++){
24          for (int j = 0; j < p3->col; j++){
25              ptr->set(i + p1->row, j, p3->get(i, j));
26          }
```

```
        }

        for (int i = 0; i < p4->row; i++){
            for (int j = 0; j < p4->col; j++){
                ptr->set(p1->row + i, p1->col + j, p4->get(i, j));
            }
        }

        return ptr;
    }
```

③三种计算方法实现

- 定义

```
Matrix* cal1(const Matrix&);
Matrix* cal2(const Matrix&);
Matrix* cal3(const Matrix&);
```

- 一般算法
  即最简单的三重循环

```
Matrix* Matrix::cal1(const Matrix& a)
{
    Matrix *ptr = new Matrix(row, a.col);
    ptr->clear();
    for (int i = 0; i < row; i++){
        for (int j = 0; j < a.col; j++){
            for (int k = 0; k < col; k++){
                ptr->set(i, j, ptr->get(i, j) + get(i, k) * a.get(k,
j));
            }
        }
    }
    return ptr;
}
```

- 归并算法
  原理上面已说过，不断地将矩阵分为四个矩阵分别计算，直到变为单个元素

```
Matrix* Matrix::cal2(const Matrix& a)
{

    if (a.row == 1)             //当前的矩阵为单个的元素
    {
        Matrix *ptr = new Matrix(a.row, a.col);
        ptr->clear((this->get(0, 0))*(a.get(0, 0)));
        return ptr;
    }
    //将第一个矩阵分解为四个子矩阵
    Matrix A11(0,0,(row+1)/2-1,(col+1)/2-1,*this);
    Matrix A12((row+1) / 2, 0, row, (col+1) / 2-1, *this);
    Matrix A21(0, (col+1) / 2, (row+1) / 2-1, col, *this);
    Matrix A22((row+1) / 2, (col+1) / 2, row, col, *this);
    //将第二个矩阵分解为四个子矩阵
    Matrix B11(0,0,(row+1)/2-1,(col+1)/2-1, a);
    Matrix B12((row+1) / 2, 0, row, (col+1) / 2-1, a);
    Matrix B21(0, (col+1) / 2, (row+1) / 2-1, col, a);
```

```
    Matrix B22((row+1) / 2, (col+1) / 2, row, col, a);

    Matrix *C11 = Matrix::add(A11.cal2(B11), A12.cal2(B21));
    Matrix *C12 = Matrix::add(A11.cal2(B12), A12.cal2(B22));
    Matrix *C21 = Matrix::add(A21.cal2(B11), A22.cal2(B21));
    Matrix *C22 = Matrix::add(A21.cal2(B12), A22.cal2(B22));

    //将C11，C12，C21，C22合并为一个完整的矩阵
    Matrix* ptr = Matrix::plus(C11, C12, C21, C22);

    return ptr;
}
```

- Strassen算法

  原理上面也已经说过

```
Matrix* Matrix::cal3(const Matrix& a)
{
    if (a.row < 2)
    {
        return this->cal1(a);
    }
    //将第一个矩阵分解为四个子矩阵
    Matrix A11(0,0,(row+1)/2-1,(col+1)/2-1,*this);
    Matrix A12((row+1) / 2, 0, row, (col+1) / 2-1, *this);
    Matrix A21(0, (col+1) / 2, (row+1) / 2-1, col, *this);
    Matrix A22((row+1) / 2, (col+1) / 2, row, col, *this);
    //将第二个矩阵分解为四个子矩阵
    Matrix B11(0,0,(row+1)/2-1,(col+1)/2-1, a);
    Matrix B12((row+1) / 2, 0, row, (col+1) / 2-1, a);
    Matrix B21(0, (col+1) / 2, (row+1) / 2-1, col, a);
    Matrix B22((row+1) / 2, (col+1) / 2, row, col, a);

    Matrix* S1 = B12 - B22;
    Matrix* S2 = A11 + A12;
    Matrix* S3 = A21 + A22;
    Matrix* S4 = B21 - B11;
    Matrix* S5 = A11 + A22;
    Matrix* S6 = B11 + B22;
    Matrix* S7 = A12 - A22;
    Matrix* S8 = B21 + B22;
    Matrix* S9 = A11 - A21;
    Matrix* S10 = B11 + B12;

    Matrix* P1 = B12 - B22;
    Matrix* P2 = B12 - B22;
    Matrix* P3 = B12 - B22;
    Matrix* P4 = B12 - B22;
    Matrix* P5 = B12 - B22;
    Matrix* P6 = B12 - B22;
    Matrix* P7 = B12 - B22;

    P1 = A11.cal3(*S1);
    P2 = S2->cal3(B22);
    P3 = S3->cal3(B11);
    P4 = A22.cal3(*S4);
    P5 = S5->cal3(*S6);
```

```
        P6 = S7->cal3(*S8);
        P7 = S9->cal3(*S10);

        Matrix *C11 = Matrix::sub(Matrix::add(P5, P4), Matrix::sub(P2, P6));
        Matrix *C12 = Matrix::add(P1, P2);
        Matrix *C21 = Matrix::add(P3, P4);
        Matrix *C22 = Matrix::sub(Matrix::add(P5, P1), Matrix::add(P3, P7));

        Matrix* ptr = Matrix::plus(C11, C12, C21, C22);

        return ptr;
    }
```

④ 主函数

首先，三个文件的类是相同的，只是调用的函数不同，例如在分治算法中的乘法函数如下：

```
Matrix* Matrix::operator*(const Matrix& a)
{
    if (a.row != this->col){
        return nullptr;
    }
    return this->cal2(a);
}
```

只需要在最后的return不同函数即可
则主函数：

```
int main()
{
    Matrix a(255,255),b(255,255),c(255,255);
    Matrix& p=a;
    Matrix& q=b;
    Matrix* i=&c;
    i=p*q;
    cout<<"一般算法已完成"<<endl;
    return 0;
}
```

声明256x256的矩阵，之后便开始计算。

II、 问题二

- 假设所需要的数据均在cache内
- 那么所有的数据，共32KB即4K个word，大约需要$400\mu s$
- 这些数据计算，即每个向量有2K个 word，则共需要4K次计算，即1K个周期，时间是$1\mu s$
- 那么总时间为$400\mu s + 1\mu s = 401\mu s$
- 那么可以计算得到结果：$4K \div 401\mu s = 99.75\ FLOPS$

III、 问题三

- 每一行占16KB，那么每两行就占满了cache，取完两行的 数据就要从DRAM中取数
- 那么取所有数据需要的时间：

    - 从cache中取的：在cache中取一次是4K个word，约$400\mu s$,一共需要取2K次，即$0.8\,\mathrm{s}$
    - 从DRAM：共2×10^5个周期，$200\mu s$
    - 共$800200\mu s$
- 一共进行了$2\times\left(4\times10^3\right)^3$次计算，等于$2^7\times10^9$
- 那么可计算出结果：

$$\frac{2^7\times10^9}{800200\mu s}=159.96\,GFLPOS$$

三、实验结果

1. 问题一

① cache miss

```
gordon@gordon-TravelMate-X349-G2-M:~/桌面/hz$ perf stat -e cache-misses ./par1.o一般算法已完成

 Performance counter stats for './par1.o':

            91,745      cache-misses

       0.441940295 seconds time elapsed

       0.441910000 seconds user
       0.000000000 seconds sys


gordon@gordon-TravelMate-X349-G2-M:~/桌面/hz$ perf stat -e cache-misses ./par2.o
分治算法已完成

 Performance counter stats for './par2.o':

         9,626,039      cache-misses

       6.023089517 seconds time elapsed

       5.934232000 seconds user
       0.087973000 seconds sys


gordon@gordon-TravelMate-X349-G2-M:~/桌面/hz$ perf stat -e cache-misses ./par3.o
Strassen算法已完成

 Performance counter stats for './par3.o':

        12,395,403      cache-misses

       6.907553048 seconds time elapsed

       6.781837000 seconds user
       0.124033000 seconds sys
```

可以看出，一般算法最简单，没有递归所以cache miss最少；而分治算法和Strassen就相对多很多了，时间也长很多；而这两个相比由于Strassen算法更复杂，cache miss也就多一些。

② IPC

```
gordon@gordon-TravelMate-X349-G2-M:~/桌面/hz$ perf stat ./par1.o
一般算法已完成

 Performance counter stats for './par1.o':

          440.65 msec task-clock                #    0.999 CPUs utilized
               4      context-switches          #    0.009 K/sec
               0      cpu-migrations            #    0.000 K/sec
             380      page-faults               #    0.862 K/sec
   1,362,822,498      cycles                    #    3.093 GHz
   3,900,997,637      instructions              #    2.86  insn per cycle
     439,235,442      branches                  #  996.780 M/sec
          95,754      branch-misses             #    0.02% of all branches

       0.440972185 seconds time elapsed

       0.440923000 seconds user
       0.000000000 seconds sys


gordon@gordon-TravelMate-X349-G2-M:~/桌面/hz$ perf stat ./par2.o
分治算法已完成

 Performance counter stats for './par2.o':

         6,041.41 msec task-clock                #    1.000 CPUs utilized
              31      context-switches          #    0.005 K/sec
               0      cpu-migrations            #    0.000 K/sec
         108,330      page-faults               #    0.018 M/sec
  18,684,443,005      cycles                    #    3.093 GHz
  41,279,398,630      instructions              #    2.21  insn per cycle
   6,495,400,655      branches                  # 1075.147 M/sec
      34,541,361      branch-misses             #    0.53% of all branches

       6.042349567 seconds time elapsed

       5.901662000 seconds user
       0.140039000 seconds sys


gordon@gordon-TravelMate-X349-G2-M:~/桌面/hz$ perf stat ./par3.o
Strassen算法已完成

 Performance counter stats for './par3.o':

         6,838.43 msec task-clock                #    1.000 CPUs utilized
              29      context-switches          #    0.004 K/sec
               0      cpu-migrations            #    0.000 K/sec
         143,510      page-faults               #    0.021 M/sec
  21,142,843,677      cycles                    #    3.092 GHz
  48,061,861,312      instructions              #    2.27  insn per cycle
   7,510,559,969      branches                  # 1098.287 M/sec
      11,688,952      branch-misses             #    0.16% of all branches

       6.839519737 seconds time elapsed
```

可以看到，一般算法的IPC为2.86，分治算法为2.21，Strassen为2.27。一般算法和另两种相比稍大，一个周期执行的指令稍多，这应该是因为运行的是256x256的矩阵，相对还是较小，所以相对有递归的算法更快一些。而另两个相比，可以看出Strassen比单纯的分治算法要快一些的。

再看branch-miss占的比率，一般算法同样是最少的，而另两者同样是Strassen更优。

③ mem_load

```
gordon@gordon-TravelMate-X349-G2-M:~/桌面/hz$ perf stat -e mem-loads ./par1.o
一般算法已完成

 Performance counter stats for './par1.o':

                 0      mem-loads

       0.441286802 seconds time elapsed

       0.441275000 seconds user
       0.000000000 seconds sys


gordon@gordon-TravelMate-X349-G2-M:~/桌面/hz$ perf stat -e mem-loads ./par2.o
分治算法已完成

 Performance counter stats for './par2.o':

                 0      mem-loads

       6.030203902 seconds time elapsed

       5.925578000 seconds user
       0.104027000 seconds sys


gordon@gordon-TravelMate-X349-G2-M:~/桌面/hz$ perf stat -e mem-loads ./par3.o
Strassen算法已完成

 Performance counter stats for './par3.o':

                 0      mem-loads

       6.879902739 seconds time elapsed

       6.714410000 seconds user
       0.164058000 seconds sys
```

这方面三者均为0。

问题二、三已在上面说明

四、遇到的问题及解决办法

本次作业并未遇到什么问题