分布式系统作业

第 2 次作业

姓名：郝裕玮
班级：计科 1 班
学号：18329015

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、考虑一个内存系统，其一级缓存为 32 KB，DRAM 为 512 MB，处理器运行频率为 1 GHz。L1 缓存的延迟为一个周期，DRAM 的延迟为 100 个周期。 在每个内存周期中，处理器获取四个字（缓存线大小为四个字）。 两个向量的点积的最高可实现性能是多少？ 注意：如有必要，请假设最佳缓存放置策略。

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

3、现在考虑使用双环点积公式将密集矩阵与向量相乘的问题。矩阵的维度为 4K x 4K。（矩阵的每一行占用 16 KB 的存储空间。）可实现的峰值性能是多少 这种技术使用基于双循环点积的矩阵向量积？

```
/* 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

对于第 1 题：

我们先编写最为简单的一般算法，即通过三层循环来进行矩阵运算（即根据定义逐个相乘和累加）。代码及其分析如下所示（分析已放在代码注释中）：

```
#include<iostream>

using namespace std;

//矩阵一般乘法
void simple(int**a,int**b,int**c,int n){
    int i,j,k;
```

```
    for(i=0;i<=n-1;i++){
        for(j=0;j<=n-1;j++){
            for(k=0;k<=n-1;k++){
                c[i][j]+=a[i][k]*b[k][j];//根据矩阵计算公式得出（注意不能直接等于，而应该是累加）
            }
        }
    }
}

int main(){
    int n=512;//矩阵维度，在这里代表矩阵是 512*512 的
    //因为分治和 Strassen 的应用场景均是矩阵维度为 2 的 n 次方，所以将 300*300 统一修改为 512*512
    int i,j;
    int** a,**b,**c;//声明二维数组指针，为何不直接声明二维数组：int a[n][n] 会在 Strassen 程序中说明

    a=new int*[n];
    b=new int*[n];
    c=new int*[n];

    for(i=0;i<=n-1;i++){
        a[i]=new int[n];
        b[i]=new int[n];
        c[i]=new int[n];
    }
    //以上均为创建二维数组的过程

    //给数组 a 赋值
    for(i=0;i<=n-1;i++){
        for(j=0;j<=j-1;j++){
            a[i][j]=i+1;
        }
    }

    //给数组 b 赋值
    for(i=0;i<=n-1;i++){
        for(j=0;j<=j-1;j++){
            b[i][j]=j+1;
        }
    }
```

```
    //为了与分治和 Strassen 保持一致（因为这两个程序涉及到递归函数），所以在这里不直接进行运算而是通过调用函数进行运算
    simple(a,b,c,n);

}
```

由于分治算法和 Strassen 算法原理是一致的，所以这里只写出更加优化的 Strassen 算法去与一般算法进行性能比较。

Strassen 代码如下所示（分析已放在代码注释中）：

```
#include <iostream>

using namespace std;

//矩阵加法
void add(int** a,int** b,int** c,int n){
    int i,j;
    for(i=0;i<=n-1;i++){
        for(j=0;j<=n-1;j++){
            c[i][j]=a[i][j]+b[i][j];
        }
    }
}

//矩阵减法
void sub(int** a,int** b,int** c,int n){
    int i,j;
    for(i=0;i<=n-1;i++){
        for(j=0;j<=n-1;j++){
            c[i][j]=a[i][j]-b[i][j];
        }
    }
}

//矩阵乘法
void mul(int** a,int** b,int** c,int n){
    int i,j,k;
    for(i=0;i<=n-1;i++){
        for(j=0;j<=n-1;j++){
            c[i][j]=0;//因为乘法需要累加，所以要先清空数组 c 原先的数据
        }
    }
```

```
    for(i=0;i<=n-1;i++){
        for(j=0;j<=n-1;j++){
            for(k=0;k<=n-1;k++){
                c[i][j]+=a[i][k]*b[k][j];
            }
        }
    }
}

//Strassen 算法实现
void strassen(int **a,int **b,int **c,int n){
    int i,j;
    if(n<=64){
        mul(a,b,c,n);
        //查阅资料可知，最优的界限值在 32～128 之间，所以这里设置为当 n 缩小至
64 时不再需要继续四等分，直接进行普通矩阵乘法即可
        //继续递归分治会导致 Strassen 算法的效率降低
    }
    else{
        int** a11,**a12,**a21,**a22;
        int** b11,**b12,**b21,**b22;
        int** c11,**c12,**c21,**c22;

        int** P1,**P2,**P3,**P4,**P5,**P6,**P7;
        int** a_final,**b_final;

        a11=new int*[n/2];
        a12=new int*[n/2];
        a21=new int*[n/2];
        a22=new int*[n/2];

        b11=new int*[n/2];
        b12=new int*[n/2];
        b21=new int*[n/2];
        b22=new int*[n/2];

        c11=new int*[n/2];
        c12=new int*[n/2];
        c21=new int*[n/2];
        c22=new int*[n/2];

        P1=new int*[n/2];
        P2=new int*[n/2];
```

```
        P3=new int*[n/2];
        P4=new int*[n/2];
        P5=new int*[n/2];
        P6=new int*[n/2];
        P7=new int*[n/2];

        a_final=new int* [n/2];
        b_final=new int* [n/2];

        for(i=0;i<=n/2-1;i++){
            a11[i]=new int[n/2];
            a12[i]=new int[n/2];
            a21[i]=new int[n/2];
            a22[i]=new int[n/2];

            b11[i]=new int[n/2];
            b12[i]=new int[n/2];
            b21[i]=new int[n/2];
            b22[i]=new int[n/2];

            c11[i]=new int[n/2];
            c12[i]=new int[n/2];
            c21[i]=new int[n/2];
            c22[i]=new int[n/2];

            P1[i]=new int[n/2];
            P2[i]=new int[n/2];
            P3[i]=new int[n/2];
            P4[i]=new int[n/2];
            P5[i]=new int[n/2];
            P6[i]=new int[n/2];
            P7[i]=new int[n/2];

            a_final[i]=new int[n/2];
            b_final[i]=new int[n/2];
        }

        //根据原矩阵与四个子矩阵的位置关系将值赋给 A 和 B 的四个子矩阵
        for(i=0;i<=n/2-1;i++){
            for (j=0;j<=n/2-1;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];
            }
        }
        //以上均为声明新数组
a11,a12,a21,a22,b11,b12,b21,b22,c11,c12,c21,c22
        //以及 P1,P2,P3,P4,P5,P6,P7,a_final,b_final

        //P1=A11*(B12-B22)
        sub(b12,b22,b_final,n/2);
        strassen(a11,b_final,P3,n/2);//代入 strassen 继续递归，会在最底层进
行乘法运算（调用 mul 函数）

        //P2=(A11+A12)*B22
        add(a11,a12,a_final,n/2);
        strassen(a_final,b22,P5,n/2);//代入 strassen 继续递归，会在最底层进
行乘法运算（调用 mul 函数）

        //P3=(A21+A22)*B11
        add(a21,a22,a_final,n/2);
        strassen(a_final,b11,P2,n/2);//代入 strassen 继续递归，会在最底层进
行乘法运算（调用 mul 函数）

        //P4=A22*(B21-B11)
        sub(b21,b11,b_final,n/2);
        strassen(a22,b_final,P4,n/2);//代入 strassen 继续递归，会在最底层进
行乘法运算（调用 mul 函数）

        //P5=(A11+A22)*(B12+B22)
        add(a11,a22,a_final,n/2);
        add(b11,b22,b_final,n/2);
        strassen(a_final,b_final,P1,n/2);//代入 strassen 继续递归，会在最底
层进行乘法运算（调用 mul 函数）

        //P6=(A12-A22)*(B21+B22)
        sub(a12,a22,a_final,n/2);
        add(b21,b22,b_final,n/2);
        strassen(a_final, b_final, P7,n/2);//代入 strassen 继续递归，会在
最底层进行乘法运算（调用 mul 函数）

        //P7=(A11-A21)*(B11+B12)
```

```
        sub(a11,a21,a_final,n/2);
        add(b11,b12,b_final,n/2);
        strassen(a_final,b_final,P6,n/2);//代入 strassen 继续递归，会在最底层进行乘法运算（调用 mul 函数）


        //C11=P5+P4-P2+P6;
        add(P5,P6,a_final,n/2);
        sub(P4,P2,b_final,n/2);
        add(a_final,b_final,c11,n/2);

        //C12=P1+P2;
        add(P1,P2,c12,n/2);

        //C21=P3+P4;
        add(P3,P4,c21,n/2);

        //C22=P5+P1-P3-P7;
        sub(P5,P3,a_final,n/2);
        sub(P1,P7,b_final,n/2);
        add(a_final,b_final,c22,n/2);

        //根据四个子矩阵与原矩阵的位置关系将值赋给数组 c
        for(i=0;i<=n/2-1;i++){
            for(j=0;j<=n/2-1;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];
            }
        }
    }

}


int main()
{
    int n=512;//矩阵维度，在这里代表矩阵是 512*512 的
    int i,j;
    int** a,**b,**c;//声明二维数组指针

    a=new int *[n];
    b=new int *[n];
    c=new int *[n];
```

```
    for(i=0;i<=n-1;i++){
        a[i]=new int[n];
        b[i]=new int[n];
        c[i]=new int[n];
    }
    //以上均为创建二维数组的过程

    //给数组 a 赋值
    for(i=0;i<=n-1;i++){
        for(j=0;j<=j-1;j++){
            a[i][j]=i+1;
        }
    }

    //给数组 b 赋值
    for(i=0;i<=n-1;i++){
        for(j=0;j<=j-1;j++){
            b[i][j]=j+1;
        }
    }

    //初始化数组 c
    for(i=0;i<=n-1;i++){
        for(j=0;j<=j-1;j++){
            c[i][j]=0;
        }
    }

    strassen(a,b,c,n);
}
```

对于一般程序和 Strassen 程序的补充：因为在 Strassen 程序中需要不断地将矩阵划分为四个子矩阵，所以在进行递归的过程中，矩阵大小在不断变化。

如果采取这种函数原型：

```
void strassen(int a[][n], int b[][n], int c[][n],int n)
```

则必须指定数组的列数，但又因为矩阵大小不断变化且列数不可以用变量表示（必须使用常量），所以最终只能采取下面这种函数原型：

```
void add(int** a,int** b,int** c,int n)
```

这种函数原型使得只需传递二维数组的指针且无需传递数组大小，但是在创建数组时较为麻烦，如下所示：

```
int** a,**b,**c;//声明二维数组指针

a=new int *[n];
b=new int *[n];
c=new int *[n];

for(i=0;i<=n-1;i++){
    a[i]=new int[n];
    b[i]=new int[n];
    c[i]=new int[n];
}
//以上均为创建二维数组的过程
```

对于第 2 题：

因为每个内存周期中，缓存线大小为 4 个字=32 位，且一个 int 型变量占 4 bytes=32 位，所以缓存线一次可以存取 4 个元素。

易知第 1 次循环时对于数组 a 和 b 会发生 cache miss，之后立刻会存入当前 a 和 b 的 4 个元素（a[i]~a[i+3]和 b[i]~b[i+3]），使得后面 3 次循环均 cache hit。

又因为每次循环都会使 i+1，所以易知每 4 次循环会在第 1 次循环时发生 2 次 cache miss。在此期间的浮点数运算次数为 4*2=8（4 次循环，每次循环里 1 次加法 1 次乘法）次。

所以在每 4 次循环中，cache miss 所用的时钟周期为 2*100 = 200 cycles，cache hit 所用的时钟周期数为（3*4-2）*1 = 10 cycles。

所以，最高实现性能 $= \frac{8}{(200+10)*\frac{1}{1\text{GHz}}} \approx 38.095\ MFlops$

对于第 3 题：

对于数组 a，它的 cache miss 发生在内部循环中，每 4 次内部循环就会发生 1 次 cache miss。

对于数组 b，在第 1 次外层循环（i=0）中，每 4 次内部循环就会发生 1 次 cache miss，但由于一级缓存为 32KB，而整个 b 数组只占用 16KB（因为是一维数组），所以在第 1 次外层循环结束后，整个 b 数组都会缓存到一级缓存中，所以在之后的循环中不会再发生 cache miss。

对于数组 c，要在 4K*4=16K 次循环后（外部循环执行 4 次），才会发生 1 次 cache miss。

所以综上所述，数组 b 和 c 的 cache miss 可以忽略不计，只需计算数组 a 的 cache miss。

在此期间的浮点数运算次数为 4*2=8（4 次循环，每次循环里 1 次加法 1 次乘法）次。

在每 4 次循环中，cache miss 所用的时钟周期为 1*100 = 100 cycles，cache hit 所用的时钟周期数为（3*4-1）*1 = 11 cycles。

所以，最高实现性能 $= \frac{8}{(100+11)*\frac{1}{1\text{GHz}}} \approx 72.072\ MFlops$

三、 实验结果

第 1 题的 2 种方法的实验结果如下所示：

（1）一般算法

```
consthall@consthall-Lenovo-XiaoXin-CHAO7000-13:~/桌面$ g++ b.cpp -o ./b
consthall@consthall-Lenovo-XiaoXin-CHAO7000-13:~/桌面$ sudo perf stat -d ./b

 Performance counter stats for './b':

          702.04 msec task-clock                #    1.000 CPUs utilized
               0      context-switches          #    0.000 K/sec
               0      cpu-migrations            #    0.000 K/sec
             900      page-faults               #    0.001 M/sec
   2,375,852,309      cycles                    #    3.384 GHz                      (49.86%)
   6,839,164,257      instructions              #    2.88  insn per cycle           (62.39%)
     270,137,293      branches                  #  384.791 M/sec                    (62.39%)
         298,235      branch-misses             #    0.11% of all branches          (62.39%)
   2,955,763,143      L1-dcache-loads           # 4210.273 M/sec                    (62.51%)
     147,460,849      L1-dcache-load-misses     #    4.99% of all L1-dcache hits    (62.68%)
       4,964,103      LLC-loads                 #    7.071 M/sec                    (50.14%)
          44,729      LLC-load-misses           #    0.90% of all LL-cache hits     (50.03%)

      0.702327816 seconds time elapsed

      0.702328000 seconds user
      0.000000000 seconds sys


consthall@consthall-Lenovo-XiaoXin-CHAO7000-13:~/桌面$ sudo perf stat -e cache-misses -e instructions -e mem-loads:p ./b

 Performance counter stats for './b':

          238,471      cache-misses
    6,854,351,650      instructions
   <not supported>      mem-loads:p

      0.704137640 seconds time elapsed

      0.704143000 seconds user
      0.000000000 seconds sys
```

（2）Strassen 算法

```
consthall@consthall-Lenovo-XiaoXin-CHAO7000-13:~/桌面$ g++ strassen.cpp -o ./str
consthall@consthall-Lenovo-XiaoXin-CHAO7000-13:~/桌面$ sudo perf stat -d ./str

 Performance counter stats for './str':

          436.80 msec task-clock                #    0.999 CPUs utilized
               0      context-switches          #    0.000 K/sec
               0      cpu-migrations            #    0.000 K/sec
           9,243      page-faults               #    0.021 M/sec
   1,475,620,663      cycles                    #    3.378 GHz                      (48.72%)
   5,060,111,308      instructions              #    3.43  insn per cycle           (61.54%)
     212,131,531      branches                  #  485.654 M/sec                    (61.54%)
         102,101      branch-misses             #    0.05% of all branches          (62.30%)
   2,170,673,813      L1-dcache-loads           # 4969.542 M/sec                    (63.21%)
       3,806,948      L1-dcache-load-misses     #    0.18% of all L1-dcache hits    (64.10%)
         211,254      LLC-loads                 #    0.484 M/sec                    (50.52%)
          29,552      LLC-load-misses           #   13.99% of all LL-cache hits     (49.61%)

      0.437139363 seconds time elapsed

      0.437143000 seconds user
      0.000000000 seconds sys


consthall@consthall-Lenovo-XiaoXin-CHAO7000-13:~/桌面$ sudo perf stat -e cache-misses -e instructions -e mem-loads:p ./str

 Performance counter stats for './str':

         1,782,397      cache-misses
     5,086,356,846      instructions
   <not supported>      mem-loads:p

      0.436101559 seconds time elapsed

      0.416075000 seconds user
      0.020003000 seconds sys
```

由上面结果对比可知：

（1）CPI：图中给出的是 IPC，IPC 方面，一般 ＜Strassen，又因为 CPI 等于 IPC 的倒数，所以 CPI 方面，一般 ＞Strassen。

（2）周期数 cycles 和指令数 instructions：均为一般 ＞Strassen，很明显是由于 Strassen 的优化使得这两者均小于一般算法。

（3）Cache-misses：一般 ＜Strassen，应该是因为 Strassen 算法不断生成新数组导致 Cache-misses 增大。

（4）mem-loads：经过多种尝试方法（使用双系统或者在虚拟机上运行）均失败，始终为<not supported>。

（5）运行时间：一般 ＞Strassen，很明显是由于 Strassen 的优化使得其时间比一般算法更短。

四、遇到的问题及解决方法

正如第三题所说，我上网查阅了很多资料，并且也参考了课程群里大家和老师的建议，均未能解决 mem-load 始终显示为<not supported>的问题，导致无法分

析该指标，希望老师和助教可以在批改后在群里给出答案，或者给出一份优秀实验报告供我们学习参考。