

CS205 C/ C++ Program Design

Assignment 3

Name: 王奕童

SID: 11910104

Part 1. Description

【最终版本代码效率】2亿长度随机数向量点积用时：40~60ms

【基本要求实现】

1. 基本向量点积运算实现；
2. 输入错误时程序正常运行，并且提醒用户重新输入；
3. 阻止用户输入过大长度的向量，避免内存超限；
4. 用户输入计算完成后，程序可以继续重复利用而不直接退出；
5. 允许用户对输入的向量长度进行确认，以及允许用户选择随机数测试模式；
6. 修正float运算时结果不精确的问题；

【时间测量方法】

1. 毫秒时间测量方法介绍【本次报告时间测评方法】
2. 纳秒时间测量方法介绍【高精度时间测评方法】

【效率提升方法】

0. 第一个版本的时间说明
 1. 函数值传递改为引用传递
 2. 改进C++的IO效率
 3. O3优化编译选项
 4. 多线程并行计算
 5. 提升随机数的生成速度
 6. 数组存储运算结果再合并
 7. 空间使用与回收
 8. 编译模式由Debug模式转为Release模式
 9. 与标准库的效率比较
 10. 与其他语言的效率比较
 11. 关于运算与输出的一些思考

【说明】本次作业报告的测评环境：

运行系统：Windows10

编译器：mingw-GCC

内存限制：2048M

C++语言版本：C++20

CPU核数：8

运行IDE：CLion 2020.2

Part 2. Result & Verification

In this part, you should present the result of your program by listing the output of test cases and optionally add a screen-shot of the result.

Test case #1:

Input:

vector1=[1.2 -1.3 -3.56]

vector2=[1 0 -1]

Output: 4.760000

Screen-short for case #1:

```
Please input the size:
3
If you want to be a random test? If s
Q
Please input vector1[0]:
1.2
Please input vector1[1]:
-1.3
Please input vector1[2]:
-3.56
Please input vector2[0]:
1
Please input vector2[1]:
0
Please input vector2[2]:
-1
Data IO Time:21686ms
4.760000
0ms
4.760000
0ms
Do you want to continue calcula
Q
```

Test case #2:

Input:

vector1=[-.3 -.6]

vector2=[.6 .3]

Output: -0.360000

Screen-short for case #2:

```
Please input the size:
2
If you want to be a random test? If s
Q
Please input vector1[0]:
-.3
Please input vector1[1]:
-.6
Please input vector2[0]:
.6
Please input vector2[1]:
.3
Data IO Time:9800ms
-0.360000
0ms
-0.360000
0ms
Do you want to continue calculating?
Q
```

Part 3. Difficulties & Solutions, or others

【基本要求实现】

1. 基本向量点积运算实现

【运算规则】 向量点积的运算规则由作业文档给出：

$\text{dot_product} = v1[0]*v2[0]+v1[1]*v2[1]+\dots+v1[n-1]*v2[n-1];$

【设计思路】 先令用户输入向量长度存入全局变量vectorSize进行初始化， 然后按照作业文档中给出的方法， 建2个指针， 分别指向2个float类型的数组：

```
float *vector1= new float[vectorSize];
```

```
float *vector2= new float[vectorSize];
```

然后再建一个double全局变量result， 运算时遍历两数组进行运算即可

【函数实现】 源码中的函数：

```
inline void singleThreadSum();
```

【难易系数】★

【难点分析】无显著难点，只需要注意需要先让用户输入向量长度再进行计算即可。

（如果顺序错误，则会导致不能输入且计算结果为0。原因：C++的全局unsigned int变量在不初始化的情形下数值为0，会导致向量长度为0，没有给用户输入的机会，也不会进行任何的循环存储操作，直接将result输出，就结果为0）

2. 输入错误时程序正常运行，并且提醒用户重新输入

【检测规则】检测用户输入的内容不符合要求的情形，如输入向量的size的时候输入非整数字符，输入向量元素的时候包含其他字符（abc!@#\$%^&）

【设计思路】先输入string进行存储，然后根据输入的是向量长度还是向量元素进行string的合理性判断：建立无限循环，如果不合理则打印提示信息，让用户重新输入；合理则退出循环

※合理性判断规则：

向量长度：必须所有字符符合isdigit函数的检测，以判断是不是符合完全由数字组成

向量元素：允许首个字符为负号，允许出现一次小数点，其他字符要符合isdigit函数的检测，以判断输入是不是float的浮点数

【函数实现】由以下函数实现：

```
inline void cinSize();  
inline bool checkSizeMaterial(const string& sizeStr);  
inline void cinElements(int signNumber);  
inline bool checkElementMaterial(const string& elementStr);
```

【难易系数】★★

【难点分析】无显著难点，只需要在输出判断提示语句的时候，以及字符串合理检测的时候注意细节即可。

3.对输入向量的长度进行控制，避免内存超限异常退出

【控制规则】1个float数据占据内存4 bytes，在本机IDE环境下测试得到最大数组长度为999999999，再增加（哪怕1）都会异常退出。因此表征向量长度的字符串的长度不能超过10。

【设计思路】传入字符串，调用length函数返回字符的长度是不是大于10即可。

【函数实现】简单易行的一个函数即可搞定：

```
inline bool checkSizeLength(const string& sizeStr);
```

【难易系数】★★

【难点分析】需要反复的测试最大向量长度，而且测试的结果可能会因机器的不同而有所差异。

4.用户输入计算完成后，程序重复利用而不直接退出

【重复规则】计算完成后允许用户输入一次字符串，如果输入字符串为Y则重复执行

【设计思路】建立一个检测字符串并且初始化为"Y"，在运行结束后重新赋值检测字符串，以达到重复运行的目的。

【函数实现】main函数中有相应的实现

【难易系数】★

【难点分析】需要合理判断无限循环的位置，以及如何正确输出首次输入和再次输入的提示信息。

5.允许用户确认向量长度，以及允许用户选择随机数测试模式

【确认规则】输入“Y”以确认，否则重新输入向量长度或者自行输入数字检测

【设计思路】与4大同小异，增加一个全局bool变量以记录是否为随机数测试模式

【函数实现】以下函数即可实现：

```
inline void cinSize();  
void checkIsUserRandom();
```

【难易系数】★

【难点分析】需要合理判断无限循环的位置，以及如何正确输出首次输入和再次输入的提示信息，以及对于随机数模式的状态记录。

6.修正float运算时结果不精确的问题

【问题概述】在测试singleThreadSum()方法时，将vector1和vector2的长度设为200M（2亿），内部元素全部为1。根据线性代数的运算法则，vector1和vector2的点积运算结果应当为 $2E+8$ ，但是实际运行的结果是：

1.67772e+07

Process finished with exit code 0

与预期有较大的偏差。

【原因分析】这是源于float类型的存储精度有限。

float类型：4bytes，6-7位有效数字，数值范围： $1.4E-45 \sim 3.4E+38$

double类型：8bytes，15-16位有效数字，数值范围： $4.9E-324 \sim 1.7E+308$

在float的数字超过精度范围时，数值就会发生错误，如

```
1  #include <iostream>  
2  using namespace std;  
3  
4  int main(){  
5      cout.setf(ios::fixed,ios::floatfield);  
6      float re=123333333333.0f;  
7      cout<<re<<endl;  
8      return 0;  
9  }
```

main

SSETest x

C:\Users\16011\CLionProjects\SSETest\cmake-bui

123333337088.000000

Process finished with exit code 0

【难易系数】★

【难点分析】需要深刻理解float类型与double类型的差异，以及float与double之间运算精确度的规则。

【解决方案】改用double类型存储运算结果，即可回到正确运算结果。（运算过程中，两个float元素先提升类型为double，然后再进行运算赋给double变量。这是利用4bytes的空间换结果正确的典型例子）

2.0E8

Process finished with exit code 0

【参考链接】<https://blog.csdn.net/a10201516595/article/details/103330344>

【时间测量方法】

1. 毫秒时间测量方法介绍【本次报告时间测评方法】

【头文件要求】

```
#include <ctime>
```

【对应函数】

clock(), 返回类型为long

【使用方法】

开始前用一个long变量t1存储开始时间，结束时用另一个long变量t2存储结束时间，t2-t1即为运行所用的毫秒时间

【参考链接】 https://blog.csdn.net/qq_26836575/article/details/78488358

2. 纳秒时间测量方法2介绍【高精度时间测评方法】

【头文件要求】

```
#include <ctime>
```

【对应函数】

```
chrono::steady_clock::now().time_since_epoch().count()
```

返回类型为long long

【使用方法】

开始前用一个long long变量t1存储开始时间，结束时用另一个long long变量t2存储结束时间，t2-t1即为运行所用的纳秒时间

【参考链接】 https://blog.csdn.net/qq_31175231/article/details/77923212

【效率提升方法】

数据生成方法：现场生成随机数而不从文件读入，节约不必要的IO时间

0. 第一个版本的运行说明

【原始效果】

测试用例：2个2亿长度随机数向量

随机数生成时间：6500ms左右

计算时间：900ms左右

详情请见以下的截图：

```
200000000
```

```
Data IO Time:6573ms
```

```
934ms
```

```
5.36787e+16
```

```
Process finished with exit code 0
```

1. 函数值传递改为引用传递

【问题分析】在第一版中，C++的计算主函数有3个参数：2个float类型数组，1个int变量，然后有函数的返回值。

```
double singleThreadSum(float v1[],float v2[],int size);
```

在我理解的理论中，C++如果对函数进行值传递，则传递的并非对象本身，而是对象的copy。在针对大数据量的运算时，如果进行2个大长度数组的copy，则建立拷贝数组需要额外的时间和空间。

【解决方案】取消函数返回值，变更为void；将传入的对象使用&符号转化为引用，避免多余的对象拷贝。

【难度系数】☆☆☆

【难点分析】指针和引用的过程中容易出错，从而导致程序异常退出(exit 3)。

【优化效果】

测试用例：2个2亿长度随机数向量

计算时间：850ms左右

详情请见以下的截图：

```
2000000000
```

```
Data IO Time:6559ms
```

```
849ms
```

```
5.36843e+16
```

```
Process finished with exit code 0
```

【参考链接】 <https://blog.csdn.net/chenxuanhanhao/article/details/104238676>

2. 改进C++的IO效率

【问题分析】在c++的cin和cout中，默认有与stdio的绑定措施。cin和cout在输出时，需要先把内容存到缓冲区，然后再输出。cout的endl会在输出时刷新缓冲区，又造成了时间的浪费。

【解决方案】取消缓冲区，取消与stdio的绑定，换行操作使用\n取代endl

【代码实现】简易的由几句代码构成的函数：

```
inline void improveCIO();
```

【难度系数】☆☆

【难点分析】这是算法题目中常见的优化策略，无太大难度。

【优化效果】

测试用例：2个2亿长度随机数向量

计算时间：810ms左右

详情请见以下的截图：

```
2000000000
```

```
Data IO Time:6715ms
```

```
813ms
```

```
53684714856761816.000000
```

```
Process finished with exit code 0
```


【参考链接】

https://blog.csdn.net/weixin_34249678/article/details/93276743?utm_medium=distribute.pc_relevant_t0.none-task-blog-BlogCommendFromMachineLearnPai2-1.edu_weight&depth_1-utm_source=distribute.pc_relevant_t0.none-task-blog-BlogCommendFromMachineLearnPai2-1.ed

3. O3优化编译选项

【问题分析】之前所做的优化，都是基于O0（默认编译选项）进行的运行，并没有完全发挥gcc编译器的性能，导致运行时间较长。

【解决方案】开启gcc的O3编译选项（最大限度的优化），通过增加编译代码的代码量来换取运行时间的缩短。

【代码实现】增加一句代码即可调用gcc编译器的O3优化选项：

```
#pragma GCC optimize(3,"Ofast","inline")
```

【难度系数】☆☆☆

【难点分析】主要问题在难以想到这一点。想到了以后运行时间会大幅缩短。

【优化效果】

测试用例：2个2亿长度随机数向量

计算时间：120ms左右

详情请见以下的截图：

200000000

Data IO Time:6465ms

100ms

53683885313176128.000000

Process finished with exit code 0

【参考链接】https://blog.csdn.net/qq_41289920/article/details/82344586

4. 多线程并行计算

【问题分析】大长度向量进行运算时，时间复杂度为 $O(n)$ 。而事实上大长度向量点积的时候，各个部分之间的运算是互不干扰的，因此可以考虑采用多线程的并行计算策略。

【提示】`std::thread::hardware_concurrency()`函数可以得到本机的CPU核数，对于调整线程数目有帮助

【解决方案】建立thread类型的vector线程池，然后把多个长度的向量运算要求逐一分发下去，并且通过join()函数让主线程等待子线程运行完毕。

【代码实现】以下两个方法可以实现：

```
inline void multiplyThreadSum();  
inline void subMultiply  
(const int& start,const int& end);
```

【难度系数】☆☆☆☆☆

【难点分析】多线程有多个难点需要克服：

- ①多线程之间数据加锁，避免data race造成数据错误；
- ②多线程间每个线程之间的任务量的分配；
- ③主线程对于其他子线程的等待操作；
- ④子线程的合理数目的数量确定（少了不能发挥最大效益，多了会因为互相通讯而拖慢时间）；

⑤多线程在O3编译优化选项开启后才有显著优势，否则会慢于单线程；

⑥避免多线程之间加锁的时候错误，导致线程死锁。

【优化效果】

测试用例：2个2亿长度随机数向量

计算时间：70ms~80ms左右

详情请见以下的截图：

（上为单线程计算用时，下为多线程计算用时）

```
Please input the size:
200000000
If you want to be a random test? If so, please print[Y].
Y
Data IO Time:1770ms
15846812515182309908704179466321002496.000000
130ms
15846812515182902565697779606795321344.000000
70ms
Do you want to continue calculating? Print[Y] to continue.
Q

Process finished with exit code 0
```

【参考链接】 <https://www.cnblogs.com/douzujun/p/10810506.html>

5. 提升随机数的生成速度

【问题分析】本次报告的大数据测试用例都是通过现场生成的随机数。随机数生成可以调用rand()函数来生成伪随机数，调用srand()函数来设置随机数的种子。但是rand()函数的运行效率有待提高，如在第一版调用rand函数时需要6000+ms来生成2亿个随机数。

【解决方案】采用Xorshift算法，先只生成2个系统时间设置种子的伪随机数，后面通过一些算数运算来生成随机数，从而提高生成随机数的效率。

【代码实现】以下方法可以实现：

```
inline float randomGenerate();
```

【难度系数】☆☆☆

【难点分析】要自己设计一个比较随机的运算方法，以及要增加全局变量来保证下一次的伪随机生成。

【优化效果】

测试用例：2个2亿长度随机数向量

随机数生成时间：1750ms~1850ms

（计算时间为进一步优化过的，大约为45-60ms）

详情请见以下的截图：


```
Please input the size:
200000000
If you want to be a random test? If so, please print[Y].
Y
Data IO Time:1820ms
15847177506160162843001639704570363904.000000
130ms
15847177506161461493784428857004130304.000000
59ms
Do you want to continue calculating? Print[Y] to continue.
Q
```

【参考链接】 <https://blog.csdn.net/u012440684/article/details/50513423>

6. 数组存储运算结果再合并

【问题分析】之前的大数据运算是在线程加锁的情况下进行正确结果的计算的，一次只能有一个线程申请修改结果的数值。在有线程修改结果的时候，子线程必须等待直到锁释放，才能再进入运算。这一方面对效率有一定拖慢，另一方面还需要考虑线程死锁问题，让问题变得更加复杂。

【解决方案】事先申请一个等同于线程数量的double类型数组，然后每个线程单独修改数组的不同部分，最终算完后再将结果合并。这样可以避免data race的问题，以及线程锁的时候线程等待问题。

【代码实现】改进原先的多线程方法，在子线程方法中增加对于访问下标的引用传入：

```
inline void multiplyThreadSum();
inline void subMultiply
(const int& start,const int& end,const int& sign);
```

【难度系数】☆☆☆

【难点分析】需要了解数组是线程安全的，以及如何让不同的线程访问不同的部分。

【优化效果】

测试用例：2个2亿长度随机数向量

计算时间：45ms~60ms

详情请见以下的截图：

（上为单线程计算用时，下为多线程计算用时）

```

Please input the size:
200000000
If you want to be a random test? If so, please print[Y].
Y
Data IO Time:1750ms
15847260176754116701138785524783251456.000000
140ms
15847260176755037562602945105599922176.000000
50ms
Do you want to continue calculating? Print[Y] to continue.
Q

```

Process finished with exit code 0

【参考链接】 <https://ask.csdn.net/questions/645691?ref=myrecommend>

7. 空间使用与回收

【问题分析】之前提出过循环计算功能。而在初始化时，都需要开辟一个新的内存空间。如果先前的内存空间不进行释放，则容易导致内存泄漏，程序异常退出等等。

【解决方案】在下面函数中，每次都有`delete []`对应数组的指针即可。

【代码实现】请见

```
void initiate();
```

【难度系数】★

【难点分析】这是于仕琪老师Lecture5上课所讲的东西。因此直接应用难度不是很大。

【优化效果】

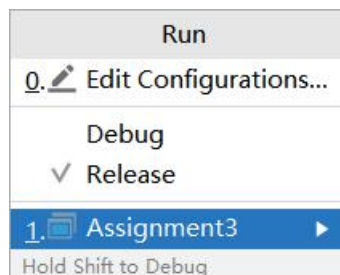
无时间上的明显优化，但是实现了在一次运算完后对内存进行释放，提高了系统空间的利用率，有效避免了内存泄漏问题，保证了程序能够正常地循环利用。

8. 编译模式由Debug模式转为Release模式

【问题分析】在IDE的debug选项状态下，c++的程序运行速度很慢，没有发挥出潜在的潜能。切换编译模式可以进一步提高效率。

【解决方案】切换IDE的编译选项由Debug转为Release。

【具体操作】点击CLion中上方的run选项，然后点击Run（第4个），然后在弹出的菜单里勾选release即可。



【难度系数】★★

【难点分析】需要知道Debug模式和Release模式的区别，以及在IDE中如何切换这两种模式。

【优化效果】

测试用例：2个2亿长度随机数向量

计算时间: 40ms~60ms (提升了运行效率的上界)

详情请见以下的截图:

(上为单线程计算用时, 下为多线程计算用时)

```
C:\Users\16011\CLionProjects\Assignment3\cmake-build-release\Assignment3.exe
Please input the size:
200000000
If you want to be a random test? If so, please print[Y].
Y
Data IO Time:1678ms
15846077461424542590616414072212553728.000000
131ms
15846077461424641760312554334762041344.000000
42ms
Do you want to continue calculating? Print[Y] to continue.
```

9. 与标准库的效率比较

【CBLAS】cblas是一个openBLAS库在C/C++语言的接口, 可以实现大长度的向量点积的高性能运算。

测试用例: 2个2亿长度, 数字均为2的大长度向量

Cblas计算时间: 47~49ms (12核linux系统下)

本方法计算时间: 40~60ms (8核windows10系统下)

运行截图:

【CBLAS】

```
[bill@localhost Openblas]$ ./a.out
800000000.000000
thw difference is 0.048693s
[bill@localhost Openblas]$
```

(感谢提供linux系统测试的11913008谢岳臻同学)

【本方法】

Please input the size:

200000000

If you want to be a random test? If so, please print[Y].

Y

Data IO Time:768ms

800000000.000000

132ms

800000000.000000

47ms

Do you want to continue calculating? Print[Y] to continue.

q

Process finished with exit code 0

10. 与其他语言的效率比较

【Java】IDE: IDEA Ultimate 2020.2

Java的代码块为了精简，取消了对输入错误的检测，只保留了随机数检查的功能（代码会连同报告一起上交）

【代码整体结构】

```
import java.io.*;
import java.security.SecureRandom;

public class Main {
    private static Reader in;
    private static PrintWriter out;
    private static final int thread_Number=Runtime.getRuntime().availableProcessors();
    private static float[]vector1;
    private static float[]vector2;
    private static double[] results;
    private static double result;
    private static int size;
    private static long xp,yp,zp;

    public static void main(String[] args) throws InterruptedException {...}
    private static void improveJavaIO(){...}
    private static float randomGenerate(){...}
    private static void javaInSize(){...}
    private static void initiate(){...}
    private static void javaInElements(int signNumber){...}
    private static void singleThreadMethod(){...}
    private static void multiplyThreadMethod()throws InterruptedException{...}

    static class MyThread implements Runnable{...}
    static class Reader {...}
}
```

采用了快读快写模板，多线程与单线程对比，以及安全随机数的生成。为精简，Java只添加了大量随机数生成的选项。

【Python】IDE: PyCharm Professional 2020.1

【代码整体结构】

```

import ...

import psutil as psutil

vector1 = []
vector2 = []
vectorSize = 0
result = 0
xp = 0
yp = 0
zp = 0
threadNumber = psutil.cpu_count()
results = [0 for _ in range(0, threadNumber)]

class myThread(threading.Thread):...
def randomGenerate():...
def pythonInSize():...
def initiate():...
def pythonInElements(sign):...
def singleThreadMethod():...
def numpyMethod():...
def subMutiply(start, end, sign):...
def mutipleThreadMethod():...
if __name__ == '__main__':...

```

同Java，有单线程与多线程的运算对比，以及随机数的生成法则。Python中有关于非法输入的检测，不过不在此赘述。测试中只考虑了随机数的生成测试用例。

【运行情况】

①生成随机数：(随机数算法：XorShift算法)

【C++】 1750-1850ms

【Java】 750ms~850ms

【Python】 87000~90000ms

②单线程运行：

【C++】 100ms~140ms

【Java】 350ms~450ms

【Python】 24000ms~25000ms

③多线程运行：

【C++】 45ms~55ms

【Java】 70ms~90ms

【Python】 20000ms~22000ms

测试运行的截图：

【C++】 ↓

C:\Users\16011\CLionProjects\Assignment3\cmake-build-debug\Assignment3.exe

Please input the size:

200000000 ① 向量长度

If you want to be a random test? If so, please print[Y].

y ② 确认选择随机数

Data IO Time 1750ms ③ 随机数生成时间

15847260176754116701138785524783251456.000000 ④

140ms

15847260176755037562602945105599922176.000000 ⑤

50ms

Do you want to continue calculating? Print[Y] to continue.

Q ⑥ 退出运行

Process finished with exit code 0

【Java】 ↓

"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...

200000000 ① 向量长度

Data IO Times: 829ms ② 随机数生成

893221009513173700000000.0000

371ms

893221009513174000000000.0000 ③

99ms

④ 多线程运行结果

Process finished with exit code 0

【Python】 ↓

C:\Users\16011\PycharmProjects\TestForCpp\venv\Scripts\python.exe

Please input the size:

200000000 ① 向量长度

Data IO Time: 87.4375 s ② 随机数生成时间

51791039104 ③ 单线程运算

25.265625 s

251431552 ④ np.dot()运算

35.171875 s

51791039104 ⑤ 多线程运算

21.75 s

Process finished with exit code 0

【结果&分析】

【结果】在这次运行测试中，C++的性能优于Java，远优于Python。

【分析】

①C++快于Java/Python的原因：C++是编译型语言、而Java和Python都是解释型语言。编译型语言是用编译器将所有代码编译成可执行机器语言然后直接运行可执行文件即

可；而解释型语言是解释器将程序一行一行翻译成机器语言执行。因此编译型语言(C/C++)相对于解释型语言(Java/Python)有更多运行速度上的优势；

②Python比C++/Java慢很多的原因：Python是动态编译语言，在编译时并不能完全确定代码内的运行对象类型，需要有更多的步骤来实现某些功能，而Java和Python是静态编译语言，在编译时即可确定对象类型，节约了时间。另外，Python的float类型精度高于C++/Java的float类型，与double精度一致，也在一定程度上造成了空间的开销。

11. 关于运算与输出的一些现象与思考

【现象1】

在较为初级的版本的测试过程中，如果取消对于运算结果的输出而仅仅进行一个运算的操作，则用时大大缩短，甚至远远小于现在最短的时间。

测试用例：2亿长度，元素全为1的大长度向量

取消运算结果输出操作前：

```
C:\Users\16011\CLionProjects\SSETest\cr
Data IO Time:639ms
141ms
200000000.000000
```

```
Process finished with exit code 0
```

取消运算结果输出操作后：

```
C:\Users\16011\CLionProjects\SSETest\cmake
Data IO Time:790ms
1ms
```

```
Process finished with exit code 0
```

【思考1】


这个原因是因为最终有一个output操作。本身2亿长度的计算在CPU中做一连串的计算的速度是非常快的。但是如果最终需要进行一次output操作，则需要进行CPU与总线之间的通讯操作。然而总线的运行速度是比CPU的速度要慢很多，比单纯的CPU运算慢了好几个数量级。所以一句printf或者cout操作会对时间有很大的影响。

【结论】要想运行速度提高，不仅要从算法层面优化，更要留心IO操作。

【指导来源】南方科技大学计算机系助理教授刘烨庞老师

【现象2】

在文本文件存储的过程中，使用十进制ofstream直接存储float的运算结果会导致IO时间大幅增加，而且生成的文本文件非常大，如下所示：




out.txt

文件类型:

文本文档 (.txt)

打开方式:

 记事本

更改(C)...

位置:

L:\temp\新建文件夹

大小:

3.56 GB (3,832,179,084 字节)

占用空间:

3.56 GB (3,832,180,736 字节)

【思考2】这个是源于存储方式的不同。float类型本身为4字节二进制数，因此如果改为使用二进制即可正确存储，并且规模较小，节约空间。（本次报告测评为节约IO时间，没有采纳从文本文件读入的方式生成随机数的方法）


1.in 属性

常规

安全

详细信息

以前的版本




1.in

文件类型:

IN 文件 (.in)

打开方式:

 记事本

更改(C)...

位置:

C:\Users\16011\CLionProjects\Assignment3\cmake

大小:

762 MB (800,000,000 字节)

占用空间:

762 MB (800,002,048 字节)

【参考链接】

https://blog.csdn.net/computerme/article/details/72904932?utm_source=blogxgwz9

Part 4.Summary

本次Assignment3题目要求明确，在做作业的过程中学习了很多的知识点：

1. C/C++的指针和内存的使用与回收
2. 对于用户输入错误的检测与反馈
3. 高效随机数生成的XorShift算法
4. C/C++的程序计时方法（毫秒级/纳秒级）
5. C/C++的IO效率的提升策略
6. C/C++函数值传递与引用传递的区别与联系
7. C/C++的优化编译选项
8. C/C++多线程
9. C/C++的编译模式的区别
10. C/C++调用Python文件里的函数
11. Java多线程
12. Python基本语法规则、Python多线程
13. Python使用标准库
14. 关于编译型语言与解释型语言的区别与联系
15. 二进制精确存储数据的方法
16. 与标准库之间的比较与学习