ICS 课程报告

汪俊轩 21307130169

选题：题号 1，优化程序性能

附件介绍：

1. 题目.pdf　所选程序对应的题目
2. 运行结果截图.pdf　部分代码运行之后的结果截图，记录了运行时间信息
3. sourse.cpp　原始程序
4. sourse(x).cpp　各个版本的优化程序，共 5 个
5. test.cpp　为原始程序在各个测试点计时的程序
6. test(x).cpp：为优化程序计时的程序，共 6 个

设计：

1. **实验环境：**

   Ubuntu on windows

2. **编译器：**

   g++11.0

3. **选题：**

   本选题要求自选一个 c/c++程序，综合运用这学期所学的知识，尝试优化程序的性能。程序的性能在程序能够拥有准确输出的情况下可以用程序所需的时间和空间的多少来表示。在本课程中，与之相关性最大的就是程序性能的优化这一章节。因此，本课程报告基于 CS：APP 原书第五章——优化程序性能中的相关知识，选择了一个数据结构课程中自己写的程序进行优化，记录优化的过程并尝试分析每一步的原因。

4. **优化方向：**

   通过课程所学与阅读 CS：APP 第五章的内容，可以发现，想要优化程序性能，可以从六个方面入手：

   (1)消除循环的低效率，通过代码移动等方法

   (2)减少过程调用，如函数调用、封装类的使用等

   (3)消除不必要的内存引用，使用临时变量代替一些内存值

   (4)提高代码的并行性，有循环展开、使用多个累计变量等手段

   (5)重新结合变换

   (6)提升加载和储存的性能，通过消除写/读相关等方式

   除了这些之外，在其他章节中也能寻到一些可能可以提升程序性能的方

式，如第六章存储器层次结构中，介绍了利用好高速缓存提升代码的性能。充分利用 cache 的性能，保证 cache 命中，也可以提升程序的性能。还有在程序中利用局部性，等等。

实验：

1. 记录原始程序的性能

记录未优化时的各个测试点的运行时间（单位：ms），如下表

程序	方法	N,M=4,8	N,M=7,7	N,M=8,8	N,M=6,8
sourse.cpp	Abstract unoptimized	60.951	968.732	1163.08	26935.602
sourse.cpp	Abstract -O1	9.573	150.199	187.519	4050.243
sourse.cpp	-Ofast -march=native	8.604	124.775	155.96	3342.038

使用了-O1 的优化之后，各个测试点的运行时间均缩短了 5/6 以上。这说明，通过测试数据可以发现，在没有修改程序源代码的情况下，单纯靠编译器的基础优化就能够显著提高程序的性能。

2. 考虑通过减少过程调用来提升程序的性能

观察程序可以发现，程序中大量使用了 vector 这一标准库容器，由于标准库容器 vector 除了实现数组的顺序存储的功能之外，还额外实现了其他功能，例如统计长度，变长储存等功能，相当于变相额外调用了函数。因此，我们将 vector 改用 c 语言中的数组实现。同时，由于编译器并不知道 n 和 m 的范围，而我们限制输入的 n 和 m 的范围为小于等于 8，所以我们可以在程序开始之前手动分配一个 8*8 的二维数组用于储存棋盘，以减少临时分配空间产生的时间代价。经过上述所说的修改，得到的程序为 sourse1.cpp，相应的为其计算各个测试点的时间的程序为 test1.cpp。

修改完成之后，再次观察程序，发现在取缔了 vector 之后，还有一个标准库容器 stack 仍被使用。由于我们只利用到了栈先进先出的功能，因此，同样使用数组替代栈的使用，再另设变量 top 保存栈中元素的个数就可以了。

实现了上述的修改之后，得到的程序为 sourse2.cpp，相应的为其计算各个测试点的时间的程序为 test2.cpp。

完成这些优化后，在 Ubuntu 中执行以下命令：

g++ test1.cpp

./a.out

g++ test2.cpp

./a.out

得到 sourse1.cpp 和 sourse2.cpp 在各个测试点的运行时间（单位：ms），如下表所示

程序	方法	N,M=4,8	N,M=7,7	N,M=8,8	N,M=6,8
sourse1.cpp（test1.cpp）	Abstract unoptimized	42.268	615.637	726.408	15553.915
sourse2.cpp（test2.cpp）	Abstrack unoptimized	14.195	226.437	279.639	5952.698

可以看到，用数组替代 vector 之后，各个测试点的运行时间相比于最原始的程序减少了 1/3 左右，而用数组替代了 vector 与 stack 之后，各个测试点的运行时间相比于最原始的程序竟减少了有大概 3/4 左右之多，可见限制原始程序速度的一个主要因素就是反反复复调用标准库容器中的各种函数带来的时间开销。因此，在需要考虑运行速度的程序的编程时，尽量手动实现 vector，stack 等方便实现的数据结构。

3. **考虑通过消除不必要的内存引用来提升程序性能。**

在完成了减少过程调用的部分之后，观察代码，可以发现代码中出现了多次访存的情况。如多次出现 R[next]与 L[next]，每次访存都需要计算内存地址，再访问内存值。如果某个内存值在程序中较接近的地方多次出现，则可以将其保存到寄存器中，具体操作是使用局部变量保存它，因此下一步我们消除程序中的不必要的内存引用，通过设置局部变量 rn 和 ln 保存下一个下棋的位置。

优化完成之后得到的程序为 sourse3.cpp，相应的计算时间的程序为 test3.cpp。

完成这些优化后，在 Ubuntu 中执行以下命令：

g++ test3.cpp

./a.out

得到 sourse3.cpp 在各个测试点的运行时间（单位：ms），如下表所示

程序	方法	N,M=4,8	N,M=7,7	N,M=8,8	N,M=6,8
sourse3.cpp（test3.cpp）	Abstract unoptimized	14.503	226.017	272.205	5985.893

比较 sourse3.cpp 和 sourse2.cpp 在各个测试点的运行时间，发现两者十分接近，微小的误查可以认为是误查。完成消除不必要内存引用的优化之后运行时间并没有显著缩短。说明该部分对于代码运行时间的影响较小，不是关键因素。可能的原因是代码中出现的重复内存引用的次数仅为 3 次，相当于把三次内存引用减少为了 1 次，在该部分上的时间缩减了 2/3，对该部分的时间有着明显地减少。然而该部分在总时间中的占比不大，导致这部分时间的缩减没有在总运行时间上体现出来。

4. **考虑通过消除循环的低效率来提升程序性能**

在不利用到微处理器的特性之前，还剩下一个消除低效循环的优化没有做。观察程序发现，判断目标点是否已经到过的判断条件作为循环条件出现，而其中的判断数目非常之多。因此可以考虑简化该循环判断条件以提升程序的性能。此处选择的做法是将棋盘外的点设置为“已走过”的状态，这样就不用判断所考虑的点是否在棋盘上，因此简化了循环的判断条件。同时，在初始化棋盘的时候需要设置边界线外距离棋盘边界为 2 的所有点的值。

优化完成之后得到的程序为 sourse4.cpp，相应的计算时间的程序为 test4.cpp。

完成这些优化后，在 Ubuntu 中执行以下命令：

```
g++ test4.cpp
./a.out
```

得到 sourse4.cpp 在各个测试点的运行时间（单位：ms），如下表所示

程序	方法	N,M=4,8	N,M=7,7	N,M=8,8	N,M=6,8
sourse4.cpp（test4.cpp）	Abstract unoptimized	13.677	210.339	260.461	5542.265

将其与 sourse3.cpp 的各个测试点的进行比较，发现各个测试点运行时间有明显减少，说明我们的优化起到了效果。虽然我们只修改了循环的一个判断条件，但是因为循环进行的次数是非常多的，稍微减少一点单个循环体

的运行市场都能够是整体运行时间产生非常明显的缩短。

5. **考虑通过利用循环展开来提升程序的性能**

循环展开某种程度上也是利用了指令的并行，利用处理器的微体系结构提升代码速度。想要进行循环展开，需要符合条件的循环，如两次循环方便分割，后一次循环的进行可以不依赖于前一次循环的进行。观察代码发现，程序运行时间的主要产生部分——搜索-回溯部分并不满足条件，因为每一次循环的进行都依赖于上一次循环的结果。因此，只剩下一些诸如棋盘初始化，棋盘输出等部分的循环适合使用循环展开提升程序性能，然而他们对于整体的程序运行时间的产生的影响较小。因此，站在优化程序性能的角度（如该程序可能被复用到一个非常大的棋盘上），我们做出此部分优化，但不作为一个新的版本重新程序测量时间，而是只测量初始化棋盘的时间，为了消除由于误查带来的影响，使得程序运行的时间超过计时的分度值，将初始化棋盘在一次运行中使其运行 1000 次，并分优化前与优化后计时。两种初始化方法都在 test5.cpp 代码中，以注释形式表述。原形式用 a 表示，优化后的用 b 表示。

分别将使用 a 方法初始化和 b 方法初始化的代码部分解开注释，在 Ubuntu 中执行以下命令：

```
g++ test5.cpp
./a.out
```

得到 test5.cpp 在各个测试点初始化棋盘 1000 次的运行时间(单位:ms)，如下表所示

程序	方法	N,M=4,8	N,M=7,7	N,M=8,8	N,M=6,8
test5（a）.cpp	Abstract unoptimized	0.161	0.187	0.217	0.183
test5（b）.cpp	Abstrack unoptimized	0.102	0.129	0.152	0.127

可见，优化后的代码在初始化时的速度明显更快。

6. **下面，尝试利用提升代码并行性的方法提升程序的性能**

观察代码发现，sourse 实现的是一个类似深度优先搜索的过程，即走棋有优先级，求解时必须按照题目要求的顺序移动，否则可能无法保证程序的

正确性。因此，sourse 部分难以使用提升代码并行性的方法来提升程序性能。然而，在各 test 代码中，输出测试数据和对应时间的部分还可以提升并行性，虽然这种做法不会导致程序运行的时间变快，但是会导致输出结果的速度变快，因此，我们增加了一个测量项为输出最终结果所需要的时间，同时在 test4 中也增加了此项方便比对。修改之后的测试程序为 test6.cpp

完成该优化后，计算输出统计表所用时间（单位：ms），如下表

程序	方法	输出所用时间
test4.cpp	Abstract unoptimized	0.038
test6.cpp	Abstract unoptimized	0.026

由此可见，输出部分的效率得到了明显提高，说明当某些部分的代码可以并行时，需要在自己的程序中将它们的并行性体现出来，使其不再受到延迟界限的限制。

7. **考虑通过重新结合变换来提升程序的性能**

由于程序中没有存在重复执行了非常多次的多个元素运算，所以重新结合变换的做法在优化本程序时难以发挥。

8. **考虑通过提升加载和缓存的性能来提升程序的性能**

本程序中虽然在搜索-回溯的时候存在写/读相关，但是要寻找到最优先的走棋方法必须要经历查找后回溯的过程，不可避免，所以写/读相关也难以被消除，若选择使用多个棋盘强行消除，则会因为复制棋盘等操作付出更大的时空代价。

评价标准：

在搜索-回溯的循环主体部分做出的优化则直接使用整个程序的运行时间来衡量效果

在其余部分做出的优化则只计算优化部分的运行时间（如某个循环），因为搜索-回溯占用的时间几乎占了程序运行总时间的全部，因此如果考虑程序运行的总时间则会变化不显