

**课 程 实 验 报 告**

**课程名称： 并行编程原理与实践**

**专业班级： CS1602**

**学 号： U201614545**

**姓 名： 谭胜克**

**指导教师： 金海**

**报告日期： 2019/7/29**

**计算机科学与技术学院**

**目录**

[1 回溯法解决Akari问题 1](#_Toc15412454)

[1.1实验目的与要求 1](#_Toc15412455)

[1.2问题描述 1](#_Toc15412456)

[1.3算法描述 2](#_Toc15412457)

[1.4实验方案 3](#_Toc15412458)

[1.5实验结果与分析 4](#_Toc15412459)

[2 并行回溯法求解Akari问题 5](#_Toc15412460)

[2.1实验目的与要求 5](#_Toc15412461)

[2.2问题描述 5](#_Toc15412462)

[2.3算法描述 5](#_Toc15412463)

[2.4 实验方案 7](#_Toc15412464)

[2.5 实验结果与分析 7](#_Toc15412465)

[3 改进的并行回溯法求解Akari问题 8](#_Toc15412466)

[3.1实验目的与要求 8](#_Toc15412467)

[3.2问题描述 8](#_Toc15412468)

[3.3算法描述 9](#_Toc15412469)

[3.4实验方案 9](#_Toc15412470)

[3.5实验结果与分析 9](#_Toc15412471)

[4 实验小结 10](#_Toc15412472)

[**附件：源代码** 11](#_Toc15412473)

# 1 回溯法解决Akari问题

## 1.1实验目的与要求

1. 了解Akari问题，探索其解决方法。
2. 了解回溯法的概念，掌握回溯法的原理，并能够使用一种编程语言使用回溯法解决问题。
3. 培养和锻炼分析问题与解决问题的能力。

## 1.2问题描述

* Akari问题描述

Akari问题有时又被称为Light up或者Beleuchtung，源于日本逻辑解密游戏系列Nikoli，同属于Nikoli谜题的除Akari之外还有Sudoku(数独)和Kakuro(数谜)等。

游戏规则很简单。点灯游戏的棋盘是一张方形格网，其中的格子可能是黑色也可能是白色。游戏目标是在格网中放置灯泡，使之能照亮所有的白色方格。如果一个方格所在的同一行或同一列有一个灯泡，并且方格和灯泡之间没有黑色格子阻挡，那么这个方格将被灯泡照亮。同时，放置的每个灯泡不能被另一个灯泡照亮。某些黑色格子中标有数字。这些数字表示在该格子四周相邻的格子中共有多少个灯泡

* 回溯法

回溯法是一种迭代算法。在回溯法中，首先将问题的解决分为若干步，其次通过枚举每一步的选择构造解空间树。在此基础上通过深度优先搜索遍历此解空间树，若当前节点的局部解不能构造出全局解，则向上回溯，否则向下扩展。重复此步骤直到找到全局解。

回溯法的关键点在于：

* 问题可分步并且可枚举每一步的选择
* 可以迅速的判断出当前局部解可否构造出全局解
* 编程要求

编程实现使用回溯法解决Akari问题。要求首先对问题进行划分，并根据每一步构造解空间，进而设计合理的数据结构与程序结构对问题进行求解，并在此基础上对算法以及程序逻辑进行优化和改进，最后进行问题的分析、讨论和展望。

* 格式说明

输入数据由若干文件组成，每个文件描述一个Akari问题的初始状态，编写程序读入此文件，根据初始状态求解。有的有一个解，有的有多个解，我们保证有解。文件由若干行组成，第一行为两个整数 n，m，代表棋盘的行数和列数。之后的 n 行每行有 m 个整数表示棋盘的每个格子的状态，若它为 -2，则表示是白格子，若它为 -1，则表示是没有数字的黑格子，若它为 0-4，则表示是数字 0-4 的黑格子。若你想把灯泡放在白色格子上面，则需要将 -2 改为 5，因为 5 表示有灯泡的格子。

## 1.3算法描述

1. 初始构想

在一开始接触Akari问题时，看到回溯法，首先就想到了8皇后问题的求解方式：从第一行第一列开始摆放皇后，逐行逐列进行尝试。若是在尝试过程中能够判断出当前方案下不可能得出解，则进行回溯，回到上一次分支开始的地方。重新进入下一个分支。

在实际执行时发现Akari问题如果采用逐行逐列进行尝试的回溯法，这个过程远比8皇后问题复杂。其主要难点就在于如何进行裁支工作，由于存在黑色方块的干扰，使得裁支难以进行，而必须等到整个矩阵遍历结束之后才能够判断。可以想象，这与遍历法解决相差无几，而且在执行过程中的判断算法还非常复杂。思考过后放弃了这个方案。

1. 选定方案

放弃了初始的想法之后，一直没有想到什么更好的方案去解决问题。某次登录网站认真查看了题目给出的编程要求之后，茅塞顿开。

将问题分为对黑块的处理和对白块的处理。对黑块的处理就是根据黑块中的数字，在其周围四个方向（上、下、左、右）填上相应个数的灯泡，那么在不同数字的黑块周围灯泡的摆放就存在不同数量的方案，我们应该做的就是穷举这些方案，若是在某个方案的某一步发现没有必要进行，则进行回溯。所以在每次分支前都需要将原矩阵进行保存，在回溯时就能够通过原始矩阵尝试下一种摆放方案。

而对于白块的处理则是放在所有的黑块处理完成之后，当所有的黑色方块都能够按照其中的数字，在其周围摆着灯泡，这时候仍然有可能存在未被一些死角，没有被灯光所覆盖。所以应该尝试着在这些空白位置放置灯泡。当没有空白位置，但是矩阵并不符合最终的要求时，本次方案失败，开始回溯，重新回到黑块解决阶段。

1. 开始实施

黑块处理阶段首要任务是如何将黑块提取出来并对黑块进行管理，这里用的是链表结构对黑块进行存储，每一个链表结点存储黑块中的数字，黑块的行、列，以及下一个黑块指针。链表结构如图1.1所示。按照黑色方块的数字顺序分别对矩阵进行便利，构造黑色方块链表。



图1.1 链表结点结构

接下来就是开始穷举方案，尝试在黑色方块周围填入灯泡。比如黑色方中的数字为4，那么只存在一种摆放方案，即上下左右均放置；若是黑色方块中的数字为2，则存在6只种灯泡的放置方案。上左、上下、上右、左下、左右、下右的格子中放置“灯泡”。其它数字的方块依次类推。在每一中方案开始之前，都将当前矩阵进行一次保存，就是将复制过后的数组传入递归中进行处理，使得在回溯时能够恢复到当前状态。当黑块链表处理完成，就进入到剩余白块的处理阶段。

白块的处理相对简单，找出当前所有的空白方块，按顺序往上摆放灯泡，期间每放置一盏灯泡就判断一次是否完成，未完成则继续，完成则返回。若是不存在下一个白块，并且没有完成，则说明此种方案不可行，需要回溯到黑块处理。

在黑块处理中，用到了两个功能函数：一个是放置灯泡的函数：put\_blub；另一个是放置断点标签的函数：put\_dot。put\_blub在传入的行、列位置放置一盏灯泡，成功返回1，失败返回0，同时若成功放置一盏灯泡，则将其周围能够点亮的方块点亮。put\_dot主要是标记出那些不应该存在灯泡的方块，这样在接下来的白块处理阶段就可以省略很多时间。

最终还需要一个检验函数，判断我们当前的状态下结果是否是正确的。按照算法思路以及题目要求，这里的判断函数首先要判断矩阵中是否存在空白位置，也就是没有被灯光覆盖到的当块，或者是存在断点；另一个，要判断所有黑色方块周围的灯泡数量数否与其中的数字相符。至于灯泡不能摆放在同一列、同一行是不用进行检测的，这一步的检测在灯泡放置阶段已经进行，不可能出现如此问题。

## 1.4实验方案

1. 在本地编写调试代码
2. 修改代码用于平台测试

在本地测试时，用的是从文件获取输入矩阵的方式进行输入，而平台给出的调用函数给的是二维的vector，所以要将相应的输入方式进行修改。

1. 进行检测

第一次检测时，7个测试集并没有完全通过，而是仅仅通过了4个，剩余3个测试集，分别是4、6、7未能通过测试。4、6测试集通过算法给出的提示为无解，而7的提示是黑色方块周围的灯泡个数不符合。

1. 修改错误测试集7

首先认真观察了第7项测试集给出的结果矩阵，发现确实存在一个黑色方块周围的灯泡数量与其中的数字不和。但是按照黑块的处理方式并不应该出现这种情况，最后我想了想，既然是结果不符合，那么修改检测函数就可以，于是在检测函数中加入了黑块数量检测的内容，再次测试，通过了第7项测试集。

1. 修改错误测试集4、6

4号、6号测试集给出的提示都为无解。这就让我很难定位错误的存在。想了好久想到会不会是黑块之间相互影响了，就是我在构造黑色方块链表的时候采用的是一次遍历的方法，只要遇到黑色方块就将其插入链表尾部，而没有区分黑块数字。那么很有可能在处理某一个方块时，将其周围的某一位置打上了断点，使得接下来的某一黑块不能按照其中的数字摆放灯泡，从而导致了无解。想到了这一点，我对黑块链表的构造函数稍加修改，改为了多次遍历，按照数字顺序分别插入队列。再次检测，问题解决，通过了第4、第6项测试，完美通关。

## 1.5实验结果与分析

如图1.2所示，为Akari问题在平台的测试结果。7项测试均成功通过，说明算法能够正确解决Akari问题。



图1.1 链表结点结构

在实验过程中发现，对于黑色方块不同的处理顺序，程序的耗时也会不一样。在测试过程中发现最优化的方案是首先处理可能性最少的黑色方块，也就所说处理顺序为：0、4、3、1、2，其中某些可行方案个数相等的可以交换。当我更换处理顺序的时候，在数据规模变大时（14\*14），整体运行误差可以达到秒级。

还有一个就是检验子函数在整个程序的运行耗时中占据重要地位。一开始我的检测字函数中没有加入黑色方块周边灯泡数量的判定过程，处理14\*14的矩阵整体耗时不到0.05秒，当添加上黑色方块周边灯泡数量的判定过程之后，整体耗时提升到1.5秒左右。

# 2 并行回溯法求解Akari问题

## 2.1实验目的与要求

1. 掌握串行程序并行化的方法。
2. 熟悉在 Linux的C++语言环境下进行并行程序设计的方法。
3. 了解并行程序设计与并行算法设计的基本分析方法与途径。

## 2.2问题描述

* Akari串行回溯算法的并行化

串行算法并行化的过程中的核心问题是问题分解和解除数据相关的问题。所谓的问题分解问题即将串行算法采取分治的理念分成可并行计算的子问题，解除数据相关的问题就是采取一定的冗余策略或锁策略使各子问题的私有数据之间不相互影响。

回溯法是易于并行化的典型算法之一，在Akari问题的回溯算法中，可以清楚的认识到回溯的具体路线的选择对产生解的正确性没有影响，因此该回溯法中每个节点的计算过程以及节点的分裂过程均是可以并行计算的。同时由算法的性质决定了每个解空间节点各自数据不会相互影响，没有任何数据相关。

* Linux下C++环境中并行编程方法
* 使用pthread进行多线程编程
* 使用fork()进行多进程编程
* 编程要求

1. 确定并行粒度

确定并行粒度也就是把算法中可并行执行的单元映射到不同task的过程。

1. 使用并行程序设计方法实现算法

使用pthread将task映射到thread或者使用fork将task映射到不同的进程。设计新的数据结构来适应并行化的编程需要。

1. 测量时间并计算加速比

运行新的并行程序，测量程序运行时间，并与第一关中的串行运行时间相比较，计算出并行程序的加速比。

* 格式说明

同第一节串行回溯法相同。

## 2.3算法描述

对串行回溯法进行修改，主要修改黑色方块处理部分的核心代码。将各个枚举方案从串行处理改为并行处理。这里是根据黑色方块中的数字所能够形成的组合去创建不同数量的线程。比如当黑色方块中的数字为3时，说明存在4种方案，所以会创建4个线程去执行之后的枚举过程。

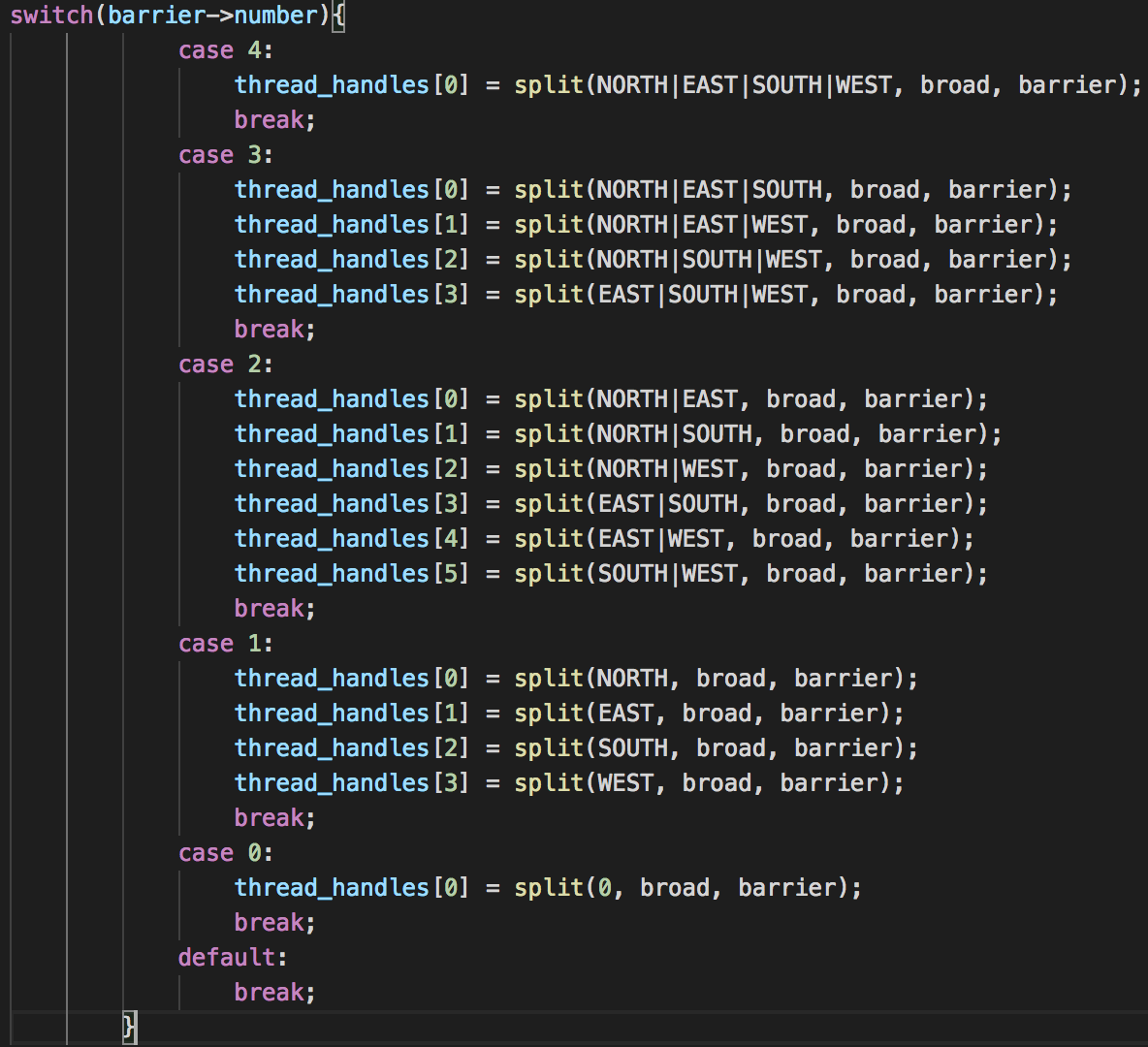


图2.1 并行化核心代码

在split函数中，根据黑色方块链表尝试在黑色方块周围放置断点或者是灯泡，如果操作成功，则进入下一个分支，也即子线程重新调用核心递归代码，再次申请子线程。为了记录每个子线程当前处理的数据状态，这里通过结构的方式进行保存。对每一个线程当前的矩阵状态、当前的黑色方块队列状态进行保存。

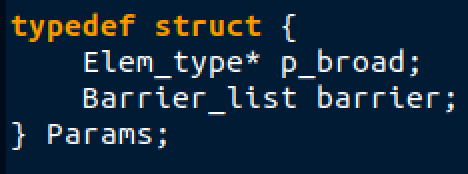


图2.2 子线程黑方块数据记录

对于白色方块的处理方式依旧保持不变，但是需要增设一个结构用以记录每一个子线程的白块状态。该结构存储当前的矩阵状态、以及下一个白块的位置。

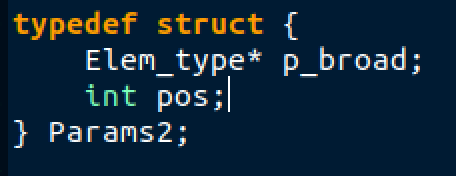


图2.3 子线程白方块数据记录

每一个线程组中所有的线程处理结束后，需要释放该线程组，所以在主递归函数中需要判断线程是否运行完成，也即需要等待线程的结束。并行算法中的其它部分与串行没有区别，这里不做过多解释。

## 2.4 实验方案

1. 在Linux中修改串行回溯法解决akari问题的代码。
2. 在平台进行测试。
3. 分析算法时间、空间复杂度。

## 2.5 实验结果与分析

图2.4、图2.5分别为串行算法以及并行算法的运行结果显示。很明显能够看到并行算法在10\*10矩阵运算的处理时间慢了将近15倍。一开始我觉得有点不解，按照算法设计的那样，我的平均并行粒度为4。所以在整体的时间上应该能够达到串行程序的4倍，可是现在却慢了15倍。

仔细分析后发现，造成这个结果的原因很有可能是数据规模不够大。少量的数据无法体现出并行计算的优势，反而在进程创建、销毁的过程中耗费了大量的时间。从而导致了整体的效果不如串行算法。

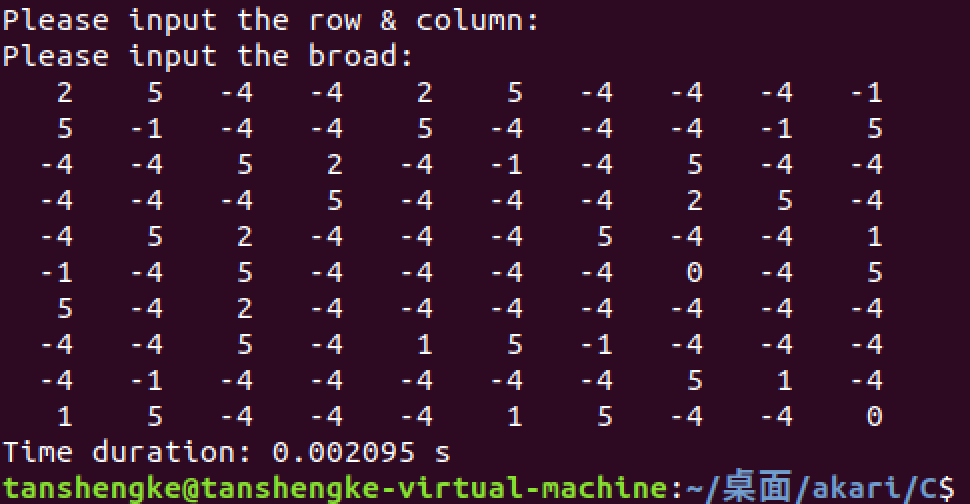


图2.4 串行回溯耗时

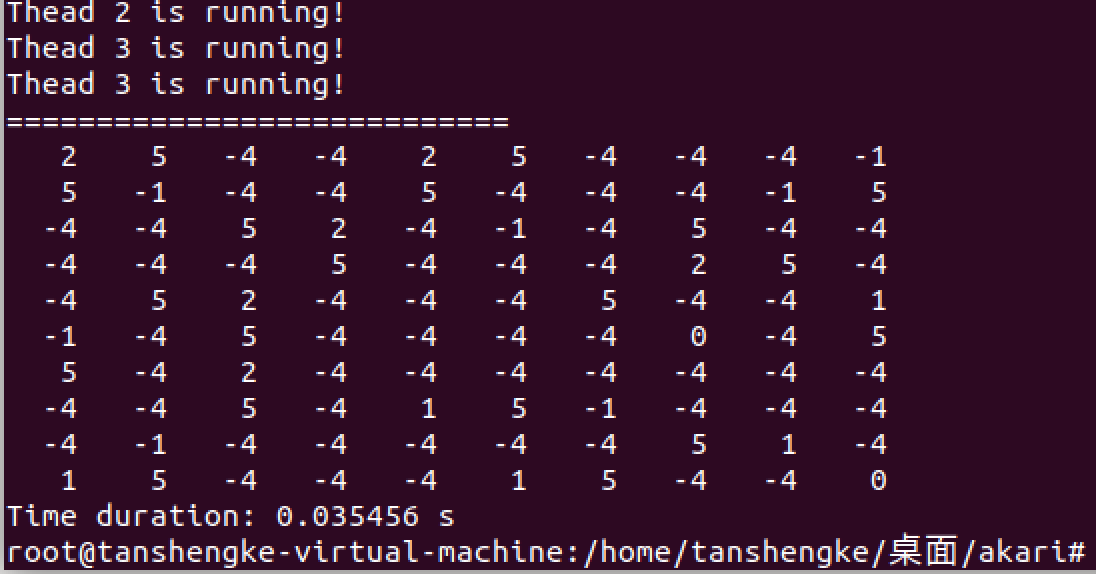


图2.5 并行回溯耗时

# 3 改进的并行回溯法求解Akari问题

## 3.1实验目的与要求

1. 掌握优化并行程序的方法；
2. 了解并行力度与性能之间的关系，掌握通过并行粒度分析程序性能的方法；
3. 了解并行程序设计与并行算法设计的基本分析方法与途径；

## 3.2问题描述

* 并行计算中的粒度与性能

多线程应用的并行任务工作量大小（粒度）会对其并行性能产生很大影响。在分解一项应用使之适用于多线程处理时，编程时通常采用的方法是从逻辑上将问题分割成尽量多的并行任务，或者在并行任务内根据共享数据与执行顺序决定进行哪些必要的通信。由于分割任务、将任务分配给线程以及在任务之间进行数据通信（共享）涉及到一定的成本，编程者通常需要聚合或整合分割的任务，用于避免随之产生的开销，尽量实现应用高效运行。通过聚合分割的任务可确定并行任务的最佳粒度。

粒度通常与工作负载在线程之间的均衡程度有关。尽管平衡大量小型任务的工作负载更容易，但这样做却可能导致通信和同步等方面的并行开销过高。此时，编程者可以通过将小型任务整合成一项任务，增加每项任务的粒度（工作量）来减少并行开销。

* akari问题并行粒度划分方法

并行粒度的划分实质上是一个将工作负载（任务）与工作单元（线程或进程）对应的过程，在本实验过程中，对于Akari问题的粒度划分方法有以下几种：

1. 每个节空间的节点分裂均产生新任务；
2. 当且仅当节空间树节点第一次分裂的时候产生新任务；
3. 当且仅当节空间树节点在数字“3”的时候分裂产生新任务；

* 编程要求

1. 根据上述分析，设计并行化的回溯算法，并用其解决Akari问题。在Linux环境下使用C/C++语言编程实现，同时测量算法执行时间，并计算加速比。
2. 使用并行程序设计方法实现算法。
3. 测量时间加速比。运行新的并行程序，测量程序运行时间，并与实验2中的并行运行时间相比较，计算出并行程序的加速比。
4. 根据数据和计算结果，思考分析不同的并行粒度对并行性能的影响。

* 格式说明

同第一节串行回溯法相同。

## 3.3算法描述

修改实验2中的并行算法，主要修改的位置是在递归函数中对于黑色方块处理的部分。之前采用的是对每个节点空间均分裂产生新任务。这种方式在问题规模变大时，线程数量将会极速增长，在线程的创建、销毁之上会消耗大量时间。总之对于每个任务新建线程是不太科学的。

考虑到这一点之后，采取相应的优化措施，就是将并行算法的并行粒度降低，以减少线程创建的开销。将程序运行耗时集中在求解问题之上。

## 3.4实验方案

1. 在Linux中修改并行回溯法解决akari问题的代码。
2. 在平台进行测试。
3. 计算加速比
4. 分析算法时间、空间复杂度。

## 3.5实验结果与分析

如图3.1、3.2所示，优化过后的并行程序总运行时间为0.15秒，比串行算法快10倍。



图3.1 串行耗时（14\*14）

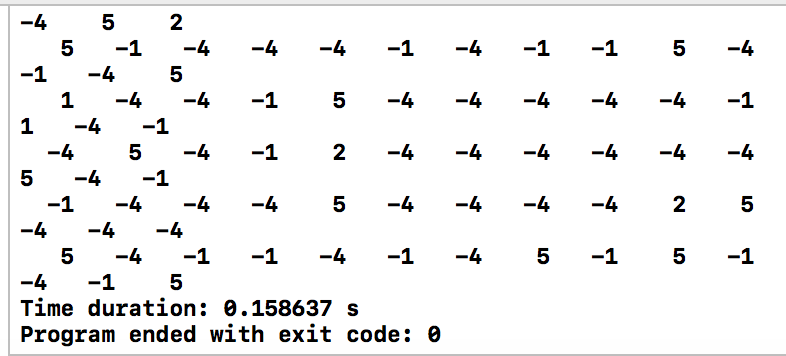


图3.2 优化并行回溯耗时（14\*14）

# 4 实验小结

通过akari问题的三个实验，让我对于串行算法并行化有了一定的了解。通过串行回溯法，到并行回溯法，再到不同并行粒度的并行回溯法。一步步深入了解并行程序设计、优化的过程。

在设计串行算法的时候所耗费的时间个人感觉是要比并行算法话费的时间多的，主要是因为串行方法是刚开始解决问题，是一个从零开始的阶段，在算法设计以及代码调试上话费了大量时间，而并行算法则是在串行算法的基础之上改进而来，免去了基础阶段的构思以及大量的代码编制过程，同时对于错误的定位也非常明确，所以相较而言总体上时间话费较少。

在对比串行算法设计的过程中，由于想减少黑色方块的处理时间，采用一次遍历构造链表的方式，让我感触颇深，虽然整体时间上确实有很大的进步，但在某些特殊矩阵的处理上却存在问题。这也让我意识到，在考虑问题时一定要全面考虑，不盲目的优化，否则会导致意想不到的后果。

总的来说通过本次试验，让我掌握了回溯法解决问题的基本思想，同时也掌握了并行回溯法的设计方案以及串行程序并行化的改造方案。总体上对pthread多线程编程的理解有了一定的提升。在今后的学习、生活中，也一定会继续努力，用并行的思想看待问题。

**附件：源代码**

1. 串行回溯法

# include <bits/stdc++.h>

# include "akari.h"

using namespace std;

namespace aka{

//请在命名空间内编写代码，否则后果自负

#include<stdio.h>

#include<stdlib.h>

#include<assert.h>

#include<time.h>

#define EMPTY -2 //空白位置 -2 0

#define DOT -3 //无法放置 -3 1

#define COVER -4 //被点亮 -4 2

#define BARRIER -1 //无数字黑块 -1 3

#define BARRIER\_0 0 //数字0黑块 0 4

#define BARRIER\_1 1 // 1 5

#define BARRIER\_2 2 // 2 6

#define BARRIER\_3 3 // 3 7

#define BARRIER\_4 4 // 4 8

#define BLUB 5 //电灯 5 9

typedef int Elem\_type;

/\*黑色方块结点 \*/

typedef struct Node{

int number; //黑色方块中的数字0～4

size\_t row;//size\_t是unsigned int

size\_t col;

struct Node\* next;

}Num\_barrier, \*Barrier\_list;

int m;//row num

int n;//column num

Elem\_type\* g\_broad;//the global broad

void read\_dimension(int\*, int\*);

void read\_broad(Elem\_type\*);

void print\_broad(Elem\_type\*);

Barrier\_list create\_barrier\_list(Elem\_type\*);

void copy\_broad(Elem\_type\*, const Elem\_type\*);

int check\_complete(Elem\_type\*);

int solve\_puzzle(Barrier\_list, Elem\_type\*);

int put\_blub(int, int, Elem\_type\*);

int put\_dot(int, int, Elem\_type\*);

int handle\_empty(int, Elem\_type\*);

int next\_empty(int, Elem\_type\*);

vector<vector<int> > solveAkari(vector<vector<int> > & g)

{

// 请在此函数内返回最后求得的结果

//read\_dimension(&m, &n);//读取行数和列数

m = g.size();

n = g[0].size();

g\_broad = (Elem\_type\*) malloc(sizeof(Elem\_type) \* m \* n);//动态分配内存给矩阵

//read\_broad(g\_broad);//读取文件中的数据存入变量g\_broad中

for(int i = 0; i < m; i++){

for(int j = 0; j < n; j++){

g\_broad[i\*m+j] = g[i][j];

}

}

Barrier\_list barriers = create\_barrier\_list(g\_broad);

int solved = solve\_puzzle(barriers, g\_broad);

if(!solved) printf("no solution!\n");

Barrier\_list p;

for(p = barriers; p;) {

barriers = barriers->next;

free(p);

p = barriers;

}

g.clear();

vector<int> temp;

for(int i = 0; i < m; i++){

temp.clear();

for(int j = 0; j < n; j++){

temp.push\_back(g\_broad[i\*m+j]);

}

g.push\_back(temp);

}

//free(g\_broad);

return g;

}

/\*从文件读取数据：行、列 \*/

void read\_dimension(int\* row, int\* col) {

//fscanf(fp, "%d %d", row, col);

scanf("%d %d", row, col);

}

/\*从文件读取初始数据 \*/

void read\_broad(Elem\_type\* broad) {

int i;

for(i = 0; i < m\*n; i++)

//fscanf(fp, "%d", &broad[i]);

scanf("%d", &broad[i]);

}

/\*输出矩阵 \*/

void print\_broad(Elem\_type\* broad) {

int r, c;

for(r = 0; r < m; r++) {

for(c = 0; c < n; c++)

printf("%d ", broad[r\*n + c]);

printf("\n");

}

}

Barrier\_list create\_barrier\_list(Elem\_type\* broad) {//创建黑色方块链表

int i;

Barrier\_list header, p;

header = p = NULL;

for(i = 0; i < m\*n; i++){

if(broad[i] == BARRIER\_0){

Barrier\_list q = (Barrier\_list)malloc(sizeof(Num\_barrier));

q->row = i / n;

q->col = i % n;

q->number = 0;

(header == NULL) ? (header = p = q) : (p->next = q, p=p->next);

}

}

for(i = 0; i < m\*n; i++){

if(broad[i] == BARRIER\_4){

Barrier\_list q = (Barrier\_list)malloc(sizeof(Num\_barrier));

q->row = i / n;

q->col = i % n;

q->number = 4;

(header == NULL) ? (header = p = q) : (p->next = q, p=p->next);

}

}

for(i = 0; i < m\*n; i++){

if(broad[i] == BARRIER\_3){

Barrier\_list q = (Barrier\_list)malloc(sizeof(Num\_barrier));

q->row = i / n;

q->col = i % n;

q->number = 3;

(header == NULL) ? (header = p = q) : (p->next = q, p=p->next);

}

}

for(i = 0; i < m\*n; i++){

if(broad[i] == BARRIER\_1){

Barrier\_list q = (Barrier\_list)malloc(sizeof(Num\_barrier));

q->row = i / n;

q->col = i % n;

q->number = 1;

(header == NULL) ? (header = p = q) : (p->next = q, p=p->next);

}

}

for(i = 0; i < m\*n; i++){

if(broad[i] == BARRIER\_2){

Barrier\_list q = (Barrier\_list)malloc(sizeof(Num\_barrier));

q->row = i / n;

q->col = i % n;

q->number = 2;

(header == NULL) ? (header = p = q) : (p->next = q, p=p->next);

}

}

if(p != NULL) p->next = NULL;

return header;

}

int solve\_puzzle(Num\_barrier\* barrier, Elem\_type\* broad) {

if(NULL != barrier) {//first phase：处理黑块

Num\_barrier\* p;

Elem\_type\* cp\_broad;

int r, c;

int handle = 0;

cp\_broad = (Elem\_type\*)malloc(sizeof(Elem\_type) \* m \* n);

copy\_broad(cp\_broad, broad);

p = barrier;

r = p->row;

c = p->col;

switch(p->number){

case 4: //一种可能

if(put\_blub(r-1, c, cp\_broad)

&&put\_blub(r, c-1, cp\_broad)

&&put\_blub(r+1, c, cp\_broad)

&&put\_blub(r, c+1, cp\_broad))

handle = solve\_puzzle(p->next, cp\_broad);

break;

case 3: //四种可能

if(put\_blub(r-1, c, cp\_broad)

&&put\_blub(r, c-1,cp\_broad)

&&put\_blub(r+1, c, cp\_broad)

&&put\_dot(r, c+1,cp\_broad)) {

handle = solve\_puzzle(p->next, cp\_broad);

}

//restore the state of broad

copy\_broad(cp\_broad, broad);

if(!handle && put\_dot(r-1, c, cp\_broad)

&&put\_blub(r, c-1,cp\_broad)

&&put\_blub(r+1, c, cp\_broad)

&&put\_blub(r, c+1,cp\_broad))

handle = solve\_puzzle(p->next, cp\_broad);

//

copy\_broad(cp\_broad, broad);

if(!handle && put\_blub(r-1, c, cp\_broad)

&&put\_dot(r, c-1,cp\_broad)

&&put\_blub(r+1, c, cp\_broad)

&&put\_blub(r, c+1,cp\_broad))

handle = solve\_puzzle(p->next, cp\_broad);

//

copy\_broad(cp\_broad, broad);

if(!handle && put\_blub(r-1, c, cp\_broad)

&&put\_blub(r, c-1,cp\_broad)

&&put\_dot(r+1, c, cp\_broad)

&&put\_blub(r, c+1,cp\_broad))

handle = solve\_puzzle(p->next,cp\_broad);

break;

case 2: //6种可能

if(put\_blub(r-1, c, cp\_broad)

&&put\_blub(r, c-1,cp\_broad)

&&put\_dot(r+1, c, cp\_broad)

&&put\_dot(r, c+1,cp\_broad))

handle = solve\_puzzle(p->next, cp\_broad);

copy\_broad(cp\_broad, broad);

if(!handle && put\_dot(r-1, c, cp\_broad)

&&put\_blub(r, c-1,cp\_broad)

&&put\_dot(r+1, c, cp\_broad)

&&put\_blub(r, c+1,cp\_broad))

handle = solve\_puzzle(p->next, cp\_broad);

copy\_broad(cp\_broad, broad);

if(!handle && put\_blub(r-1, c, cp\_broad)

&&put\_dot(r, c-1,cp\_broad)

&&put\_dot(r+1, c, cp\_broad)

&&put\_blub(r, c+1,cp\_broad))

handle = solve\_puzzle(p->next, cp\_broad);

copy\_broad(cp\_broad,broad);

if(!handle && put\_dot(r-1, c, cp\_broad)

&&put\_dot(r, c-1,cp\_broad)

&&put\_blub(r+1, c, cp\_broad)

&&put\_blub(r, c+1,cp\_broad))

handle = solve\_puzzle(p->next, cp\_broad);

copy\_broad(cp\_broad, broad);

if(!handle && put\_dot(r-1, c, cp\_broad)

&&put\_blub(r, c-1,cp\_broad)

&&put\_blub(r+1, c, cp\_broad)

&&put\_dot(r, c+1,cp\_broad))

handle = solve\_puzzle(p->next, cp\_broad);

copy\_broad(cp\_broad, broad);

if(!handle && put\_blub(r-1, c, cp\_broad)

&&put\_dot(r, c-1,cp\_broad)

&&put\_blub(r+1, c, cp\_broad)

&&put\_dot(r, c+1,cp\_broad))

handle = solve\_puzzle(p->next, cp\_broad);

break;

case 1 : //四种可能

if(put\_blub(r-1, c, cp\_broad)

&&put\_dot(r, c-1,cp\_broad)

&&put\_dot(r+1, c, cp\_broad)

&&put\_dot(r, c+1,cp\_broad))

handle = solve\_puzzle(p->next, cp\_broad);

copy\_broad(cp\_broad, broad);

if(!handle && put\_dot(r-1, c, cp\_broad)

&&put\_blub(r, c-1,cp\_broad)

&&put\_dot(r+1, c, cp\_broad)

&&put\_dot(r, c+1,cp\_broad))

handle = solve\_puzzle(p->next, cp\_broad);

copy\_broad(cp\_broad, broad);

if(!handle && put\_dot(r-1, c, cp\_broad)

&&put\_dot(r, c-1,cp\_broad)

&&put\_blub(r+1, c, cp\_broad)

&&put\_dot(r, c+1,cp\_broad))

handle = solve\_puzzle(p->next, cp\_broad);

copy\_broad(cp\_broad, broad);

if(!handle && put\_dot(r-1, c, cp\_broad)

&&put\_dot(r, c-1,cp\_broad)

&&put\_dot(r+1, c, cp\_broad)

&&put\_blub(r, c+1,cp\_broad))

handle = solve\_puzzle(p->next, cp\_broad);

break;

case 0:

if(put\_dot(r-1, c,cp\_broad) //上方

&& put\_dot(r, c-1, cp\_broad) //左边

&& put\_dot(r+1, c, cp\_broad) //下边

&& put\_dot(r, c+1, cp\_broad)) //右边

handle = solve\_puzzle(p->next, cp\_broad);

break;

default:

break;

}

//free(cp\_broad);

cp\_broad = NULL;

return handle;

} else {//second phase：处理白块

return handle\_empty(next\_empty(-1, broad), broad);

}

}

/\*\* name:next\_empty

\* function:获取下一个未被灯光覆盖的方块

\* input：int cur：当前坐标， int\* broad：处理过的数据数组

\* return：i：下一个方块坐标； -1:不存在未被覆盖的方块

\*/

int next\_empty(int cur, Elem\_type\* broad){

int i;

for(i = cur+1; i < m \*n; i++)

if(broad[i] == EMPTY)

return i;

return -1;

}

/\*\*name：handle\_empty

\* function：处理未被点亮的空白方块

\* input：int cur：空白方块坐标

\* int\* broad：处理完成黑色方块的数据数组

\* output：处理结果 1:处理完成，得到正确结果

\* 0:处理失败，不存在解

\*/

int handle\_empty(int cur, Elem\_type\* broad) {

Elem\_type\* cp\_broad;

int handle = 0;

if(check\_complete(broad)){ //完全覆盖，得出正确结果

copy\_broad(g\_broad, broad);

return 1;

}

if(cur == -1) //没有空白位置

return 0;

cp\_broad = (Elem\_type\*)malloc(sizeof(Elem\_type) \* m \* n);

copy\_broad(cp\_broad, broad);

put\_blub(cur / n, cur % n, cp\_broad);//在cur所在的位置放一个灯，cur/n和cur%n是为了得到坐标

handle = handle\_empty(next\_empty(cur, cp\_broad), cp\_broad);

if(!handle){

//retore the previous state of the broad

copy\_broad(cp\_broad, broad);

handle = handle\_empty(next\_empty(cur, cp\_broad), cp\_broad);

}

free(cp\_broad);

return handle;

}

/\*

\*Copy the src broad to des

\*/

void copy\_broad(Elem\_type\* des, const Elem\_type\* src) {

int i;

for(i = 0; i < m\*n; i++)

des[i] = src[i];

}

/\*\* name：check\_complete

\* function：最终效果检查，是否所有的空白位置都被灯光覆盖

\* input：int\* broad：处理过后的数据数组

\* return：1:不存在未被照亮的方块 0:存在未被照亮的方块

\*/

int check\_complete(Elem\_type\* broad){

int i;

for(i = 0; i < m\*n; i++)

if(broad[i] == EMPTY || broad[i] == DOT) //存在empty or dot broad[i] < COVER

return 0;

int ps[4][2] = {-1, 0, 1, 0, 0, -1, 0, 1};

for (int i = 0; i < m\*n; i++) {

if(broad[i] >= 0 && broad[i]<=4){

int cnt = 0;

int x = i/n;

int y = i%n;

for (int k = 0; k < 4; ++k) {

int dx = x + ps[k][0], dy = y + ps[k][1];

int tmp = dx\*m + dy;

if (dx >= 0 && dx < n && dy >= 0 && dy < m && broad[tmp] == 5) ++cnt;

}

if (cnt != broad[i]) {

return 0;

}

}

}

return 1;

}

/\*\*name：put\_blub

\* function：在坐标（row，col）处放置灯泡，并且点亮相应方块

\* input：int row：x坐标

\* int col：y坐标

\* int\* broad：处理过的数据数组

\* return：放置结果 1:成功 0:失败

\*/

int put\_blub(int row, int col, Elem\_type\* broad) {

int r, c;

//坐标是否合法，非法无需处理

if((row < 0 && row >= m)

|| (col < 0 && col >= n))

return 0;

//judge can put? ：为cover也不能放置，因为该坐标的行或列上一定存在电灯

if(broad[row \* n + col] != EMPTY && broad[row\*n+col] != BLUB)

return 0;

else if(broad[row\*n+col] == BLUB)

return 1;

broad[row \* n + col] = BLUB;

// illuminate four directions

for(r = row -1 , c = col; r >= 0 && broad[r\*n+c] < BARRIER; r--)

broad[r\*n + c] = COVER;

for(r = row, c = col - 1; c >= 0 && broad[r\* n+c] < BARRIER; c--)

broad[r\*n + c] = COVER;

for(r = row + 1, c = col; r < m && broad[r\*n + c] < BARRIER; r++)

broad[r\*n + c] = COVER;

for(r = row, c = col +1; c < n && broad[r\*n + c] < BARRIER; c++)

broad[r\*n + c] = COVER;

return 1;

}

/\*\*name：put\_dot

\* function：设置标记点，标记该位置不能放置灯泡，优化后续的白块处理效率

\* input：int row：x坐标

\* int col：y坐标

\* int\* broad：处理过后的数据数组

\* return：能否放置标记 1:放置成功 0:放置失败

\*/

int put\_dot(int row, int col, Elem\_type\* broad) {

if((row >= 0 && row < m)

&& (col >= 0 && col < n)){ //判断位置是否在矩阵内

if(broad[row\*n + col] == BLUB)

return 0;

if(broad[row\*n + col] == EMPTY) {

broad[row\*n + col] = DOT;

}

}

return 1;

}

}

1. 并行回溯法

//

// akari.h

// ab

//

// Created by Doris on 2019/7/19.

// Copyright © 2019年 tsk. All rights reserved.

//

#ifndef akari\_h

#define akari\_h

#include<stdio.h>

#include<stdlib.h>

#include<assert.h>

#include<pthread.h>

#include<time.h>

#pragma comment(lib, "pthreadVC2.lib")

#define EMPTY 0

#define DOT 1

#define COVER 2

#define BARRIER 3

#define BARRIER\_0 4

#define BARRIER\_1 5

#define BARRIER\_2 6

#define BARRIER\_3 7

#define BARRIER\_4 8

#define BLUB 9

#define NORTH (1<<0)

#define EAST (1<<1)

#define SOUTH (1<<2)

#define WEST (1<<3)

typedef int Elem\_type;

typedef struct Node{

int number;

size\_t row;

size\_t col;

struct Node\* next;

}Num\_barrier, \*Barrier\_list;

int m;//row num

int n;//column num

Elem\_type\* g\_broad;//the global broad

void read\_dimension(FILE\*, int\*, int\*);

void read\_broad(FILE\*,Elem\_type\*);

void print\_broad(Elem\_type\*);

Barrier\_list create\_barrier\_list(Elem\_type\*);

void copy\_broad(Elem\_type\*, const Elem\_type\*);

int check\_complete(Elem\_type\*);

void\* solve\_puzzle(void\*);

//int solve\_puzzle(Barrier\_list, Elem\_type\*);

int put\_blub(int, int, Elem\_type\*);

int put\_dot(int, int, Elem\_type\*);

void\* handle\_empty(void\*);

int next\_empty(int, Elem\_type\*);

#endif /\* akari\_h \*/

//

// main.cpp

// ab

//

// Created by Doris on 2019/7/19.

// Copyright © 2019年 tsk. All rights reserved.

//

#include"akari.h"

pthread\_mutex\_t lock = PTHREAD\_MUTEX\_INITIALIZER;

int handle = 0;

typedef struct {

Elem\_type\* p\_broad;

Barrier\_list barrier;

} Params;

typedef struct {

Elem\_type\* p\_broad;

int pos;

} Params2;

int main() {

FILE \*fp;

double start, finish, diff;

if((fp = fopen("/Users/Doris/Desktop/lab/ab/ab/akari.txt", "r")) == NULL)

return 1;

read\_dimension(fp, &m, &n);

g\_broad = (Elem\_type\*) malloc(sizeof(Elem\_type) \* m \* n);

read\_broad(fp, g\_broad);

print\_broad(g\_broad);

Barrier\_list barriers = create\_barrier\_list(g\_broad);

Params\* params = (Params\*) malloc(sizeof(Params));

Elem\_type\* cp\_broad =(Elem\_type\*) malloc(sizeof(Elem\_type)\*m\*n);

copy\_broad(cp\_broad, g\_broad);

params->p\_broad = cp\_broad;

params->barrier = barriers;

start = clock();

solve\_puzzle((void\*) params);

finish = clock();

diff = (finish - start) / CLOCKS\_PER\_SEC;

if(handle) {

print\_broad(g\_broad);

printf("Time duration: %lf s\n", diff);

}else

printf("No solution!\n");

free(g\_broad);

fclose(fp);

Barrier\_list p;

for(p = barriers; p;) {

barriers = barriers->next;

free(p);

p = barriers;

}

return 0;

}

void read\_dimension(FILE\* fp, int\* row, int\* col) {

printf("Please input the row & column:\n");

fscanf(fp, "%d %d", row, col);

}

void read\_broad(FILE\* fp, Elem\_type\* broad) {

printf("Please input the broad:\n");

int i;

for(i = 0; i < m\*n; i++)

fscanf(fp, "%d", &broad[i]);

}

void print\_broad(Elem\_type\* broad) {

int r, c;

printf("============================\n");

for(r = 0; r < m; r++) {

for(c = 0; c < n; c++)

printf("%d ", broad[r\*n + c]);

printf("\n");

}

}

Barrier\_list create\_barrier\_list(Elem\_type\* broad) {

int i;

Barrier\_list header, p;

header = p = NULL;

for(i = 0; i < m\*n; i++){

if(broad[i] == BARRIER\_0){

Barrier\_list q = (Barrier\_list)malloc(sizeof(Num\_barrier));

q->row = i / n;

q->col = i % n;

q->number = 0;

(header == NULL) ? (header = p = q) : (p->next = q, p=p->next);

}

}

for(i = 0; i < m\*n; i++){

if(broad[i] == BARRIER\_4){

Barrier\_list q = (Barrier\_list)malloc(sizeof(Num\_barrier));

q->row = i / n;

q->col = i % n;

q->number = 4;

(header == NULL) ? (header = p = q) : (p->next = q, p=p->next);

}

}

for(i = 0; i < m\*n; i++){

if(broad[i] == BARRIER\_3){

Barrier\_list q = (Barrier\_list)malloc(sizeof(Num\_barrier));

q->row = i / n;

q->col = i % n;

q->number = 3;

(header == NULL) ? (header = p = q) : (p->next = q, p=p->next);

}

}

for(i = 0; i < m\*n; i++){

if(broad[i] == BARRIER\_1){

Barrier\_list q = (Barrier\_list)malloc(sizeof(Num\_barrier));

q->row = i / n;

q->col = i % n;

q->number = 1;

(header == NULL) ? (header = p = q) : (p->next = q, p=p->next);

}

}

for(i = 0; i < m\*n; i++){

if(broad[i] == BARRIER\_2){

Barrier\_list q = (Barrier\_list)malloc(sizeof(Num\_barrier));

q->row = i / n;

q->col = i % n;

q->number = 2;

(header == NULL) ? (header = p = q) : (p->next = q, p=p->next);

}

}

if(p != NULL) p->next = NULL;

return header;

}

pthread\_t\* split(int flag, Elem\_type\* broad, Barrier\_list barrier) {

Elem\_type\* cp\_broad = (Elem\_type\*) malloc(sizeof(Elem\_type) \* m\*n);

copy\_broad(cp\_broad,broad);

int i, success = 0;

int r = barrier->row, c = barrier->col;

for(i = 0; i < 4; i++) {

int direction = (1<<i);

int type = (flag & direction);

switch(direction) {

case NORTH:

success = (type != 0) ? put\_blub(r-1, c, cp\_broad):put\_dot(r-1, c, cp\_broad);

break;

case EAST:

success = (type != 0) ? put\_blub(r, c-1, cp\_broad):put\_dot(r, c-1, cp\_broad);

break;

case SOUTH:

success = (type != 0) ? put\_blub(r+1, c, cp\_broad):put\_dot(r+1, c, cp\_broad);

break;

case WEST:

success = (type != 0) ? put\_blub(r, c+1, cp\_broad):put\_dot(r, c+1, cp\_broad);

break;

}

if(!success)

break;

}

if(!success) {

free(cp\_broad);

return NULL;

}

else {

//alway check the result before we split another thread in case we do the unuseful work

if(handle) {

free(cp\_broad);

return NULL;

}

Params\* p\_params = (Params\*) malloc(sizeof(Params));

pthread\_t\* p\_handle = (pthread\_t\*) malloc(sizeof(pthread\_t));

p\_params->barrier = barrier->next;

p\_params->p\_broad = cp\_broad;

pthread\_create(p\_handle, NULL, solve\_puzzle, (void\*)p\_params);//split

return p\_handle;

}

}

void\* solve\_puzzle(void\* p\_params) {

//printf("function called \n");

Params\* params = (Params\*)p\_params;

Barrier\_list barrier = params->barrier;

Elem\_type\* broad = params->p\_broad;

if(handle) {

free(broad);

free(params);

return NULL;

}

if(NULL != barrier) {//first phase

int i;

pthread\_t\*\* thread\_handles;

thread\_handles = (pthread\_t\*\*) calloc(6, sizeof(pthread\_t\*));//max thread num is 6

switch(barrier->number){

case 4:

thread\_handles[0] = split(NORTH|EAST|SOUTH|WEST, broad, barrier);

break;

case 3:

thread\_handles[0] = split(NORTH|EAST|SOUTH, broad, barrier);

thread\_handles[1] = split(NORTH|EAST|WEST, broad, barrier);

thread\_handles[2] = split(NORTH|SOUTH|WEST, broad, barrier);

thread\_handles[3] = split(EAST|SOUTH|WEST, broad, barrier);

break;

case 2:

thread\_handles[0] = split(NORTH|EAST, broad, barrier);

thread\_handles[1] = split(NORTH|SOUTH, broad, barrier);

thread\_handles[2] = split(NORTH|WEST, broad, barrier);

thread\_handles[3] = split(EAST|SOUTH, broad, barrier);

thread\_handles[4] = split(EAST|WEST, broad, barrier);

thread\_handles[5] = split(SOUTH|WEST, broad, barrier);

break;

case 1:

thread\_handles[0] = split(NORTH, broad, barrier);

thread\_handles[1] = split(EAST, broad, barrier);

thread\_handles[2] = split(SOUTH, broad, barrier);

thread\_handles[3] = split(WEST, broad, barrier);

break;

case 0:

thread\_handles[0] = split(0, broad, barrier);

break;

default:

break;

}

for(i = 0; i < 6;i++)

if(thread\_handles[i] != NULL) {

printf("Thead %d is running!\n", i);

pthread\_join(\*thread\_handles[i], NULL);

}

for(i = 0; i < 6;i++)

if(thread\_handles[i] != NULL)

free(thread\_handles[i]);

free(thread\_handles);

free(broad);

free(params);

return NULL;

} else {//second phase

Params2\* pp = (Params2\*) malloc(sizeof(Params2));

pp->pos = next\_empty(-1, broad);

pp->p\_broad = broad;

handle\_empty((void\*) pp);

return NULL;

}

}

void \*handle\_empty(void\* params) {

Params2\* pp = (Params2\*) params , \*pp2;//pp2 as the params of the splited thread

Elem\_type\* broad, \*cp\_broad;

pthread\_t\* thread\_handle;

int cur;

broad = (Elem\_type\*)pp->p\_broad;

cur = pp->pos;

if(check\_complete(broad)){

pthread\_mutex\_lock(&lock);

if(!handle) {

copy\_broad(g\_broad, broad);

handle= 1;

}

pthread\_mutex\_unlock(&lock);

free(pp->p\_broad);

free(pp);

return NULL;

}

if(cur == -1) {

free(pp->p\_broad);

free(pp);

return NULL;

}

//alway check the result before we split another thread in case we do the unuseful work

if(handle) {

free(pp->p\_broad);

free(pp);

return NULL;

}

thread\_handle = (pthread\_t\*) malloc(sizeof(pthread\_t));

cp\_broad = (Elem\_type\*)malloc(sizeof(Elem\_type) \* m \* n);

copy\_broad(cp\_broad, broad);

put\_blub(cur / n, cur % n, cp\_broad);

pp2 = (Params2\*) malloc(sizeof(Params2));

pp2->pos = next\_empty(cur, cp\_broad);

pp2->p\_broad = cp\_broad;

//split the thread when we put the blub on it

pthread\_create(thread\_handle, NULL, handle\_empty, (void \*) pp2);

//keep the initial thread for the condition that we do not put the blub

pp->pos = next\_empty(cur, broad);

handle\_empty((void\*) pp);

pthread\_join(\*thread\_handle, NULL);

free(thread\_handle);

return NULL;

}

/\*

\*Copy the src broad to des

\*/

void copy\_broad(Elem\_type\* des, const Elem\_type\* src) {

int i;

for(i = 0; i < m\*n; i++)

des[i] = src[i];

}

int check\_complete(Elem\_type\* broad){

int i;

for(i = 0; i < m\*n; i++)

if(broad[i] < COVER)

return 0;

return 1;

}

int put\_blub(int row, int col, Elem\_type\* broad) {

int r, c;

if((row < 0 && row >= m) || (col < 0 && col >= n))

return 0;

//judge can put?

if(broad[row \* n + col] != EMPTY && broad[row\*n+col] != BLUB)

return 0;

else if(broad[row\*n+col] == BLUB)

return 1;

broad[row \* n + col] = BLUB;

// illuminate four directions

for(r = row -1 , c = col; r >= 0 && broad[r\*n+c] < BARRIER; r--)

broad[r\*n + c] = COVER;

for(r = row, c = col - 1; c >= 0 && broad[r\* n+c] < BARRIER; c--)

broad[r\*n + c] = COVER;

for(r = row + 1, c = col; r < m && broad[r\*n + c] < BARRIER; r++)

broad[r\*n + c] = COVER;

for(r = row, c = col +1; c < n && broad[r\*n + c] < BARRIER; c++)

broad[r\*n + c] = COVER;

return 1;

}

int put\_dot(int row, int col, Elem\_type\* broad) {

if((row >= 0 && row < m)

&& (col >= 0 && col < n)){

if(broad[row\*n + col] == BLUB)

return 0;

if(broad[row\*n + col] == EMPTY) {

broad[row\*n + col] = DOT;

}

}

return 1;

}

int next\_empty(int cur, Elem\_type\* broad){

int i;

for(i = cur+1; i < m \*n; i++)

if(broad[i] == EMPTY)

return i;

return -1;

}