華中科技大學

课 程 实 验 报 告

课程名称： 并行编程原理与实践

专业班级： CS1804

学　　号： U201814604

姓　　名： 黄俊淇

指导教师： 金海

报告日期： 2021.06.30

计算机科学与技术学院

目录

1. 汉诺塔计算

1.1 实验内容

1.1.1 串行环境下的汉诺塔问题求解

在串行环境下编写解决汉诺塔问题的C语言小程序，并按要求输出对应的每一层的移动次数。

1.1.2 pthread 环境下的汉诺塔问题求解

编写使用多线程解决汉诺塔问题的C语言小程序，并按要求输出对应的每一层的移动次数。

1.1.3 OpenMP 环境下的汉诺塔问题求解

编写使用 OpenMP 解决汉诺塔问题的C语言小程序，并按要求输出对应的每一层的移动次数。

1.1.4 MPI 环境下的汉诺塔问题求解

编写使用 MPI 解决汉诺塔问题的C语言小程序，并按要求输出对应的每一层的移动次数。

1.2 算法描述

1.2.1 串行环境下的汉诺塔问题求解

当 n=1，则最底下一层移动次数为 1；当 n=2，则最底下一层移动次数为 1，上层移动次数为 2……因此找到规律，从下到上第 n 层移动次数为 2 的 n-1 次幂。因此串行算法就是遍历 n 次，每次打印 2 的 n-1 次幂。

显然，串行算法时间复杂度即为 0（n），而没有开辟任何空间，因此空间复杂度为 0（1）。

1.2.2 pthread 环境下的汉诺塔问题求解

通过 n 个线程计算每一层输出的次数，并保存在一个数组里，最后输出数组下标 1～n 的元素。

时间复杂度为 0（1），空间复杂度为 0（n）。

1.2.3 OpenMP 环境下的汉诺塔问题求解

将 1.2.2 的算法改为使用 open MP 的算法。

时间复杂度为 0（1），空间复杂度为 n 个线程的开销。

1.2.4 MPI 环境下的汉诺塔问题求解

与前三关相比，MPI 环境下得到的并行进程数是由参数 MPI_COMM_WORLD 得到，而不是程序员控制，因此该环境下需要根据进程数划分任务，每个任务打印对应层数的移动次数。因为测试集最大也只输入 45，因此创建全局变量 data[50]保存每层的移动次数，recv[50]用来接收每层的移动次数。

每个任务需要计算的层数由 50/（进程数-1）得到，如果不能整除，则每层需要计算的层数+1。假设进程数为 3，0 号进程接收消息并不参与计算，因此参与计算只有 1，2 号两个进程，那么 1 号进程打印 0-24 层，2 号进程打印 25-49 层，最后发给 0 号进程，0 号进程每次接受完就将 recv 数组保存的移动次数移动到 data，最后 data 数组里保留的就是 1-50 各层的移动次数。再根据输入 n，决定输出 1-n 层的移动次数。

每个进程的时间复杂度为对应需要计算的层数，因此时间复杂度为 O(n)，虽然开辟了两个大小为 50 的数组，但是大小不变，因此空间复杂度为 O(1)。

1.3 实验方案

开发环境：如下图所示

测试环境：educoder 测试平台

编程语言：C++

1.3.1 串行环境下的汉诺塔问题求解

输入 n=4，因此由算法应该依次打印 2 的 0 次幂，1 次幂，2 次幂，3 次幂，即 1，2，4，8。

串行代码如下：

```
void fb(int n){
    int i = 1;
    long long num = 1;
    while (i < n){
        printf("%lld ",num);
        num = num << 1;
        i++;
    }
    printf("%lld",num);
}
```

1.3.2 pthread 环境下的汉诺塔问题求解

输入 n=10，因此由算法应该依次打印 2 的 0 次幂，1 次幂，2 次幂……9 次幂即 1，2，4，8，16，32，64，128，256，512。

关键代码如下：

```
long long data[50];


void* fb2(int *n)
{
    data[*n] = pow(2,*(int *)n-1);
    return ((void *)0);
}
/***** Begin *****/
int main()
{
    int n;
```

```
    scanf("%d",&n);
    pthread_t tid[n];
    for (int i = 0; i < n; ++i) {
        pthread_create(&tid[i],NULL,fb2,&i);
    }

    for (int i = 0; i < n; ++i) {
        pthread_join(tid[i],NULL);
    }

    for (int i = 1; i <= n; ++i) {
        printf("%lld ",data[i]);
    }
    return 0;
}
```

1.3.3 OpenMP 环境下的汉诺塔问题求解

输入 n=25，因此由算法应该依次打印 2 的 0 次幂，1 次幂，2 次幂……24 次幂，即 1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 32768 65536 131072 262144 524288 1048576 2097152 4194304 8388608 16777216。

关键代码如下：

```
int main()
{
    int n;
    scanf("%d",&n);
    long long nums[50];
#pragma omp parallel for
    for(int i = 0; i <= n; i++){
        nums[i] = pow(2,i-1);
    }
```

```
        for(int i = 1; i <= n; i++){
            printf("%lld ", nums[i]);
        }
        return 0;
    }
```

1.3.4 MPI 环境下的汉诺塔问题求解

输入 n=45，因此由算法应该依次打印 2 的 0 次幂，1 次幂，2 次幂……44 次幂，即 1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 32768 65536 131072 262144 524288 1048576 2097152 4194304 8388608 16777216 33554432 67108864 134217728 268435456 536870912 1073741824 2147483648 4294967296 8589934592 17179869184 34359738368 68719476736 137438953472 274877906944 549755813888 1099511627776 2199023255552 4398046511104 8796093022208 17592186044416。

关键代码如下：int count = 50/(numprocs-1);

```
    if(50%(numprocs-1) != 0){
        count++;
    }//以进程数划分任务
     if (myid != 0) {   //非 0 号进程发送消息
          for(int i = (myid-1)*count; i < myid*count; i++){
             data[i] = fb(i);
        }
          MPI_Send(data, 50, MPI_LONG_LONG_INT, 0, 99,
               MPI_COMM_WORLD);
    }
    else {     // myid == 0，即 0 号进程接收消息
        int n;
        scanf("%d",&n);
          for (source = 1; source < numprocs; source++) {
```

```
                MPI_Recv(recv, 50, MPI_LONG_LONG_INT, source, 99,
                    MPI_COMM_WORLD, &status);
                for(int j = (source-1)*count; j < source*count; ++j){
                    data[j] = recv[j];
                }
            }
        for(int i = 1; i <= n; i++){
            printf("%lld ", data[i]);
        }
    }
```

1.3.5 串行、Open MP 环境比较

统一输入 n=10 进行测量。

1.4 实验结果与分析

1.4.1 串行环境下的汉诺塔问题求解

实验结果如图 1.1 所示

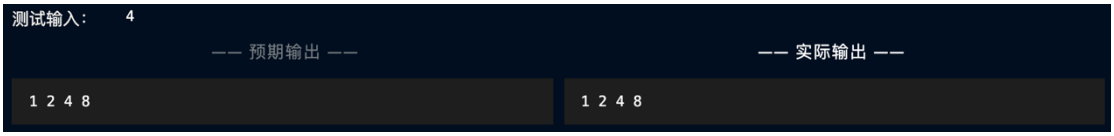

图 1.1 串行测试结果

实验结果如预期相同，因此该算法是正确的。

1.4.2 pthread 环境下的汉诺塔问题求解

实验结果如图 1.2 所示

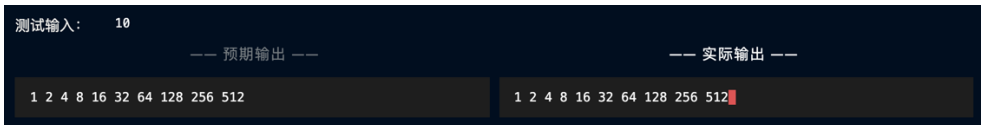

图 1.2 pthread 测试结果

实验结果如预期相同，因此该算法是正确的。

1.4.3 OpenMP 环境下的汉诺塔问题求解

实验结果如图 1.3 所示

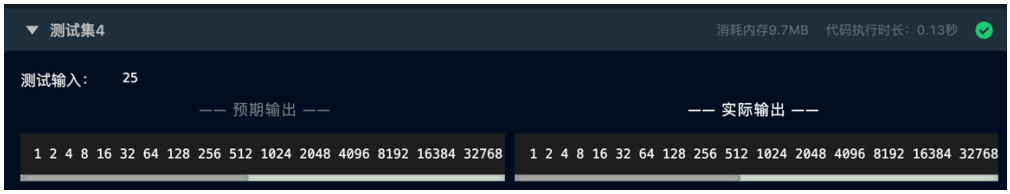

图 1.3 OpenMP 测试结果

实验结果如预期相同，因此该算法是正确的。

1.4.4 MPI 环境下的汉诺塔问题求解

实验结果如图 1.4 所示

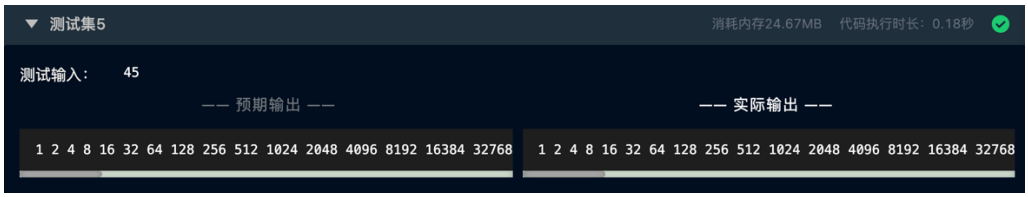

图 1.4 MPI 测试结果

实验结果如预期相同，因此该算法是正确的。

1.4.5 串行、Open MP 环境比较

n=10，串行如图 1.5 所示， Open MP 如图 1.6 所示

```
10
1 2 4 8 16 32 64 128 256 512
time=1625659949
```

图 1.5 串行

```
10
1 2 4 8 16 32 64 128 256 512
time=1625659839
```

图 1.6 Open MP

可见，当 n=10 的时候，Open MP 效率比串行要高，因此并行化成功。

2. 数独问题求解

2.1 实验内容

2.1.1 使用回溯法解决数独问题

编程实现使用回溯法解决 Sudoku 问题

2.1.2 使用并行回溯法求解数独问题

在 Sudoku 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。因此可以并行化回溯法解决数独。

2.1.3 使用改进的并行回溯法求解数独问题

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。

2.2 算法描述

2.2.1 使用回溯法解决数独问题

1. 对问题进行划分 寻找九宫格中的空白格子
2. 选择枚举 枚举每一步的选择。 在空白格子中填入 1-9 中的一个数字，利用数独规范，对比同一列、同一行，以及同一个九宫格数字，找出其所有可行解。
3. 构造解空间 根据上述讨论构造解空间，从第一个空白格子开始尝试填入 1-9 中的一个数字，填入后判断是否为一个可行解，若为可行解，则解空间向下进行分枝，否则向上进行回溯

若给定数独有 n 个空白格子，那么时间复杂度为 O(n!)，空间复杂度为 O（1）

2.2.2 使用并行回溯法求解数独问题

对于每个空白格子创建一个新的线程用来回溯，即每个解空间的节点分裂均产生新任务。

时间复杂度仍为 O(n!)，空间复杂度为 O（1），但相比于串行，多了 n! 个线程的开销。

2.2.3 使用改进的并行回溯法求解数独问题

在串行化算法的基础上，对第一个字符映射到 9 个不同的 matrix，然后创建 9 个线程分别运行串行化算法，最后在 9 个结果中选择其中正确的结果，该结果即为正确的并行化回溯法结果。

时间复杂度为 O（（n-1）!），虽然有 9 个原 matrix 的副本，但由于大小不变，因此空间复杂度还是 O（1）。

2.3 实验方案

使用图 2.1 所示数独，则该数独最后结果唯一，如图 2.2 所示

5	3	-	-	7	-	-	-	-
6	-	-	1	9	5	-	-	-
-	9	8	-	-	-	-	6	-
8	-	-	-	6	-	-	-	3
4	-	-	8	-	3	-	-	1
7	-	-	-	2	-	-	-	6
-	6	-	-	-	-	2	8	-
-	-	-	4	1	9	-	-	5
-	-	-	-	8	-	-	7	9

图 2.1 测试数独

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

图 2.2 数独正确结果

2.4 实验结果与分析

2.4.1 使用回溯法解决数独问题

实验结果如图 2.3 所示

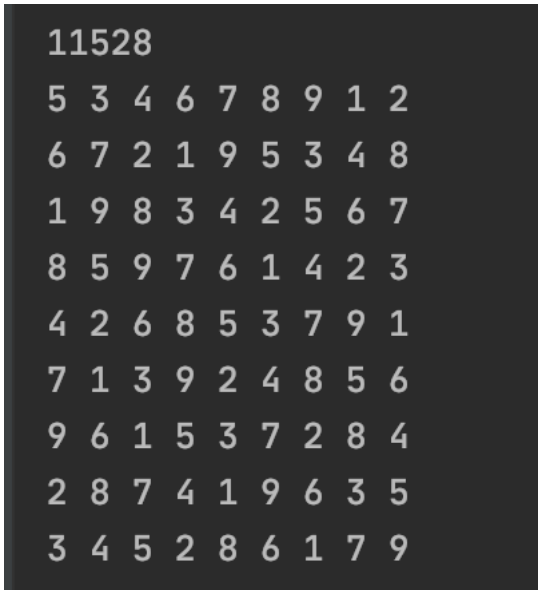

图 2.3 串行测试结果

第一行为程序执行时间，后续为数独结果，与预期一致，可见算法正确。

关键代码如下：

```
bool check(vector<vector<char>> &matrix, int x, int y){
    bool flag = true;
    for (int i = 0; i < 9; ++i) {
```

```
            if (matrix[x][i] == matrix[x][y] && i != y){
                flag = false;
            }
            if (matrix[i][y] == matrix[x][y] && i != x){
                flag = false;
            }
        }

        int xx = x/3;
        int yy = y/3;
        for (int i = xx*3; i < (xx+1)*3; ++i) {
            for (int j = yy*3; j < (yy+1)*3; ++j) {
                if (i != x && j != y && matrix[i][j] == matrix[x][y]){
                    flag = false;
                }
            }
        }

        return flag;
    }

    void BT(vector<vector<char>> &matrix, int x, int y, int &sum){
        if (sum == 0)
            return;
        if (matrix[x][y] == '-'){
            for (int i = 1; i < 10; ++i) {
                matrix[x][y] = i + '0';
                sum--;
                if (check(matrix,x,y)){
                    BT(matrix,x+(y+1)/9,(y+1)%9,sum);
                    if (sum == 0)
                        return;
                }
                    matrix[x][y] = '-';
                    sum++;
            }
        }
        else
            BT(matrix,x+(y+1)/9,(y+1)%9,sum);
    }

void solveSudoku(vector<vector<char>> &matrix) {
    //to do
```

```
    int sum = 0;
    for(int i = 0; i < 9; i++){
        for(int j = 0; j < 9; j++){
            if(matrix[i][j] == '-')
                sum++;
        }
    }
    BT(matrix,0,0,sum);
}
```

2.4.2 使用并行回溯法求解数独问题

实验结果如图 2.4 所示

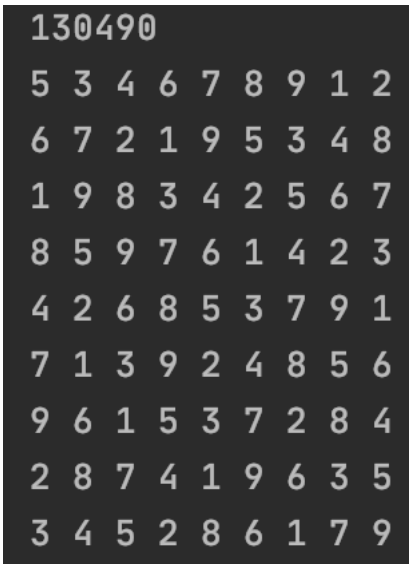

图 2.4 并行测试结果

第一行为程序执行时间，后续为数独结果，与预期一致，可见算法正确。

时间较串行增长是因为多了很多线程开销，但最后数独结果还是正确的。check 函数与 2.4.1 一致，关键代码如下：

```
void BT(vector<vector<char>> &matrix, int x, int y, int &sum){
        if (sum == 0)
            return;

        if (matrix[x][y] == '-'){
            auto trace = [&](){
                for (int i = 1; i < 10; ++i) {

                    matrix[x][y] = i + '0';
                    sum--;
                    if (check(matrix,x,y)){
                        BT(matrix,x+(y+1)/9,(y+1)%9,sum);
                        if (sum == 0)
                            return;
```

```
                    }
                    if (sum == 0 && check(matrix,x,y))
                        return;
                    matrix[x][y] = '-';
                    sum++;

                }
            };
            thread thread(trace);
            thread.join();
        }
        else
            BT(matrix,x+(y+1)/9,(y+1)%9,sum);
    }
```

2.4.3 使用改进的并行回溯法求解数独问题

实验结果如图 2.5 所示

图 2.5 并行优化测试结果

第一行为程序执行时间，后续为数独结果，与预期一致，可见算法正确。

时间相较串行增长是因为做了映射，并且最后需要对九个映射进行确认，但数独最终结果还是正确的，并且相较于之前的并行化有不小的提升，原因在于增大了粒度，只在第一次分裂时产生新任务，减少了线程开销。

关键代码如下：

```
    void BT(vector<vector<char>> &matrix, int x, int y, int &sum){
        if (sum == 0)
            return;

        if (matrix[x][y] == '-'){
            auto trace = [&](){
                for (int i = 1; i < 10; ++i) {
```

```
                    matrix[x][y] = i + '0';
                    sum--;
                    if (check(matrix,x,y)){
                        BT(matrix,x+(y+1)/9,(y+1)%9,sum);
                        if (sum == 0)
                            return;
                    }
                    matrix[x][y] = '-';
                    sum++;


                    if (sum == 0 && check(matrix,x,y))
                        return;
                }
            };
            trace();
        }
        else
            BT(matrix,x+(y+1)/9,(y+1)%9,sum);
}

bool check_multi_matrix(vector<vector<char>> &matrix){
    for(int i = 0; i < 9; i++){
        for(int j = 0; j < 9; j++){
            if(!check(matrix,i,j))
                return false;
        }
    }
    return true;
}

void solveSudoku(vector<vector<char>> &matrix) {
    //to do

    int sum = 0;
    vector<vector<vector<char>>> multi_matrix(9,matrix);
    for(int i = 0; i < 9; i++){
        for(int j = 0; j < 9; j++){
            if(matrix[i][j] == '-')
                sum++;
        }
    }
    int flag = 0;
```

```
        for(int i = 0; i < 9; i++){
            for(int j = 0; j < 9; j++){
                if (flag == 1)
                    break;
                if(matrix[i][j] == '-'){
                    for (int k = 0; k < 9; ++k) {
                        multi_matrix[k][i][j] = k+1+'0';
                    }
                    flag = 1;
                }
            }
        }
        sum--;

        vector<int>sums(9,sum);
        thread threads[9];
        for (int i = 0; i < 9; ++i) {
            threads[i] = thread(BT,ref(multi_matrix[i]),0,0,ref(sums[i]));
        }
        for (int i = 0; i < 9; ++i) {
            threads[i].join();
        }
        for(int i = 0; i < 9; i++){
            if(check_multi_matrix(multi_matrix[i])){
                for (int j = 0; j < 9; ++j) {
                    for (int k = 0; k < 9; ++k) {
                        matrix[j][k] = multi_matrix[i][j][k];
                    }
                }
                break;
            }
        }
    }
```

实验小结

本次实验通过串行和并行方法分别解决了汉诺塔问题和数独问题，这也是大学三年第一次使用多线程的方式解决实际问题，虽然简单，但还是收获不小。

除了多线程以外，还了解到了 Open MP 和 MPI 的并行方法。Open MP 可以简化多线程的使用，不用复杂的创建线程语句，而 MPI 在我看来更多的是一种可以参考的设计模式：设计一个主线程，但不参与任务计算，而是汇总其他子线程的计算结果，最后统一交付；其他子线程只负责自己的计算，与其他子线程互不干扰。这种方式应该可以应用在网络中解决一些主从关系的问题。

对于汉诺塔问题，并行化确实提高了程序效率。

而对于数独问题，我认为可能还是数据量不够大，因此无法体现并行的高效。

总而言之，对于可并行程度不高，且数据量较小的程序，增加并行方法只能是增加开销，反之才能体现并行好处。