华中科技大学

课程设计报告

题目： 并行编程原理与实践

院　　系：计算机科学与技术学院
姓　　名：张睿
专业班级：校交 1902
学　　号：U201912633
指导教师：金海

报告日期：2022/7/10

计算机科学与技术学院

目录

实验一 并行排序算法实践 1
1 实验目的与要求 1
2 算法描述 1
2.1 pthread 1
2.2 openMP 2
2.3 MPI 2
3 实验方案 2
3.1 开发及运行环境 2
3.2 串行程序设计 3
3.3 Pthread 4
3.4 OpenMP 5
3.5 MPI 5
4 实验结果与分析 7
4.1 时间复杂度分析 7
4.2 并行方法比较 7
实验二 杨辉三角 9
1. 实验目的与要求 9
2. 算法描述 9
2.1 openMP 9
2.2 MPI 9
3. 实验方案 10
3.1 开发及运行环境 10
3.2 串行程序设计 10
3.3 openMP 11
3.4 MPI 11
4. 实验结果与分析 13
实验小结 14
1. 主要工作 14

2. 心得体会 ····· 14

实验一 并行排序算法实践

1 实验目的与要求

在多种并行环境下（pthread、OpenMP、MPI）实现排序算法。

2 算法描述

以归并排序为主，在串行、并行条件下实现归并排序，归并排序得原理如下图所示：

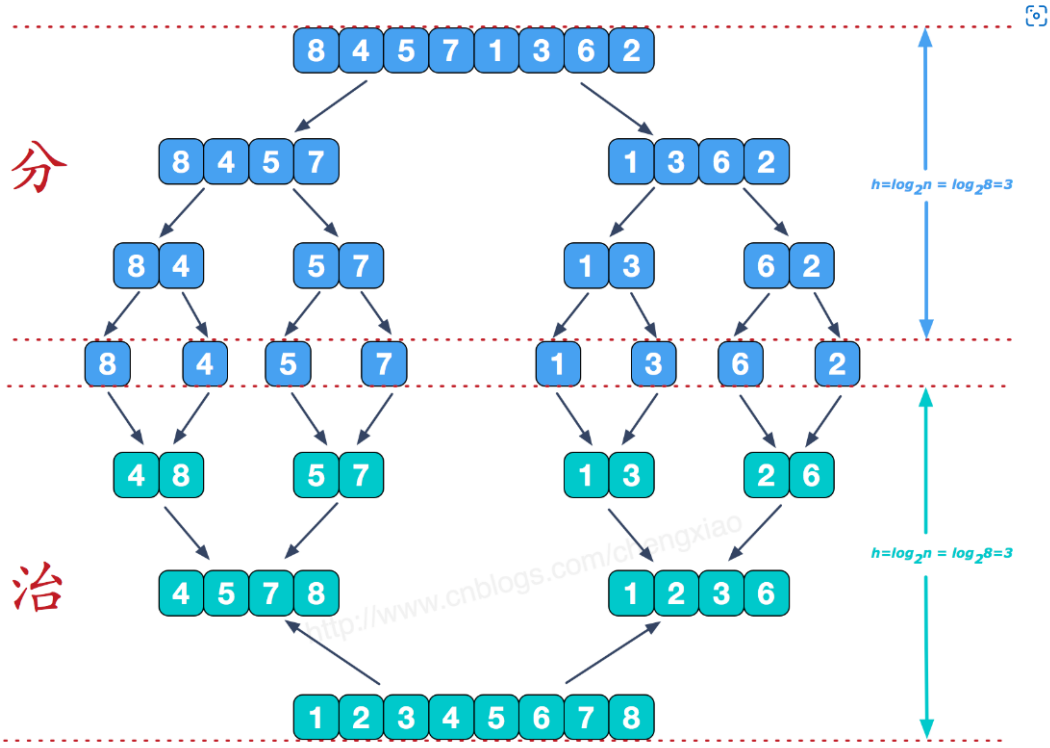

图 0.1 归并排序原理（图源）

可以看到，归并排序本身就是一种分而治之的思想，十分适合并行处理，这里主要选用归并排序完成实验。

2.1 pthread

由上图可知，当把每两部分分开排序时，可以看到其彼此之间时不互相依赖的，所以可以简单的直接在本层创建两个线程去实现不同部分的排序，然后排序完再 join 后进行 merge 的操作。

2.2 openMP

对于 openMP 来说，个人主要熟悉的是其针对于 for 循环的优化，由于我们采用的是归并排序的递归实现，所以其实并不是很容易的使用 openMP 进行并行。但是，我们重新观察这个问题，可以发现，其实使用递归排序的非递归实现便可以解决这个问题，主要是有了 for 循环的出现。
由上图可知，我们只需要自底向上进行处理，每一层都是一个内部没有什么依赖的 for 循环，这个 for 循环我们便可以使用 openMP 进行并行。

2.3 MPI

MPI 实现归并排序较为复杂，一方面是奇偶的问题较难解决，另一方面是是不同线程用同一套完整代码，代码较为复杂，这里采用对快排进行 MPI 的并行化。

对于快排来说，我们需要考虑进程个数的影响，分为 2 的整数幂和非整数幂。

1. 进程数为 2 的整数幂
   对于进程数为 2 的整数幂，由快排的特性我们可以知道，最多需要分发 $n = \log_2 进程数$ 的次数，所以可想而知，对于进程数为 2 的整数幂的情况来说，进程数是可以正常分配需要的数据的。
   所以我们采用一种主从式的结构，由 0 号进程进行输入和输出，意味着，其它进程在处理完后需将信息传给 0 号进程。

2. 进程数不为 2 的整数幂
   对于进程数不为 2 的整数幂，情况会变得复杂一点，可假设有 5 个进程数，那么需要分发 3 次，可想而知会有某个进程发现无法进一步分发数据，所以这里我们采用进程数循环减 1 的方式，找到可用的进程进行数据分配

3 实验方案

3.1 开发及运行环境

开发及运行环境均采用 educoder，其中配置了虚拟机以供使用。

3.2 串行程序设计

对于串行归并排序，主要是两个函数，一个函数(merge sort) 负责将数据分开来，这里采用递归的方式实现，另一个函数 merge，负责将数据按照大小顺序合并起来。

```
void merge(int *a, int low, int mid, int high){
    int temp[1005];
    int i=low,j=mid+1,k;
    for(k = 0; k < (high-low+1); k++) {
        if(i >= mid+1) break;
        if(j >= high+1) break;
        if(a[i] < a[j]){
            temp[k] = a[i];
            i++;
        }else{
            temp[k] = a[j];
            j++;
        }
    }

    while(i<=mid) {
        temp[k] = a[i];
        k++;
        i++;
    }

    while(j <= high){
        temp[k] = a[j];
        k++;
        j++;
    }

    for(int k=0;k<(high-low+1);k++) {
        a[low+k] = temp[k];
    }
    return;
}
```

```
void merge_sort(int *a, int low, int high) {
    if(high <= low){
        return;
    }
    int mid = (high+low)>>1;
    merge_sort(a, low, mid);
    merge_sort(a, mid+1, high);
    merge(a, low, mid, high);
    return;
}
```

3.3 Pthread

对于 pthread 并行，主要是将 merge_sort 中的两个递归的 merge_sort 以开线程的方式展开，然后在合并前 join 即可。

```
void merge_sort(void *arg) {
    Params *argst = (Params*)arg;
    int high = argst->high;
    int low = argst->low;
    if(high <= low){
        return;
    }
    int mid = (high+low)>>1;
    pthread_t th1, th2;
    Params pm1,pm2;
    pm1.a = argst->a;
    pm2.a = argst->a;
    pm1.low = low;
    pm1.high = mid;
    pm2.low = mid+1;
    pm2.high =high;
    pthread_create(&th1, NULL, merge_sort, &pm1);
    pthread_create(&th2, NULL, merge_sort, &pm2);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    merge(a, low, mid, high);
    return;
}
```

这里需要注意的是参数的传递，需要如图所示将参数打包为一个结构体进行传递。

3.4 OpenMP

对于 openMP 我们需要将归并排序写成非递归形式，并对每层 for 循环进行并行。

```
for (i = 2; i < high; i *= 2) {
    #pragma omp parallel for private(j) shared(high, i)
    for (j = 0; j < high-i; j += i*2) {
        merge(j, j+i, (j+i*2 <= high ? j+i*2 : high), data, temp);
    }
}
```

这里需要注意的是上界的处理。

3.5 MPI

在快排的基础上，根据对应算法描述节实现函数 parallelQS 函数即可。

```
void parellelQS(int* data, int low, int high, int m, int id, int nowID, int N)
{
  int i, j, r = high, max_l = -1; //r 表示划分后数据前部分的末元素下标，max_l 表示后部分数据的长度
  int* t;
  MPI_Status status;
  if (m == 0) {  //无进程可以调用
    if (nowID == id) quicksort(data, low, high);
    return;
  }
  if (nowID == id) {  //当前进程是负责分发的
    while (id + exp_2(m - 1) > N && m > 0) m--;  //寻找未分配数据的可用进程
    if (id + exp_2(m - 1) < N) { //还有未接收数据的进程，则划分数据
      r = partition(data, low, high);
      max_l = high - r;
      MPI_Send(&max_l, 1, MPI_INT, id + exp_2(m - 1), nowID, MPI_COM
```

```
M_WORLD);
        if (max_l > 0)  //id 进程将后部分数据发送给 id+2^(m-1)进程
          MPI_Send(data + r + 1, max_l, MPI_INT, id + exp_2(m - 1), nowID,
MPI_COMM_WORLD);
      }
    }
    if (nowID == id + exp_2(m - 1)) {   //当前进程是负责接收的
      MPI_Recv(&max_l, 1, MPI_INT, id, id, MPI_COMM_WORLD, &status);
      if (max_l > 0) {   //id+2^(m-1)进程从 id 进程接收后部分数据
        t = (int*)malloc(max_l * sizeof(int));
        MPI_Recv(t, max_l, MPI_INT, id, id, MPI_COMM_WORLD, &status);
      }
    }
    j = r - 1 - low;
    MPI_Bcast(&j, 1, MPI_INT, id, MPI_COMM_WORLD);
    if (j > 0)    //负责分发的进程的数据不为空
      parellelQS(data, low, r - 1, m - 1, id, nowID, N);  //递归调用快排函数，对
前部分数据进行排序
    j = max_l;
    MPI_Bcast(&j, 1, MPI_INT, id, MPI_COMM_WORLD);
    if (j > 0)    //负责接收的进程的数据不为空
      parellelQS(t, 0, max_l - 1, m - 1, id + exp_2(m - 1), nowID, N);  //递归调用
快排函数，对前部分数据进行排序
    if ((nowID == id + exp_2(m - 1)) && (max_l > 0))    //id+2^(m-1)进程发送结
果给 id 进程
      MPI_Send(t, max_l, MPI_INT, id, id + exp_2(m - 1), MPI_COMM_WORL
D);
    if ((nowID == id) && id + exp_2(m - 1) < N && (max_l > 0))    //id 进程接收
id+2^(m-1)进程发送的结果
      MPI_Recv(data + r + 1, max_l, MPI_INT, id + exp_2(m - 1), id + exp_2(m -
 1), MPI_COMM_WORLD, &status);
}
```

4 实验结果与分析

4.1 时间复杂度分析

1. 串行

设 n 为待排序数组中的元素个数， Tn 为算法需要的时间复杂度，则有

$$T(n)=\begin{cases}0 & n\le 1\\ T(\frac{n}{2})+T(\frac{n}{2})+C(n) & n>1\end{cases}$$

其中 T(n/2) 是将两个含有 n 个数据元素的序列合并为一个含有 n 个数据元素的序列所需要的比较操作次数。

由主方法分析可得，其时间复杂度为$O(nlogn)$

空间复杂度分析: $O(n)$,需要一个额外的 n 维数组.

2. Pthread

对于 pthread，其时间复杂度没有太大的变化，主要的提升在于让原本先后进行的两路同时执行。

3. openMP

对于 openMP，其时间复杂度也没有太大的变化，主要提升在于自下而上进行归并时，每一层的结点相当于同时进行。

4. MPI

由于 MPI 涉及进程之间的通信，其复杂度较难分析。

4.2 并行方法比较

1. 使用难易度：

Pthread 需要显式地明确每个线程的行为；MPI 需要指定合理的消息传递策略并通过相应接口在各进程间进行消息传递；OpenMP 只需要简单声明这块代码并行执行。在实现过程中能明显感受到 OpenMP 框架对并行算法的抽象度高、无需配置即可在 gcc 中使用，使程序员通过一些 pragma

指令和函数即可让编译器能在合适的地方插入线程，甚至无需懂得创建和销毁线程的细节就能写出多线程化程序，可以把重点放在决定哪里需要多线程和优化数据结构上；

2. 环境要求：

   Pthread 为 Linux 原生库，只需包含头文件即可，可移植性强。MPI 相关的函数声明与数据结构的定义包含在 mpi.h 中，但需要安装如 MPICH 等工具构建 MPI 编程环境。openMP 则对编译器提出了要求。

3. 通信方式：

   OpenMP 和 pthread 基于共享内存方式，而 MPI 基于进程间通讯。

实验二 杨辉三角

1. 实验目的与要求

使用串行和两种并行方式（OpenMP 和 MPI）实现打印杨辉三角形的 C 语言程序。

2. 算法描述

杨辉三角本身的算法较为简单，只需要模拟其构成的算法，即当前元素的值为上一行的左右元素之和，用公式表述为 a[i][j] = a[i-1][j]+a[i-1][j+1]，模拟即可。在此基础上可以做进一步的优化，由递推公式可知当前的数据之和上一行有关，所以可以用滚动数组进行优化，每次只保存一行的数据，递推公式可简化为 a[i] = a[i]+a[i+1], 根据这个递推公式就可写出其串行程序。

2.1 openMP

对于 openMP 的并行方式，可以考虑 for 循环的并行，由于算法本身的限制，层与层之间互相依赖，无法进行并行，但层内则可以通过消除依赖实现并行。层内的依赖主要产生于滚动数组的使用，由于 a[i] 和 a[i+1]相关，无法同时进行更新，要解决这个依赖，我们只需使用两个数组，交替使用即可消除。消除后便可用 parellel for 对 for 循环进行并行。

2.2 MPI

对于 MPI 的并行方式，和 openMP 不同，由于这里采用的通信的方式，所以我们使用主从的结构，用 id 为 0 的结点接收并打印相关数据，用其它结点进行计算。对于其它结点的计算，这里不能简单的模拟杨辉三角的构成，而是使用通项公式来进行计算。n 行 m 列的元素通项公式如下：

$$C_{n-1}^{m-1}=\frac{(n-1)!}{(m-1)!\times(n-m)!}$$

可以看到这是一个组合数。只需要根据进程数，分配任务，根据通项公式计算即可。

3. 实验方案

3.1 开发及运行环境

开发及运行环境均采用 educoder，其中配置了虚拟机以供使用。

3.2 串行程序设计

根据对应算法描述节用滚动数组逐行输出并计算杨辉三角：

```
int main()
{
    int n;
    scanf("%d", &n);
    printf("[[1]");
    int a[2][100];
    a[0][0] = 1;
    int flag = 0;
    for(int layer=2;layer<=n;layer++) {
        for(int i=0;i<layer;i++) {
            if(i==0) a[flag][i] = 1;
            else if(i==layer-1) a[flag][i] = 1;
            else{
                a[flag][i] = a[1-flag][i-1]+a[1-flag][i];
            }
        }
        printf(",[");
        for(int i=0;i<layer-1;i++){
            printf("%d,", a[flag][i]);
        }
        printf("%d]", a[flag][layer-1]);

        flag = 1 - flag;
    }
    printf("]");
    return 0;
```

3.3 openMP

根据对应算法描述节拆分滚动数组更新循环并将其并行化：

```
for(int layer=2;layer<=n;layer++) {
    #pragma omp parallel for
    for(int i=0;i<layer;i++) {
        if(i==0) a[flag][i] = 1;
        else if(i==layer-1) a[flag][i] = 1;
        else{
            a[flag][i] = a[1-flag][i-1]+a[1-flag][i];
        }
    }
```

3.4 MPI

根据对应算法描述节拆分滚动数组更新循环并将其并行化：

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>
#include <math.h>
#include <string.h>
/***** Begin *****/

void  main(int argc, char* argv[])
{
   int numprocs, myid;
   MPI_Status status;
   int i,j;
   int a[100] = {0};
   int n;
   int k = 1;
   int m = 1 ;
   int tmp;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myid);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

   if(myid == 0) {
      scanf("%d",&n);
```

```
    MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
    printf("[");
    for (int layer = 1; layer<=n; layer++) {
      printf("[");
      tmp = layer % 3;
      if(tmp == 0) tmp = 3;
      MPI_Recv(a, 10, MPI_INT, tmp , 99,
        MPI_COMM_WORLD, &status);
      for(int j = 1 ; j <= layer ; j++) {
        printf("%d",a[j]);
        if( j != layer)
          printf(",");
      }
      printf("]");
      if(layer != n)
        printf(",");
    }
    printf("]");
  }

  if(myid) {
    MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
    for(i = myid ; i <= n ; i += numprocs - 1 ) {
      for(j = 1 ; j <= i ; j ++) {
        a[j] = k / m;
        k *= i - j;
        m *= j ;
      }
    m = 1;
    k = 1;
    MPI_Send(a, 10, MPI_INT, 0, 99,
      MPI_COMM_WORLD);
    }
  }
  MPI_Finalize();

}


/***** End *****/
```

4. 实验结果与分析

尽管优化了核心的计算步骤，但不管是 openMP 还是 MPI 都没有对复杂度有明显的降低，只有常数级别的优化，主要是因为层与层之间的依赖较大无法消除。

实验小结

1. 主要工作

1. 基于不同的并行方式（pthread、openMP、MPI）实现了排序算法的并行化和杨辉三角的计算。
2. 比较了不同的并行方式的效率和差异。

2. 心得体会

这次并行编程实验还是比较困难的，主要有以下两方面的原因：
1. 对并行编程的设计不熟，导致将串行算法并行化较为困难。
2. 对并行编程相关的函数不熟，需要反复查阅。

虽然花了较多的时间，个人感觉也学到了许多东西，特别是把串行算法并行化的这个步骤，不仅熟悉了相关的函数语法，更重要的是让我学会了从并行的特点去看待问题，怎么样去分解，怎么样去通信，都有考虑到，也学会了如何将一个算法改到能够并行化的形式，这样的思想我感觉是我从实验中学到的最宝贵的经验。

至于课程建议主要有以下两个方面，一方面是希望实验课可以提前，希望理论和实验能够相辅相成，不仅能够节约期末的时间，也可以和理论课互相弥补，从多方面让我们更加深入地了解并行算法，说实话现在理论课如果不回看已经忘得七七八八了。另一方面，希望可以增大一点实验地难度，课程中讲了并行的四个步骤，我觉得是非常重要的思想，但是并没有在实验中体现，个人觉得这是比较遗憾的。

最后，感谢金海老师一个学期的教导，以及实验课老师对实验平台的搭建，希望并行编程这门课可以越来越好。