華中科技大學

本科生毕业设计（论文）开题报告

并行编程与设计实验报告

院　　系　计算机科学与技术学院

专业班级　计算机校交 1902 班

姓　　名　Sukuna

学　　号　U2019114514

指导教师　金海

2023 年 1 月 18 日

目　录

一　实验一

1　串行环境下的排序算法

1.1　实验目的与要求

在串行环境下编写串行排序的 C 语言小程序，并按要求输出对应的排序结果。串行排序方法任选。

1.2　算法描述

在本实验中我使用的排序算法为快速排序，有关快速排序算法的描述见算法1

Algorithm 1 快速排序

Input: 数组的个数 n 以及 n 个数组元素

Output: 升序排序之后的数组

1: 首先设定一个分界值，通过该分界值将数组分成左右两部分

2: 将大于或等于分界值的数据集中到数组的右边，小于分界值的数据集中到数组的左边。此时，左边部分中各元素都小于分界值，而右边部分中各元素都大于或等于分界值

3: 左边和右边的数据可以独立进行排序。对于左侧的数据，又可以取一个分界值，将该部分数据分成左右两部分，右侧的数据也可以做类似的处理

4: 重复上述过程，直到排序完成

1.3　复杂度分析

快速排序通过 partition 操作将长度将长度为 n 的数组分成 3 部分，假设左边有 i 个元素，右边有 n-1-i 个元素，中间的基准值为 pivot。假设所有可能的输入等概率出现，那么所有的划分情况也会等概率出现，即 i 会以相等的概率取区间

[0, n-1] 中的每一个值，所以快速排序的平均时间复杂度为

$$A[n] = \underbrace{n-1}_{cost\ of\ partition} + \underbrace{\sum_{i=0}^{n-1}\frac{1}{n}[A(i)+A(n-1-i)]}_{cost\ of\ left\ and\ right\ parts}, n \geq 2$$

递归的初始情况：A[1]=A[0]=0，另外可以注意到左右两部分递归具有对称性，所以有

$$\sum_{i=0}^{n-1} A(i) = \sum_{i=0}^{n-1} A(n-1-i)$$

于是上述公式可以简化成如下形式：

$$A(n) = (n-1) + \frac{2}{n}\sum_{i=0}^{n} A(i)$$

于是到了熟悉的数列求通项公式环节，以下为数学推导，采用错位相减法

$$nA(n) - (n-1)A(n-1) = n(n-1) + 2\sum_{i=0}^{n-1} A(i)$$

$$\Longrightarrow nA(n) = (n+1)A(n-1) + 2(n-1)$$

$$\Longrightarrow \frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

考虑初始情况 B(1)=B(0)=0，接下来求解这个数列通项

$$B(n) = B(n-1) + \frac{4}{n+1} - \frac{2}{n}$$

$$\Longrightarrow B(n) - \frac{2}{n+1} = B(n-1) - \frac{2}{n} + \frac{2}{n+1}$$

令 $C(n) = B(n) - \frac{2}{n+1}$，于是有 $C(n) = C(n-1) + \frac{2}{n+1}$，初始情况为 $C(1) = -1, C(0) = -2$，所以 $C(n) = \sum_{i=0}^{n} \frac{2}{i+1} = O(logn)$，所以 $B(n) = O(logn)$,$A(n) = (n+1)B(n) = O(nlogn)$，从而我们得知了快速排序的时间复杂度为 $O(nlogn)$

程序中仅开辟了一个一维数组对数据进行保存，易知快速排序程序的空间复杂度为 $O(n)$。

1.4 实验结果和分析

输入数据：8 49 38 65 97 76 13 27 49

快速排序过程如下：

$\{49\ , 38, 65, 97, 76, 13, 27, 49\}$

第一次划分之后:

$\{27, 38, 13\}, 49, \{76, 97, 65, 49\}$

左部数据继续排序

$\{13\}, 27, \{38\}, 49, \{76, 97, 65, 49\}$

右部数据继续排序

$13, 27, 38, \{49, 65\}, 76, \{97\}$

$13, 27, 38, 49, \{65\}, 76, 97$

最后得到升序序列 $\{13, 27, 38, 49, 49, 65, 76, 97\}$

2 pthread 环境下的并行排序算法设计

2.1 实验目的与要求

编写使用多线程排序算法的 C 语言小程序，并按要求输出对应的排序后的结果。本关要求我们掌握进程和线程的概念以及 pthread 的使用方法。

2.2 算法描述

使用 pthread 多线程并行执行快速排序的基本算法流程见算法2。基本的思路是：使用 pthread 实现快速排序，在串行快速排序的基础上，将数组分成 n 段，使用 n 个线程并行对每段进行快速排序处理，最后合并所有段得到最后的排序结果。

需要注意的是，本实验中需使用 pthread_barrier 函数保证所有线程都在分段排好序之后，进行类似于归并排序的操作。由于每个段的长度可能会不一样，因此需要定义两个数组分别保存每个段在数组中的起始位置和结束位置。由于在创建线程时需要将串行排序函数以及串行排序函数的参数地址作为创建线程函数的参数，基本的快速排序函数需要接收多个参数，因此将这些参数使用结构体进行封装，在准备参数的时候，将其放到创建好的参数结构体中，把结构体指针作为创建线程函数的参数传入即可。然而基本的串行快速排序函数没办法解析以结构体方式传入的参数，因此需要把对基本的快速排序函数的调用封装到 pthread_sort 函数中，在这个函数里面首先解析参数结构体里面的字段，然后将特定的参数送至基本的快速排序函数，对其进行调用。

Algorithm 2 多线程快速排序算法流程

Input: 数组的个数 n 以及 n 个数组元素, 线程数 N

Output: 升序排序之后的数组

1: 接收用户输入，进行数组的初始化; 初始化一个 pthread_barrier_t 变量, 以便后续进行线程的同步操作；进行数组的分段操作，计算分段的数目为 $S = \frac{n}{N}$

2: 将每个分段的起止索引进行保存, 第 i 个分段 $(0 \leq i < S)$ 的起止索引为 $[i * S, (i+1) * S - 1], i = S$ 也就是最后一个分段 $[i * s, n - 1]$。

3: 为每个分段创建一个线程，对每个分段数组进行快速排序操作。等待所有分段排序完成之后进行下一步

4: 对每一个已经排好序的段进行合并。合并策略是：为每个分段设置一个指针，初始时都指向每个分段的第一个元素，然后每次选取这些指针所指元素中最小的一个放至到结果集，被选择的指针需要向后移动，然后继续进行比较操作，直到所有的段合并完成。

2.3 复杂度分析

我们已经证明了快速排序的时间复杂度为 $O(nlogn)$，由于使用多线程，线程数为 N，每一个数组分段的规模为 $S = \frac{n}{N}$，则每一个数据段的时间复杂度为 $O(SlogS) = O(\frac{n}{N}log\frac{n}{N})$，N 个段并行进行排序的时间复杂度就是 $O(\frac{n}{N}log\frac{n}{N})$。最后还需要 $O(n)$ 的时间对 N 个分段进行合并，因此总的时间复杂度是 $O(\frac{n}{N}log\frac{n}{N}) + O(n)$，多线程并行执行的时间与线程个数 N 以及数组元素个数 n 的相对大小有关，但是其时间复杂度的上界仍然是 $O(nlog(n))$。

对于空间复杂度，一个一维数组保存数组元素，两个一维数组 $(element\ number < n)$ 保存每个分段的在原数组中的起止位置，因此空间复杂度为 $O(n)$。

2.4 实验结果和分析

假设使用线程的数目为 3，输入数据：8 49 38 65 97 76 13 27 49

排序过程过程如下：

首先进行分段，分段数目为 3：

$\{49, 38\}, \{65, 97\}, \{76, 13, 27, 49\}$

对每个分段进行快速排序之后：

$\{38, 49\}, \{65, 97\}, \{13, 27, 49, 76\}$

对每个分段进行合并之后：

$\{13, 27, 38, 49, 49, 65, 76, 97\}$

最后得到升序序列为 $\{13, 27, 38, 49, 49, 65, 76, 97\}$

3 OpenMP 环境下的并行排序算法设计

3.1 实验目的和要求

编写使用 OpenMP 的 C 语言小程序，并按要求输出对应的排序后的结果。本关要求我们掌握什么是 openMP 以及使用 openMP 进行并行编程的方法。

3.2 算法描述

OpenMP 是一种用于共享内存并行系统的多线程程序设计方案,OpenMP 提供了对并行算法的高层抽象描述，提供多种编译制导指令，这些指令以 #pragma omp 开头，后边跟具体的功能指令，格式为：#pragma omp 指令 [子句][, 子句]...。我们在编程时只需要在顶层设计中给出并行化执行策略，而无需考虑底层的实现。支持 OpenMP 的编译环境能够根据程序中添加的 pragma 指令，自动将程序进行并行的处理。

在本实验中我使用了如下几个编译制导指令：

(1) `#pragma omp parallel`：该编译制导指令用一个结构块之前，表示这段代码块将被多个线程并行执行。

(2) `#pragma omp task firstprivate(list)`：该编译制导指令中的 `task` 指令表示定义一个显示的任务，线程组内的某一个线程会来执行此任务。`task` 的出现很好的解决了 `for` 和 `sections` 指令的“缺陷”：无法根据运行时的环境动态进行任务划分的，必须预先知道任务的划分情况。`task` 指令是动态定义任务的，在运行过程中，使用一个 `task` 就会定义一个任务，任务就会在一个线程上面去执行，`task` 还能够进行任务的嵌套定义，适用于递归的情况，对于快速排序非常适用。子句 `firstprivate` 用于指定 `list` 中的变量在每个线程中

都会有一份私有副本，且私有变量要在进入并行域或者任务分担域时，继承主线程中的同名变量作为初始值。使得 `list` 中的私有数据不受外部同名变量的影响。

(3) `#pragma omp single nowait`：`single` 选项是在并行块中使用的，它告诉编译器，接下来紧跟的代码将只会有一个线程执行；可能在处理多线程不安全代码时非常有用；`nowait` 子句指出并发线程可以忽略其他制导指令暗含的路障同步。

算法3中，仅对 OpenMP 的编译制导指令使用在哪些地方进行描述。其他部分不再赘述。

Algorithm 3 基于 OpenMP 的快速排序算法

Input: 数组的个数 n 以及 n 个数组元素, 线程数 N

Output: 升序排序之后的数组

1: 开设 8 个线程来执行并行代码段，将主程序中调用快速排序的语句块用花括号 {} 括起来，在花括号上面指定 #pragma omp parallel，表明并行执行快速排序程序

2: 在并行块中，调用快速排序函数的语句前指定 #pragma omp single nowait 表示主程序中对快速排序的总调用只能由第 0 号线程执行一次，其他线程不能执行。

3: 在快速排序程序的递归调用语句之前指定 #pragma omp task firstprivate(arr,start,keyIndex) 表示为递归子程序的调用定义任务，且为任务创建私有副本，其中 arr 是数组，start 是开始的索引，keyIndex 是基准值的索引

3.3 复杂度分析

下面仅根据我个人对于 OpenMP 程序的理解给出复杂度的分析过程，由于我不清楚对编译器会具体怎么对线程进行调度，所以我的理解不一定正确。

在算法描述部分已经给出的对 OpenMP 的编译制导指令的使用，可以知道最重要的一部分在快速排序的递归调用的任务定义上面。由于每一次递归调用都会定义一个任务，我们可以把这个递归调用想象成一棵树，如图1所示。

我觉得递归树上的每一行的所有任务是可以并行执行的，如果线程调度比

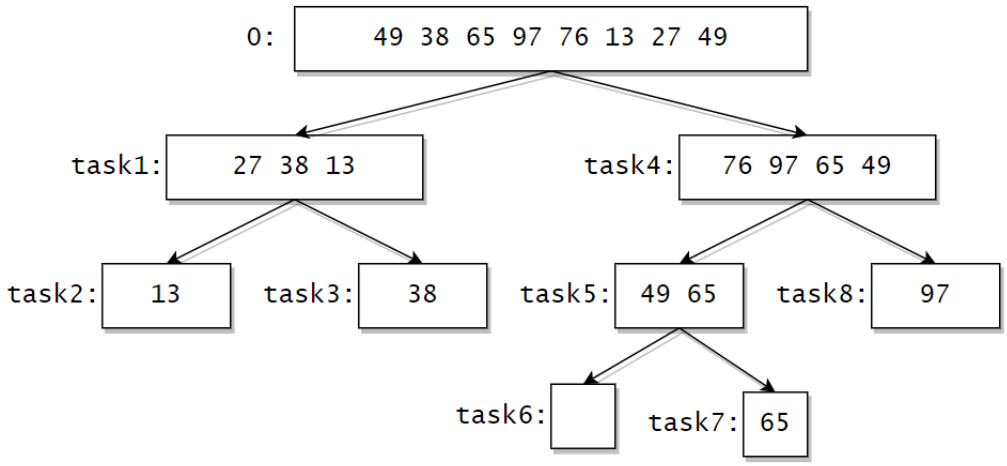

图 1　快速排序递归过程

较合理，排序程序的时间复杂度最优可以达到 $O(log_2 n)$。task 是“动态”定义任务的，在运行过程中，需要使用 task 就会定义一个任务，任务就会在一个线程上去执行，那么其它的任务就可以并行的执行。可能某一个任务执行了一半的时候，或者甚至要执行完的时候，程序可以去创建第二个任务，任务在一个线程上去执行，一个动态的过程。由于这些每个任务的执行时机不能确定，所以在并行化环境中时间复杂度可以达到 $O(n)$，但是如果线程的数目不能满足递归树上的每一个结点，时间复杂度会增大但一定 $\lesssim O(nlogn)$。

程序中仅开辟了一个一维数组对数据进行保存，空间复杂度为 $O(n)$。

3.4　实验结果和分析

仍然使用之前的输入: 8 49 38 65 97 76 13 27 49

但由于任务执行或者说线程的调度的不确定性，其中间执行过程不一定每一次都是一样的，但是不论中间过程是怎样的，总体执行的流程还是与图1所示的递归树一致。执行的结果是 $\{13, 27, 38, 49, 49, 65, 76, 97\}$

4　MPI 环境下的并行排序算法设计

4.1　实验目的与要求

编写使用 MPI 的 C 语言小程序，并按要求输出对应的排序后的结果。掌握 MPI 的基本知识以及使用方法。

4.2 算法描述

MPI 是一个跨语言（编程语言如 C, Fortran 等）的通讯协议，用于编写并行计算机。支持点对点和广播。MPI 是一个信息传递应用程序接口，包括协议和和语义说明，他们指明其如何在各种实现中发挥其特性。MPI 的目标是高性能，大规模性，和可移植性。MPI 在今天仍为高性能计算的主要模型。与 OpenMP 并行程序不同，MPI 是一种基于信息传递的并行编程技术。消息传递接口是一种编程接口标准，而不是一种具体的编程语言。简而言之，MPI 标准定义了一组具有可移植性的编程接口。

为了使用 MPI 的消息传递机制进行并行编程，我们在基本的快速排序程序上进行一层封装，在封装内部实现信息的通信以及对快速排序基本程序的调用。我们将这个函数命名为 $quicksort\_recursive$，算法流程见算法4。

Algorithm 4 quicksort_recursive

Input: 数组 arr、数组大小 arrSize、当前进程号 currProcRank、最大进程号 maxRank、当前进程所在层号 rankIndex

Output: 无

1: 计算与当前进程需要发送一半数据的进程号是多少；rankIndex 加 1 表示移到下一层；如果计算出来的进程号大于最大进程号（目标进程不可用），则自己调用快速排序函数对 arr 数组进行排序

2: 如果计算出的目标进程号是可用的，则利用 partition 函数对 arr 进行划分，获取划分的基准值在 arr 中的索引 pivotIndex。

3: 总是将两个划分中元素个数较少的子数组发送给目的进程；然后递归调用 quicksort_recursive，把剩下的子数组作为参数传入；最后接收被发出去的元素个数较少的子数组的数据（应当是已经排好序且通过某个进程发送过来的）。

下面对 main 函数的实现逻辑进行描述。总体的思路大致是通过 0 号进程进程数据的输入、数据最开始的分发以及数据最后的整合。

(1) 初始化 MPI 环境，获取当前进程号和进程数；

(2) 判断当前进程的进程号是否为 0，如是则读取用户输入数组的大小 n

以及数组元素（仅 0 号进程读取）；

(3) 对于所有的进程，计算进程号对应的递归树的层号，递归树的一个示例如图1所示。然后通过调用 MPI_Barrier 函数等待进程通信组中的所有进程到达这一点。

(4) 判断当前进程的进程号是否为 0，如是则调用快速排序子程序，将整个未排序的数组（unsorted_arr）、其大小（n）、其进程号（rank）、最大进程号（size-1）、0 号进程所在的层数（rankPow）作为参数；

(5) 如果是其他进程，则使用 MPI_Probe 函数先探测接收消息的内容，一旦检测到任何发送者发送给任何接收者的消息，使用结构体 status 接收有关消息的信息，包括其来源（source_process）和大小（subarray_size），其中大小使用 MPI_Get_count 函数通过传入 status 作为参数获得（实际上这里的消息就是子数组，大小就是子数组的元素个数）。利用获得的子数组的大小开辟一个缓冲区（subarray），然后使用 MPI_Recv 函数接收子数组数据，放至定义好的缓冲区中，对缓冲区中的子数组调用快速排序子程序进行排序，参数为子数组（subarray）、子数组大小（subarray_size）、当前进程号（rank）、最大进程号（size-1）、当前进程所在层数（rankPow），最后将排序后的子数组的数据发送给 0 号进程。

(5) 数组排序完成，结束 MPI 通信进程。

4.3 复杂度分析

使用 MPI 的消息通信机制的并行程序，其时间复杂度的大小与数组元素的个数和进程的数目有关，进程数目也不是越多越好，因为这样会加剧通信的开销以降低程序执行的效率。但不管怎样，该并行程序的时间复杂度一定 $\lesssim O(nlogn)$。

程序中仅开辟了一个一维数组对数据进行保存，空间复杂度为 $O(n)$。

4.4 实验结果和分析

假设指定的进程数目为 4 个（进程号为 0-3），用户输入的数组数据仍然使用包含 8 个元素的 49, 38, 65, 97, 76, 13, 27, 49。其执行过程如图2所示

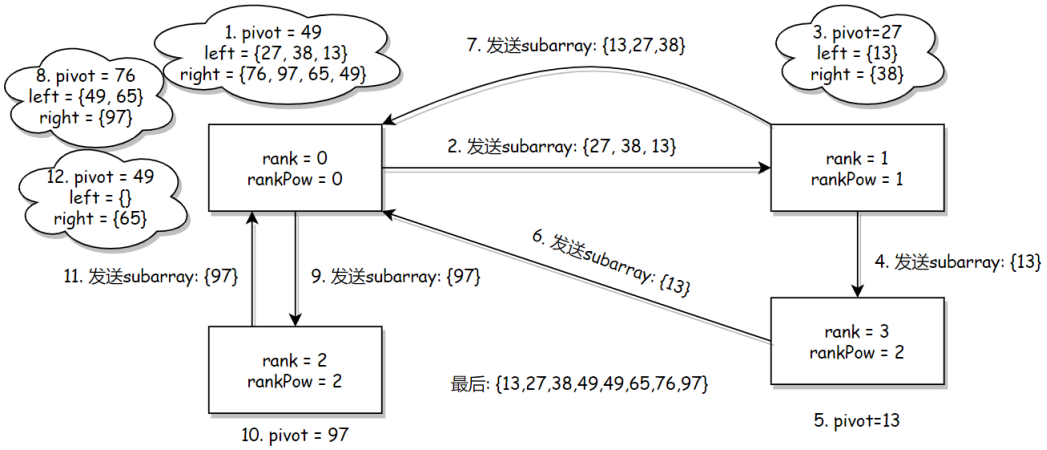

图 2　MPI 快速排序并行执行过程

二　实验二

1　串行环境下的杨辉三角输出

1.1　实验目的与要求

串行环境下编写打印杨辉三角形的 C 语言小程序，并按要求输出对应的杨辉三角形。本关需要掌握杨辉三角的基本格式。

1.2　算法描述

通过观察杨辉三角的数据排列方式，我们可以总结出以下规律：

(1) 杨辉三角的两个腰边上的数都是 1；

(2) 对于其他的位置：当前值 = 上一行前一列的值 + 上一行同一列的值。

串行环境下的杨辉三角的计算的算法描述见算法5。

Algorithm 5 串行环境下的杨辉三角输出

Input: 需要输出的杨辉三角的层数 n

Output: 计算并打印层数为 n 的杨辉三角

1: 初始化一个大小为 $n \times n$ 的二维数组 a，置 a[i][0] = 1，a[i][i] = 1，其中 $0 \leq i < n$，这个过程就是将杨辉三角的两个腰边上的数赋值为 1。

2: 从左到右，从上到下对其他位置上的数进行计算，计算的公式为 a[i][j] = a[i-1][j-1] + a[i-1][j]，其中 $2 \leq i < n \; and \; 1 \leq j < i$

3: 最后将杨辉三角按照格式进行输出即可。

1.3　复杂度分析

杨辉三角的空间复杂度等同于存储杨辉值的二维数组大小，所以空间复杂度为 $O(n^2)$。由于杨辉三角形中对于每个元素的计算都是常量时间，因此对于 n 层的杨辉三角形，其时间复杂度为 $1+2+3+...+n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$，即 $O(n^2)$。

1.4 实验结果和分析

假设用户输入的层数 n = 6，杨辉三角的计算过程和结果如图3所示。

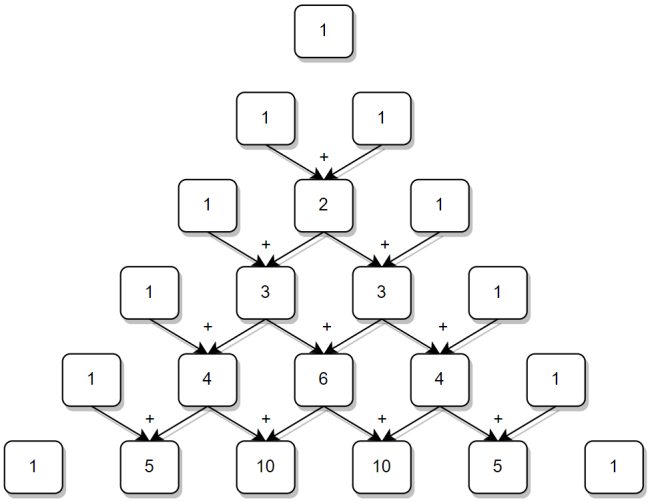

图 3　杨辉三角计算过程

2　OpenMP 环境下的杨辉三角输出

2.1 实验目的和要求

掌握 OpenMP 的基本知识和使用方法，编写使用 OpenMP 打印杨辉三角形的 C 语言小程序，并按要求输出对应的杨辉三角形。

2.2 算法描述

相比于实验一中的利用 OpenMP 实现并行快速排序，杨辉三角的计算过程可以并行的部分就在于两个 for 循环，因此我们需要使用能够对 for 循环有帮助的编译制导指令。对于指令 `#pragma omp parallel for`，它表示接下来的 for 循环将被多线程执行。

但是 OpenMP 对于需要并行化的 for 循环有如下要求：

(1) 循环的变量变量（就是 i）必须是有符号整形，其他的都不行；

(2) 循环的比较条件必须是 <、<=、>、>= 中的一种；

(3) 循环的增量部分必须是增减一个不变的值（即每次循环是不变的）；

(4) 如果比较符号是 <、<=，那每次循环 i 应该增加，反之应该减小；

(5) 循环不能从内部循环跳到外部循环，`goto` 和 `break` 只能在循环内部跳转，异常必须在循环内部被捕获

在串行环境下的杨辉三角形各个元素的计算方法已经在算法5中给出，其中包含两个 for 循环，一个 for 循环对两个腰边直接赋值，另一个 for 循环计算杨辉三角内部的各个元素。由于这两个 for 循环都符合 OpenMP 并行化的 for 循环条件，因此我们只需要在两个 for 循环语句的上面添加编译制导指令 `#pragma omp parallel for` 即可。不过在并行程序执行之前，首先需要设置执行并行代码的线程数目，我在实验的过程中设置的是 4 个线程。

2.3 复杂度分析

使用 for 制导语句会将 for 循环拆开来尽可能平均的分配到各个线程进行执行，因此假设线程的个数为 K，串行计算杨辉三角形的时间复杂度为 $O(n^2)$，则多个线程并行执行 for 循环的时间复杂度为 $O(\frac{n^2}{K}) \lesssim O(n^2)$

杨辉三角的空间复杂度等同于存储杨辉值的二维数组大小，所以空间复杂度为 $O(n^2)$。

2.4 实验结果和分析

假如用户希望输出的杨辉三角的层数为 6，无论线程如何分配执行，总体的执行流程仍然与图3一致。最后结果都会是 [[1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1], [1, 5, 10, 10, 5, 1]]。

3 MPI 环境下的杨辉三角输出

3.1 实验目的与要求

掌握 MPI 的基本知识和使用方法，编写使用 MPI 打印杨辉三角形的 C 小程序，并按要求输出对应杨辉三角形。

3.2 算法描述

对于如何使用 MPI 对杨辉三角进行并行计算，我参考了一个网站上面的例子，网址为 https://daugerresearch.com/vault。由于之前的串行计算杨辉三角的程序不好拆解，我首先对串行程序进行了改写，为后续的并行编程做准备。改写后的串行程序为了节省空间，仅使用了一个一维数组对结果进行保存。下面介绍改写后串行程序的基本函数：

(1) addright & addleft：这两个函数都可以执行一维数组中相邻元素相加的操作，其结果仍然放至该一维数组中，以节省内存分配和内存访问。通过交替调用 addleft 和 addright 函数可以避免潜在的内存移动和复制操作。

(2) iterationloop：该函数在指定次数的迭代中执行循环，交替调用 addleft 和 addright 函数，如果循环次数为奇数则调用 addleft，如果循环次数为偶数则调用 addright。

(3) computeloop：分配内存，对一维数组进行初始化，调用 iterationloop 函数，并将最后一维数组的结果进行返回。

(4) main：调用 computeloop 函数，获得最终结果，最后释放内存。

上述串行计算杨辉三角的程序的执行步骤，其实就是把二维数组压缩成一维数组，然后直接利用上一行的计算结果计算这一行的数据，结果保存在同一个一维数组中，利用的是杨辉三角内部元素当前值 = 上一行前一列的值 + 上一行同一列的值这个特性。假设打印杨辉三角的层数为 6，则程序一维数组的大小将会为 7，该串行程序的计算杨辉三角的过程如图4所示。

接下来在上述串行程序的基础上使用 MPI 进行并行编程。大致的思路就是将整行一维数组平均分配给每个处理器，不同的处理器处理不同的分区，处理器之间相互传递消息，以保证数据计算的正确性。如算法6所示。

下面对算法6中提及的一些变量和函数及其作用进行说明：

(1) idproc，nproc：nproc 描述当前有多少处理器正在运行此作业，idproc 标识处理器的名称 (0 nproc-1)。这两个变量由 main 提供给 computeloop 函数，由 computeloop 函数提供给 iterationloop 函数。

(2) leftIDProc & rightIDProc：该变量描述了所指定的左右相邻的处理

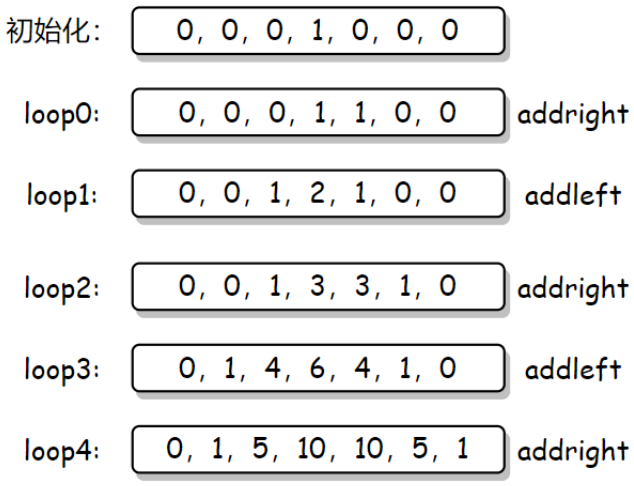

图 4　一维数组的杨辉三角计算过程

器 ID，用于迭代循环内的通信，在迭代循环进行之前进行计算。

(3) MPI_Irecv，MPI_Send，and MPI_Wait：这些是基本的 MPI 接口，将一个处理器的结果数据与另一个处理器的初始数据连接起来。leftIDProc 和 RightIDProc 标识消息发送和接收的处理器 ID。

(4) iterationloop 函数被赋予一个数组，该数组比该处理器实际负责的空间要大一个元素，额外的元素用作“保护”元素。保护元素位于分区的右边缘。addleft 之后，将该处理器最左边的元素通过 MPI_Send 提供给左边的处理器，该处理器最右边的元素通过 MPI_Recv 从右边的处理器获得。addright 之后，使用 MPI_Send 将此处理器最右侧的元素发送给右侧相邻的处理器，此处理器最左侧的元素通过使用 MPI_Recv 接收来自左侧相邻处理器的消息获得。

3.3　复杂度分析

本实验将杨辉三角的计算从二维数据压缩到了一维数组，因此空间复杂度为 $O(n)$。

尽管并行程序的执行使用多处理器，但由于 MPI 的处理器通信机制也会带来一定的开销，处理器的数目以及数组大小的相对性也会对程序的执行时间有影响。在这里只能带有前提假设的给出 MPI 并行环境下时间复杂度的估计值，假设处理器的个数为 K，且任务分配均匀，则时间复杂度为 $O(\frac{n^2}{K}) \lesssim O(n^2)$。

3.4 实验结果和分析

MPI 环境下的杨辉三角的计算过程如图5所示。图中每一行代表一次迭代过程，红色的箭头表示消息传递方向，这张图也表明了 addright 和 addleft 的交替执行过程。

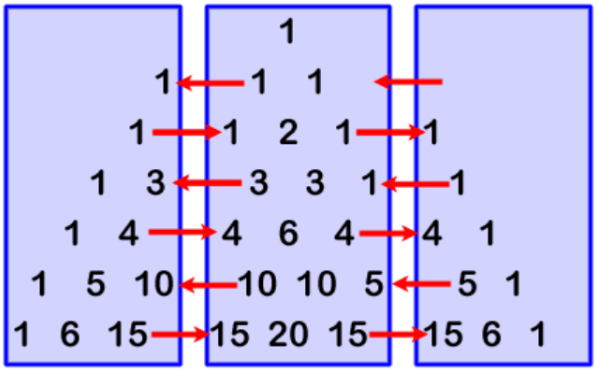

图 5　MPI 环境下杨辉三角计算过程

Algorithm 6 MPI 环境下的杨辉三角输出

Input: 需要输出的杨辉三角的层数 n

Output: 计算并打印层数为 n 的杨辉三角

1: main：MPI 初始化将当前处理器 id 以及处理器个数保存至 idproc 和 nproc 变量中。仅 0 号处理器读取用户的输入 n，并利用 MPI_Bcast 函数将该变量的值广播给所有处理器。调用 computeloop 函数，n,nproc 以及 idproc 作为参数。

2: computeloop：根据 n 和 nproc 的大小对每个处理器需要处理的任务（数据量）进行划分。如果是 0 号处理器，则开辟一个大数组的空间用于整合所有分任务的计算结果；对于其他的处理器，根据划分的任务数据量大小开辟空间。对数组进行初始化操作，即将数组中间某个元素赋值为 1，其余全部赋值为 0。调用 iterationloop 函数，将任务数组、数组的大小、输出层数 n、idproc 以及 nproc 作为参数。

3: iterationloop：首先指定 leftIDProc 和 RightIDProc 表示用于通信的处理器 ID 编号。然后对任务数组进行迭代循环，循环的次数即为输出的层数。

4: 如果循环次数为奇数则调用 addleft 函数，然后从 rightIDProc 接收任务数组最右边的元素，将本任务数组左边的元素发送给 leftIDProc，使用 MPI_Wait 函数确保数据进行了传输。

5: 对于循环次数为偶数的情况则调用 addright，然后从 leftIDProc 接收任务数组最左边的元素，发送任务数组最右边的元素给 rightIDProc，使用 MPI_Wait 函数确保数据进行了传输。

6: 由于需要打印每一次迭代的结果数据，需要在 iterationloop 中利用 MPI_Gather 函数，将所有处理器的任务数组数据整合到 0 号处理器的大数组中。

三　总结

通过并行编程原理与实践课程，我学习到了许多之前完全没有接触到的新鲜事物，对于并行处理的底层实现机制也有了自己的理解，通过实践课程我也掌握了 pthread、OpenMP 以及 MPI 环境下并行编程的具体实现方法。

MPI 实质上是多进程，并且不支持多线程；初始化时在每个处理机上创建一个进程，执行期间进程数目不能动态改变；MPI 其实就是一种在各个进程之间进行通讯的标准。

多核下的多线程程序设计需要将应用程序看做是众多相互依赖的任务集合，将应用程序划分为多个独立的任务，并确定这些任务之间的相互依赖关系，这就称为分解。包括任务的分解、数据的分解、数据流的分解：

(1) 任务分解：对应用程序根据其执行的功能进行分解的过程；

(2) 数据分解：数据级并行，将应用程序根据各任务所处理的数据而非按任务来进行分解；

(3) 数据流分解：基于数据在这些任务之间是如何流动的进行分解。

多线程程序设计中，线程的数目不是越多越好，对于某个程序，如果线程过多反而会严重地降低程序的性能。这种影响主要在于：将给定的工作量划分给过多的线程会造成每个线程的工作量过少，导致线程启动和终止的开销比程序实际工作的时间还要多。同时太多并发线程的存在将导致共享有限硬件资源的开销增大。如保存和恢复寄存器状态的开销、保存和恢复线程 cache 状态的开销、废除虚存的开销。

因此在进程并行程序的设计时，我们需要考虑很多问题，以便获得更好的程序执行效果。想要熟练的掌握并行编程的方法，还需要更多的学习和实践。

最后我想给本课程提一个小建议：我认为实验课程可能与理论课程可以同步进行，这样大家的学习效果和实践效果会更好。

在这里教你怎么引用:[1]

参考文献

[1] Durgam S, Earley W, Lipschitz A, et al. An 8-week randomized, double-blind, placebo-controlled evaluation of the safety and efficacy of cariprazine in patients with bipolar I depression[J]. American Journal of Psychiatry, 2016, 173(3): 271-281.

华中科技大学本科生毕业设计（论文）开题报告评审表

<table>
<tr><td>姓名</td><td></td><td>学号</td><td></td><td>指导教师</td><td></td></tr>
<tr><td colspan="2">院（系）专业</td><td colspan="4"></td></tr>
<tr><td colspan="6">指导教师评语
1. 学生前期表现情况。
2. 是否具备开始设计（论文）条件？是否同意开始设计（论文）?
3. 不足与建议。</td></tr>
<tr><td colspan="6"></td></tr>
<tr><td colspan="6">指导教师（签名）:

年　　月　　日</td></tr>
<tr><td colspan="6">教研室（系、所）或开题报告答辩小组审核意见</td></tr>
<tr><td colspan="6"></td></tr>
<tr><td colspan="6">教研室（系、所）或开题报告答辩小组负责人（签名）:

年　　月　　日</td></tr>
</table>