目 录

1 实验一 串行环境下斐波那契数列计算

1.1 实验目的与要求

- 熟悉 linux 环境编程
- 在串行环境下编写计算斐波那契数列的程序
- 掌握基本编程方法与 linux 知识

1.2 实验内容

1. 要求

编写程序，计算斐波那契数列，并且将结果按照要求打印出来。特别地，需要注意空格的存在。

例如：

测试输入：

```
1
```

预期输出：

```
1
```

测试输入：

```
4
```

预期输出：

```
1 1 2 3
```

2. 实验设计

根据斐波那契额的数学公式：

$$f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & n > 1 \end{cases}$$

可以很容易地想到利用迭代的方法计算斐波那契数列。

主要代码如下：

```
for (i = 1; i <= n; i++)
{
    if(i == 1){
        printf("%lld", temp1);
    }
    else{
        printf(" %lld", temp1);
    }
    next_fibo = temp1 + temp2;
    temp1 = temp2;
    temp2 = next_fibo;
}
```

可以看出每次计算出一个对应位上的斐波那契数，然后将其打印出来即可。此种方法的时间复杂度可以说是比较低的了，为 O(n).

1.3 实验结果

将所写代码放入平台进行测试，可以看到 5 个测试集全部通过。可以说明代码是正确的。

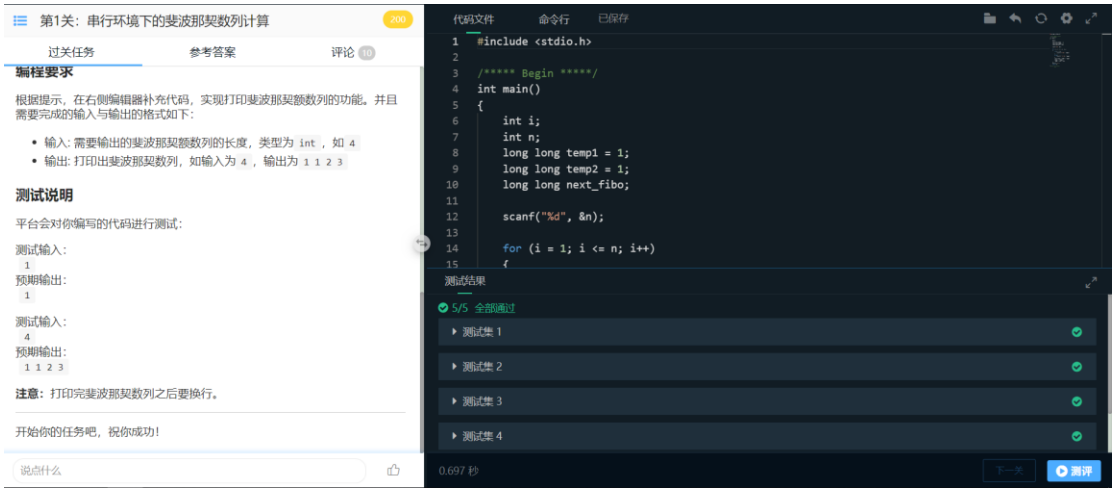

图 1-1 串行环境斐波那契数列测试

由于平台测试时间具有不确定性，并且与平台在线人数有关系，不能正确反映程序真正的执行时间，因此在本实验报告里只证明程序的正确性，加速比分析之类的在思考题报告中反映。

2 实验二 pthread 斐波那契数列计算

2.1 实验目的与要求

- 掌握进程与线程的概念
- 掌握基于 pthread 的并行编程设计方法
- 掌握数据分块与任务分解
- 调试并行程序并分析结果

2.2 算法描述

使用 pthread 来使程序并行主要就是在一个进程内创建多个线程来完成分块后的任务，最后将结果汇总。

使用 pthread 需要使用几个基本函数，如图 2-1 pthread 并行流程图：

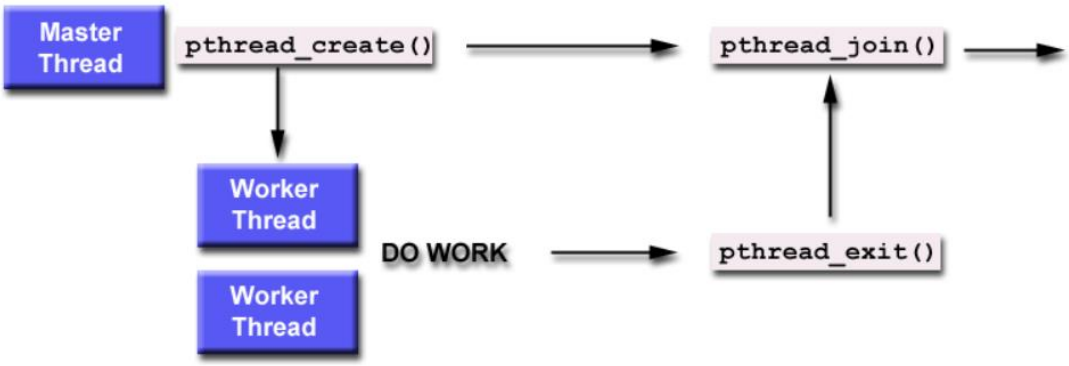

图 2-1 pthread 并行流程图

如上图所示，在主线程中使用 pthread_create（）创建子线程，然后让这些线程计算各自需要计算的斐波那契数列，将结果保存在主线程中的数组中，等所有子线程结束后，在主线程中按照要求顺序输出数组的值即可。

显然在理论上，使用 pthread 能显著加快程序的运行速度，但在实际中还需要考虑线程创建的开销以及数据集的大小等问题。

同样地，在并行中，要尽可能使线程之间的工作没有关联，因此在串行环境中那种迭代的方法已经不是很适用了，因此使用通项公式计算斐波那契数列每一项的值：

$$a_n = \frac{1}{\sqrt{5}}[(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n]$$

这样只需要将每个 n 带进去，计算结果存入数组对应位置即可。需要记住的是不能在子线程里面进行输出，否则顺序会乱。

同时为了使线程计算正确位置的斐波那契数，需要划定起始位置与结束位置，将这个作为线程处理函数的参数。为了想 pthread 处理函数传参，需要将参数打包，因此使用结构体包含这两个参数。具体思路是先确定每个线程需要处理的个数，然后根据第几个线程确定起始结束位置。部分主要代码如下：

```
1.  perGroupNum = ceil((num - 1)/4.0);
2.  threadNum = 4 > perGroupNum ? perGroupNum : 4;
3.  if(threadNum < 4){
4.      perGroupNum = ceil((num - 1.0)/threadNum);
5.  }
6.  //printf("%d %d\n", perGroupNum, threadNum);
7.
8.  //get thread's parameter
9.  for(i=0; i<threadNum; ++i){
10.     farg[i].startpos = i * perGroupNum + 2;
11.     farg[i].endpos = ((i+1) * perGroupNum + 2) >= num ? num+1 : ((i+1) * per
    GroupNum + 2);
12.     //printf("%d %d\n", farg[i].startpos, farg[i].endpos);
13. }
```

这样使用 pthread_join 等到所有子线程结束后，才打印斐波那契数列。

这样做有可能会影响并行程序的效率。

第一点就是需要计算起始位置与结束位置，这需要一定的开销，不可避免。

第二点就是需要在最后才能打印斐波那契的值，极大限制了并行程序的效率，这点只要评测方式不改变，就避免不了串行。

2.3 实验方案

2.3.1 开发与运行环境

本实验在测试平台上测试，也在我的本机上通过了测试。这里给出本机的环境。

系统版本：ubuntu1804.2

编译环境：gcc 7.4.0

CPU：Intel® Core™ i5-7200U CPU @ 2.50GHz 2.70GHz

RAM: 8GB

2.3.2 实验过程

根据算法描述编写程序，先通过本机测试，观察输出结果，如果大致正确就放到平台测试。注意观察时间。

如果平台测试不通过，需要考虑输出是否符合要求。

2.4 实验结果与分析

最后在平台上通过了所有的测试集。结果如图：

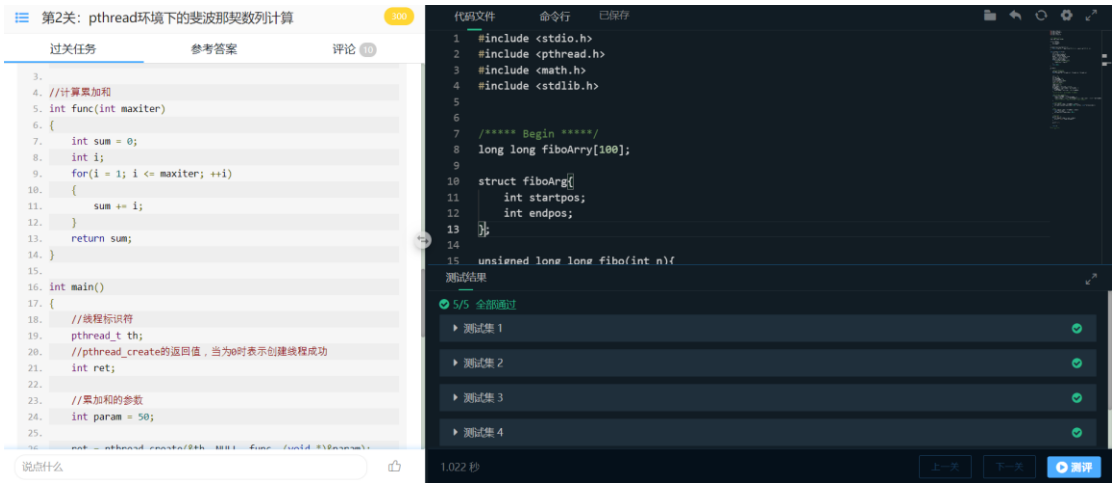

图 2-2 pthread 测试结果图

可以看到代码是正确的。

同时，与实验一类似，由于平台测试时间具有不确定性，并且与平台在线人数有关系，不能正确反映程序真正的执行时间，因此在本实验报告里只证明程序的正确性，加速比分析之类的在思考题报告中反映。

3 实验三 OpenMP 斐波那契数列计算

3.1 实验目的与要求

- 掌握 OpenMP 的概念
- 掌握基于 OpenMP 的并行编程设计方法
- 掌握在 OpenMP 环境下斐波那契数列的计算方法
- 掌握使用 OpenMP 的条件，了解什么情况下才能对 for 循环并行化
- 调试并行程序并分析结果

3.2 算法描述

OpenMP 是多线程的一种实现方式，它采用 fork-join 模式，主线程分叉指定一定数量的子线程，并为它们划分任务。

这种方式特别适合在多核 CPU 机器上运行，编译器会根据程序中添加的 pragma 指令，自动并行，大大降低了并行编程的复杂度，同时在现有代码上进行并行时，也不需要大规模更改代码，因此是一种十分方便的方法。

在使用 OpenMP 时，只需要在 for 循环之前插入 pragma 就可以实现并行化。

将#pragma omp parallel for 插入 for 循环之前即可，也可以显示指定要创建的线程数 num_threads（），除此之外还可以明确控制 for 循环的调度算法（静态，动态，指导， 自动和运行时），使用 schedule（）指定。

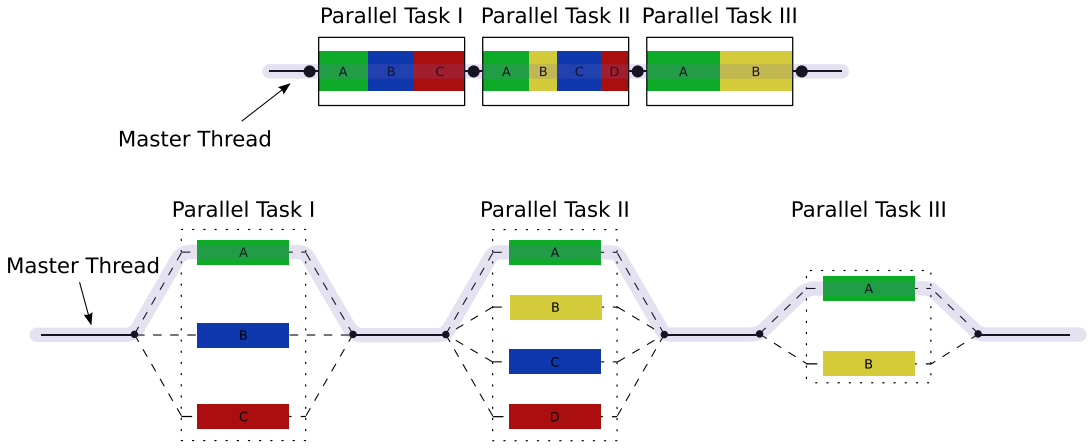

图 3-1 fork-join 模式

同时了解到使用 OpenMP 还有诸多限制条件，比如对 for 循环进行优化就需要以下要求：

- 循环的变量变量(就是 i)必须是有符号整形，其他的都不行。
- 循环的比较条件必须是< <= > >=中的一种
- 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）。
- 如果比较符号是< <=，那每次循环 i 应该增加，反之应该减小
- 循环必须是没有奇奇怪怪的东西，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。

3.3 实验方案

3.3.1 开发与运行环境

本实验在测试平台上测试，也在我的本机上通过了测试。这里给出本机的环境。

系统版本：ubuntu1804.2

编译环境：gcc 7.4.0

CPU：Intel® Core™ i5-7200U CPU @ 2.50GHz 2.70GHz

RAM: 8GB

3.3.2 实验过程

根据算法描述编写程序，由于本次与使用 pthread 时代码变化不大，因此直接在原有代码上进行修改即可。gcc 编译时也不需要加上-pthread。

测试时先通过本机测试，观察输出结果，如果大致正确就放到平台测试。注意观察时间。

如果平台测试不通过，需要考虑输出是否符合要求。

3.4 实验结果与分析

最后在平台上通过了所有的测试集。结果如图：

图 3-2 OpenMP 测试结果图

可以看到代码是正确的。

同时，与实验一类似，由于平台测试时间具有不确定性，并且与平台在线人数有关系，不能正确反映程序真正的执行时间，因此在本实验报告里只证明程序的正确性，加速比分析之类的在思考题报告中反映。

同样地，由于最后输出存在串行，所以时间上的比较也不准确。

4 实验四 MPI 斐波那契数列计算

4.1 实验目的与要求

- 掌握 MPI 的概念
- 掌握基于 MPI 的并行编程设计方法
- 掌握在 MPI 环境下斐波那契数列的计算方法
- 理解 MPI 使程序并行化的原理
- 调试并行程序并分析结果

4.2 算法描述

就如任务指导所说，MPI 是一个跨语言的通讯协议，用于编写并行计算机。支持点对点和广播。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。

这里就要具体深入了解 MPI 的消息传递机制了。在本次实验中我们只需要了解 MPI 的基本函数即可。

根据任务指导上的说明，MPI 基本函数的定义如下：

1. MPI_Init

```
int MPI_Init (int* argv ,char** argc[])
```

该函数通常应该是第一个被调用的 MPI 函数用于并行环境初始化，其后面的代码到 MPI_Finalize()函数之前的代码在 mpirun 跑的每个进程中都会被执行一次。

2. MPI_Finalize

```
int MPI_Finalize (void)
```

该函数用于退出 MPI 系统，所有进程正常退出都必须调用。表明并行代码的结束,结束除主进程外其它进程。串行代码仍可在主进程(rank 为 0 的进程)上运行，但不能再有 MPI 函数。

3. MPI_Comm_size

```
int MPI_Comm_size (MPI_Comm comm ,int* size)
```

该函数用于获得并行的进程个数。并将个数返回 size。其中：

- comm：指定一个通信子,也指定了一组共享该空间的进程，这些进程组成该通信子的 group。在调用该函数时，一般将 MPI_COMM_WORLD 作为参数传给 comm。
- size：获得通信子 comm 中规定的 group 包含的进程的数量。

4. MPI_Comm_rank

```
int MPI_Comm_rank (MPI_Comm comm ,int* rank)
```

该函数用于得到本进程在通信空间中的 rank 值，并将该值返回给 rank。rank 值即在 group 中的逻辑编号(该 rank 值为 0 到 p-1 间的整数，相当于进程的 ID。其中：

- comm：指定一个通信子,也指定了一组共享该空间的进程，这些进程组成该通信子的 group。在调用该函数时，一般将 MPI_COMM_WORLD 作为参数传给 comm。
- rank：获得本进程在通信子 comm 中规定的 group 中的逻辑编号。

5. MPI_Send

```
int MPI_Send( void *buff, int count, MPI_Datatype datatype, int dest, int
tag, MPI_Comm comm)
```

该函数用于对指定的进程以阻塞的方式发送数据。其中：

- buff：你要发送的数据。
- count：你发送的消息的个数(注意：不是长度，例如你要发送一个 int 整数，这里就填写 1，如要是发送“hello”字符串，这里就填写 6(C 语言中字符串未有一个结束符，需要多一位))。
- datatype：你要发送的数据类型，这里需要用 MPI 定义的数据类型，如 MPI_INT。
- int dest：目的地进程号，你要发送给哪个进程，就填写目的进程的进程号，也就是 rank 值。
- tag：消息标签，接收方需要有相同的消息标签才能接收该消息。
- comm：通讯域。表示你要向哪个组发送消息。

6. MPI_Recv

```
int MPI_Recv( void *buff, int count, MPI_Datatype datatype, int source, int
tag, MPI_Comm comm, MPI_Status *status)
```

该函数用于对指定的进程以阻塞的方式接收数据。其中：

- buff：你接收到的消息要保存到哪个变量里。
- count：你接收的消息的个数(注意：不是长度，例如你要接收一个 int 整数，这里就填写 1，如要是接收“hello”字符串，这里就填写 6(C 语言中字符串未有一个结束符，需要多一位))。它是接收数据长度的上界. 具体接收到的数据长度可通过调用 MPI_Get_count 函数得到。
- datatype：你要接收的数据类型，这里需要用 MPI 定义的数据类型，如 MPI_INT。

- `source`：接收端进程号，你要需要哪个进程接收消息就填写接收进程的进程号，也就是 `rank` 值。
- `tag`：消息标签，需要与发送方的 `tag` 值相同的消息标签才能接收该消息。
- `comm`：通讯域。表示你要接收哪个组发送过来的消息。
- `status`：消息状态。接收函数返回时，将在这个参数指示的变量中存放实际接收消息的状态信息，包括消息的源进程标识，消息标签，包含的数据项个数等。

以上就是基本的函数介绍，下面给出使用方法，主要介绍不同线程的处理方法。

首先要注意的是，不能在 0 线程外进行输入，否则程序将陷入死循环。之前做实验的时候一直使程序陷入死循环，经过晚上查资料也没有解决，后来定位错误确定是由于在 0 进程外进行了输入操作导致了这个 bug。因此尝试在 0 进程内进行输入，后来问题解决。但是这样的话，会引入新的问题。

新的问题就是如何把输入的要计算的斐波那契数目传递给其他进程。这里一开始我也没有想到很好地办法，后来想着 MPI 本身的通信工具就是 MPI_Send 和 MPI_Recv.

所以一开始 0 好进程要进行的工作就是输入要计算的斐波那契数量，然后使用 MPI_Send 将这个变量发送到其他进程。

```
1.  MPI_Send(&num, 1, MPI_INT, 1, 38, MPI_COMM_WORLD);
2.  MPI_Send(&num, 1, MPI_INT, 2, 38, MPI_COMM_WORLD);
3.  MPI_Send(&num, 1, MPI_INT, 3, 38, MPI_COMM_WORLD);
```

接收到了 num 参数后，我们转向其他进程研究。

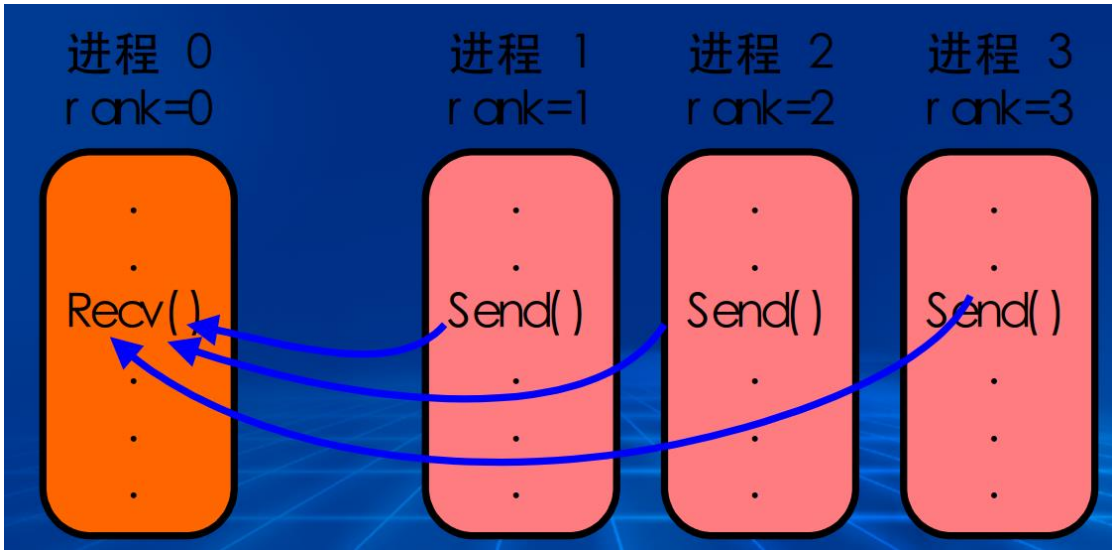

图 4-1 MPI 信息传递机制

MPI 使用阻塞机制传递信息，首先我们使用 MPI_Recv 来接受 num。

```
1.  MPI_Recv(&num, 1, MPI_INT, 0, 38, MPI_COMM_WORLD, &status);
2.
3.  int startpos = num/(numprocs - 1) * (myid - 1);
4.  int endpos = (myid == (numprocs - 1))? num - 1 : startpos + num/(numprocs -
    1) - 1;
5.  //starts with 0
6.  int len = endpos - startpos + 1;
7.  for(int i = startpos+1; i <= endpos+1; ++i){
8.      fiboArry[i] = fibo(i);
9.      //printf("In [rank %d], compute fibo: %d %llu\n", myid, i, fiboArry[i]);

10.     //MPI_Send(&fiboArry[i], sizeof(long long), MPI_LONG_LONG, 0, 38, MPI_CO
    MM_WORLD);
11. }
12. MPI_Send(fiboArry+startpos+1, len, MPI_LONG_LONG, 0, 99, MPI_COMM_WORLD);
```

接受了之后我们就可以开始计算每个进程计算的起始位置和终止位置。计算

完成后，我们就需要利用 MPI_Send 来传递数据了，从 fiboArry+startpos+1 开始（该进程计算的第一个斐波那契数），然后 len 表示传递的长度。

之后我们就需要在 0 号进程处理接收到的信息了。

```
1.  for(int i = 1; i< numprocs; ++i){
2.      int received_count = 0;
3.      MPI_Probe(i, 99, MPI_COMM_WORLD, &status);
4.      // get real length before recv
5.      MPI_Get_count(&status, MPI_LONG_LONG, &received_count);
6.      //printf("in rank 0, received_count from %d: %d\n", i, received_count);


7.      // int temp = 0;
8.      // MPI_Recv(&temp, 1, MPI_INT, i, 99, MPI_COMM_WORLD, &status);
9.      MPI_Recv(pfibo, received_count, MPI_LONG_LONG, i, 99, MPI_COMM_WORLD, &s
    tatus);
10.     //printf("from  %d   %lld  ->  %lld\n", i, *(pfibo), *(pfibo + received_
    count - 2));
11.     //startpos += received_count;
12.     pfibo += received_count;
13. }
```

首先我们需要确定收到信息的长度，也就是非 0 号进程打过来的 len，这里我们就要用到 MPI_Probe 和 MPI_Get_count 了，具体用法可以 google，在这里只是介绍它们作什么用。然后我们就可以往主进程的数组里填数字了，开始位置为 pfibo，长度为 received_count，不要忘了当一个进程的信息全部接收完后，需要将 pfibo 增加 received_count，改变后面的起始位置。

4.3 实验方案

4.3.1 开发与运行环境

本实验在测试平台上测试，也在我的本机上通过了测试。这里给出本机的环境。

系统版本：ubuntu1804.2

编译环境：gcc 7.4.0　　MPICH 3.3a2

CPU：Intel® Core™ i5-7200U CPU @ 2.50GHz 2.70GHz

RAM: 8GB

4.3.2 实验过程

根据算法描述编写程序，先通过本机测试，观察输出结果，如果大致正确就放到平台测试。注意观察时间。

在本机测试需要安装 mpich，编译时需要使用 mpic++编译，运行时也需要使用 mpirun，如果在本机可以改变进程的数量，但是在测试平台就只能使用规定的 4 个进程。

如果平台测试不通过，需要考虑输出是否符合要求。

4.4 实验结果与分析

最后在平台上通过了所有的测试集。结果如图：

图 4-2 MPI 测试结果图

可以看到代码是正确的。

同时，与实验一类似，由于平台测试时间具有不确定性，并且与平台在线人数有关系，不能正确反映程序真正的执行时间，因此在本实验报告里只证明程序的正确性，加速比分析之类的在思考题报告中反映。

本次测试时间略长，猜测与平台有关，并且非 0 号进程一开始需要等待 0 号进程的输入，因此在这段时间它们被阻塞了，后来 0 号进程也必须等待其他进程的结果全部计算完传输过来后，才能进行输出，并且输出也是串行的，因此时间测量不准确。

5 实验五 CUDA 斐波那契数列计算

5.1 实验目的与要求

- 掌握 CUDA 的概念
- 掌握基于 CUDA 的并行编程设计方法
- 掌握在 CUDA 环境下斐波那契数列的计算方法
- 了解 CUDA 计算原理
- 调试并行程序并分析结果

5.2 算法描述

CUDA 是由 NVIDIA 公司创立的基于他们公司生产的图形处理器 GPUs(Graphics Processing Units)的一个并行计算平台和编程模型。通过 CUDA，GPUs 可以用来计算，用来进行并行编程，达到高性能计算的目的。

CUDA 中将 CPU 及内存称为主机，GPU 及显存称为设备。CUDA 操作基本单位为线程。多个线程可以组成线程块，线程块是并行的，不同线程块之间不能通信，也没有执行顺序。多个线程块组成网络。

在使用 CUDA 之前，需要了解 CUDA 函数。

想要分配设备内存的话就需要用到 `cudaMalloc` 函数。该函数的声明如下：

```
cudaError_t cudaMalloc (void **devPtr, size_t    size );
```

其中：

- `devPtr`：分配设备内存的指针的地址
- `size`：想要分配多大空间的设备内存

想要对设备内存中的数据进行拷贝的话，就需要用到 `cudaMemcpy` 函数。该函数的声明如下：

```
cudaError_t cudaMemcpy ( void* dst, const void* src, size_t count, cudaMemcpyKind
kind );
```

其中：

- dst：想要把数据拷贝到哪里
- src：想要拷贝的数据来自哪里
- count：拷贝多少内存
- kind：拷贝的动作类型，比如想要将设备内存拷贝到主机内存，就传入 cudaMemcpyDeviceToHost。

有分配内存的 API，当然也有释放内存的 API。cudaFree 函数可以释放使用 cudaMalloc 分配的内存。

对内存有了一定的理解后，我们就需要知道怎样去使用这些内存了。在 CUDA C 中想要区分主机代码和设备代码很简单，就是看有没有特殊的前缀，例如__global__。在函数前面加上__global__表示这个函数是要在 GPU 上执行的。例如如下计算数组内所有数据的平方和的函数就是可以在 GPU 上执行的：

因此，在之前计算斐波那契的函数前面需要加上__global__。

```
1.  __global__ void calFibo(long long *fiboArry)
2.  {
3.      int tid = threadIdx.x;
4.      fiboArry[tid] = (long long)((pow((1+sqrt(5.0)), tid) - pow((1-
    sqrt(5.0)), tid))/( pow((double)2, tid) * (sqrt(5.0)) ) + 0.5);
5.  }
```

由上面看出，在这里采用的方法是调用与需要计算的斐波那契数目相同的线程，每个线程进行一个数的计算。这样似乎是最快的方法，但是实际过程中如果数据集不大，那么线程的开销就会非常大，得不偿失。

值得注意的是，在斐波那契数列计算的最后加上了 0.5。这是因为在平台上测试代码的时候，发现有几个数不匹配，结果都是少了 1，而之前的几次都没有出现类似的问题，猜测是 CUDA 计算的问题，因此最后加上 0.5 尝试，结果正确。

之后在 main 函数理面就需要分配设备内存，调用计算函数，对设备中内存进行拷贝了。

核心代码如下：

```
1.  cudaMalloc((void**)&fiboCal, sizeof(long long)*(num+1));
2.  calFibo<<<1,num+1>>>(fiboCal);
3.  cudaMemcpy(&fiboResult,fiboCal,sizeof(long long)*(num+1),cudaMemcpyDeviceToH
    ost);
```

第一行是分配设备内存，分配设备内存的指针的地址就是 long long*类型的指针变量，大小为 sizeof(long long)*(num+1)，因为 0 号斐波那契数不可避免地进行了计算。

第二行就是调用 calFibo 函数了，1 代表使用 1 个线程块，num+1 代表使用 num+1 个线程。

第三行就是将每个线程计算的结果合起来送入 fiboResult 中，数据开始于 fiboCal，大小为 sizeof(long long)*(num+1)，cudaMemcpyDeviceToHost 表示数据从设备内存拷贝到主机内存。

5.3 实验方案

5.3.1 开发与运行环境

本实验在测试平台上测试，因为本机上没有 NVIDIA 显卡，因此无法配置 CUDA 环境，只能在平台上测试。

5.3.2实验过程

根据算法描述编写程序，直接到平台上进行测试。

如果平台测试不通过，需要考虑输出是否符合要求。

5.4 实验结果与分析

最后在平台上通过了所有的测试集。结果如图：

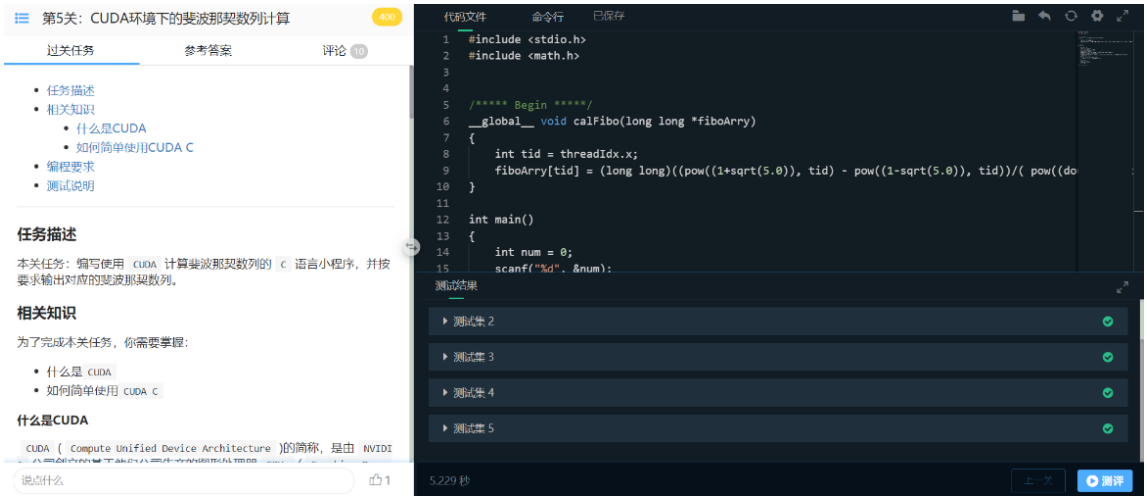

图 5-1 pthread 测试结果图

可以看到代码是正确的。

最后出来的结果也是很不理想，平台所用的服务器可能提供不了很好的计算性能，因此时间特别长，也是不能很好地衡量时间。

但是实际中 GPU 的计算性能是十分好的，因此用于计算的地方非常多。比如矢量计算等，大型游戏需要这种计算。只是平台的限制不能很好地体现这个特性。

附录

1. 串行

```
#include <stdio.h>

/***** Begin *****/
int main()
{
    int i;
    int n;
    long long temp1 = 1;
    long long temp2 = 1;
    long long next_fibo;

    scanf("%d", &n);

    for (i = 1; i <= n; i++)
    {
        if(i == 1){
            printf("%lld", temp1);
        }
        else{
            printf(" %lld", temp1);
        }
        next_fibo = temp1 + temp2;
        temp1 = temp2;
        temp2 = next_fibo;
    }
    printf("\n");
```

```
27.     return 0;
28. }
29. /***** End *****/
```

2. pthread

```
1.  #include <stdio.h>
2.  #include <pthread.h>
3.  #include <math.h>
4.  #include <stdlib.h>
5.
6.
7.  /***** Begin *****/
8.  long long fiboArry[100];
9.
10. struct fiboArg{
11.     int startpos;
12.     int endpos;
13. };
14.
15. unsigned long long fibo(int n){
16.     return 1/sqrt(5) * (pow((1+sqrt(5))/2, n) - pow((1-sqrt(5))/2, n));
17. }
18.
19. void *calFibo(void *arg){
20.     int i = 0;
21.     int startpos=0,endpos=0;
22.     struct fiboArg *myarg;
23.     myarg = (struct fiboArg *)arg;
24.     startpos = myarg->startpos;
```

```
25.     endpos = myarg->endpos;
26.     for(i=startpos;i<endpos;++i){
27.         //printf(" %llu", fibo(i));
28.         fiboArry[i] = fibo(i);
29.     }
30.     return NULL;
31. }
32.
33. int main()
34. {
35.     //define thread pid
36.     pthread_t threads[4];
37.     void *thread1_ret, *thread2_ret, *thread3_ret, *thread4_ret;
38.
39.
40.     int i = 0;
41.     int num = 0;
42.     int threadNum = 0;
43.     int perGroupNum = 0;
44.     struct fiboArg farg[50];
45.     scanf("%d", &num);
46.     //printf("%d\n", 1);
47.     fiboArry[0] = 0;
48.     fiboArry[1] = 1;
49.     perGroupNum = ceil((num - 1)/4.0);
50.     threadNum = 4 > perGroupNum ? perGroupNum : 4;
51.     if(threadNum < 4){
52.         perGroupNum = ceil((num - 1.0)/threadNum);
53.     }
54.     //printf("%d %d\n", perGroupNum, threadNum);
```

```
55.
56.     //get thread's parameter
57.     for(i=0; i<threadNum; ++i){
58.         farg[i].startpos = i * perGroupNum + 2;
59.         farg[i].endpos = ((i+1) * perGroupNum + 2) >= num ? num+1 : ((i+1) * p
    erGroupNum + 2);
60.         //printf("%d %d\n", farg[i].startpos, farg[i].endpos);
61.     }
62.
63.     //create threads
64.     for(int index=0; index < threadNum; ++index){
65.         pthread_create(&threads[index], NULL, calFibo, (void *)&(farg[index]))
    ;
66.     }
67.
68.     //wait
69.     for(int index=0; index < threadNum; ++index){
70.         void **returnVal;
71.         pthread_join(threads[index], NULL);
72.     }
73.
74.     //show fibo
75.     printf("%d", 1);
76.     for(int index=2; index <= num; ++index){
77.         printf(" %lld", fiboArry[index]);
78.     }
79.
80.     printf("\n");
81.
82.
```

```
83.     return 0;
84. }
85. /***** End *****/
```

3. OpenMP

```
1.  #include <stdlib.h>
2.  #include <stdio.h>
3.  #include <math.h>
4.  #include <string.h>
5.
6.  /***** Begin *****/
7.  long long fiboArry[100];
8.
9.  unsigned long long fibo(int n){
10.     return 1/sqrt(5) * (pow((1+sqrt(5))/2, n) - pow((1-sqrt(5))/2, n));
11. }
12.
13. int main()
14. {
15.     int i;
16.     int n;
17.     // long long temp1 = 1;
18.     // long long temp2 = 1;
19.     // long long next_fibo;
20.
21.     scanf("%d", &n);
22.     fiboArry[0] = 0;
23.     fiboArry[1] = 1;
24.
```

```
25.     #pragma omp parallel for
26.     for(i = 2; i <= n; ++i){
27.         fiboArry[i] = fibo(i);
28.     }
29.
30.     // printf("%d", 1);
31.     for(i = 1;i<=n;++i){
32.         printf("%llu ", fiboArry[i]);
33.     }
34.     printf("\n");
35.     return 0;
36. }
37. /***** End *****/
```

4. MPI

```
1.  #include <stdlib.h>
2.  #include <stdio.h>
3.  #include <mpi.h>
4.  #include <math.h>
5.  #include <string.h>
6.
7.  /***** Begin *****/
8.  unsigned long long fibo(int n){
9.      return 1/sqrt(5) * (pow((1+sqrt(5))/2, n) - pow((1-sqrt(5))/2, n));
10. }
11.
12. int main(int argc, char* argv[])
13. {
14.     int numprocs, myid, source;
```

```
15.    MPI_Status status;
16.    MPI_Init(&argc, &argv);
17.    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
18.    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
19.    if (myid != 0) {  //非 0 号进程发送消息
20.        long long fiboArry[100];
21.        //long long test[100];
22.        int num = 0;
23.        MPI_Recv(&num, 1, MPI_INT, 0, 38, MPI_COMM_WORLD, &status);
24.
25.        int startpos = num/(numprocs - 1) * (myid - 1);
26.        int endpos = (myid == (numprocs - 1))? num - 1 : startpos + num/(numpr
    ocs - 1) - 1;
27.        //starts with 0
28.        int len = endpos - startpos + 1;
29.        for(int i = startpos+1; i <= endpos+1; ++i){
30.            fiboArry[i] = fibo(i);
31.            //printf("In [rank %d], compute fibo: %d %llu\n", myid, i, fiboArry[i]);
32.            //MPI_Send(&fiboArry[i], sizeof(long long), MPI_LONG_LONG, 0, 3
    8, MPI_COMM_WORLD);
33.        }
34.        MPI_Send(fiboArry+startpos+1, len, MPI_LONG_LONG, 0, 99, MPI_C
    OMM_WORLD);
35.        //printf("In [rank %d], len : %d   array: %lld\n", myid, len, *(fiboArry+st
    artpos+1));
36.
37.        // int temp = 0;
38.        //temp = fibo(3);
```

```
39.         // temp = 3;
40.         // MPI_Send(&temp, 1, MPI_INT, 0, 99, MPI_COMM_WORLD);
41.     }
42.     else {   // myid == 0，即 0 号进程接收消息
43.         long long fiboArry[100];
44.         //int testmain[100];
45.         // long long fiboArryTemp[100];
46.         long long *pfibo = fiboArry;
47.         int num = 0;
48.         scanf("%d", &num);
49.         MPI_Send(&num, 1, MPI_INT, 1, 38, MPI_COMM_WORLD);
50.         MPI_Send(&num, 1, MPI_INT, 2, 38, MPI_COMM_WORLD);
51.         MPI_Send(&num, 1, MPI_INT, 3, 38, MPI_COMM_WORLD);
52.         for(int i = 1; i< numprocs; ++i){
53.             int received_count = 0;
54.             MPI_Probe(i, 99, MPI_COMM_WORLD, &status);
55.             // get real length before recv
56.             MPI_Get_count(&status, MPI_LONG_LONG, &received_count);
57.             //printf("in rank 0, received_count from %d: %d\n", i, received_count);

58.             // int temp = 0;
59.             // MPI_Recv(&temp, 1, MPI_INT, i, 99, MPI_COMM_WORLD, &sta
    tus);
60.             MPI_Recv(pfibo, received_count, MPI_LONG_LONG, i, 99, MPI_C
    OMM_WORLD, &status);
61.             //printf("from  %d   %lld  ->  %lld\n", i, *(pfibo), *(pfibo + received_c
    ount - 2));
62.             //startpos += received_count;
63.             pfibo += received_count;
64.         }
```

```
65.
66.         printf("%d", 1);
67.         for(int i = 1; i<=num - 1; ++i){
68.             printf(" %lld", fiboArry[i]);
69.         }
70.         printf("\n");
71.     }
72.     MPI_Finalize();
73.
74.     return 0;
75. }
76. /***** End *****/
```

5. CUDA

```
1.  #include <stdio.h>
2.  #include <math.h>
3.
4.
5.  /***** Begin *****/
6.  __global__ void calFibo(long long *fiboArry)
7.  {
8.      int tid = threadIdx.x;
9.      fiboArry[tid] = (long long)((pow((1+sqrt(5.0)), tid) - pow((1-
    sqrt(5.0)), tid))/( pow((double)2, tid) * (sqrt(5.0)) ) + 0.5);
10. }
11.
12. int main()
13. {
14.     int num = 0;
```

```
15.     scanf("%d", &num);
16.     long long fiboResult[100];
17.     long long *fiboCal;
18.     cudaMalloc((void**)&fiboCal, sizeof(long long)*(num+1));
19.     calFibo<<<1,num+1>>>(fiboCal);
20.     cudaMemcpy(&fiboResult,fiboCal,sizeof(long long)*(num+1),cudaMemcp
    yDeviceToHost);
21.     printf("%d", 1);
22.     for(int i = 2; i<= num; ++i){
23.         printf(" %lld", fiboResult[i]);
24.     }
25.     printf("\n");
26.     cudaFree(fiboCal);
27.     return 0;
28. }
29. /***** End *****/
```