

**并行原理实践报告**

|  |  |
| --- | --- |
|  |  |
| 姓 名： | 李超峰 |
| 学 院： | 计算机科学与技术学院 |
| 专 业： | CS1605 |
| 学 号： | U201614633 |
| 指导教师： | 陆枫 |

|  |  |
| --- | --- |
| 分数 |  |
| 教师签名 |  |

2019 年 7 月28 日

目录

[实验任务一 3](#_Toc15337371)

[1.1任务要求 3](#_Toc15337372)

[1.2实验内容 3](#_Toc15337373)

[1.3实验结果 9](#_Toc15337374)

# 实验任务一

## 1.1任务要求

1. 在串行环境下编写计算斐波那契数列的C语言小程序，并按要求输出对应的斐波那契数列。

2. 编写使用多线程计算斐波那契数列的C语言小程序，并按要求输出对应的斐波那契数列。

3．编写使用OpenMP计算斐波那契数列的C语言小程序，并按要求输出对应的斐波那契数列。

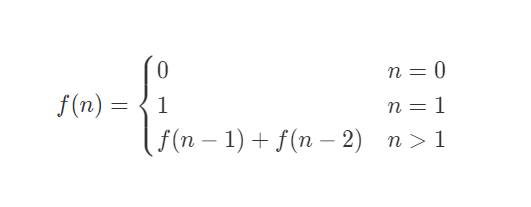
4. 编写使用MPI计算斐波那契数列的C语言小程序，并按要求输出对应的斐波那契数列。

5. 编写使用CUDA计算斐波那契数列的C语言小程序，并按要求输出对应的斐波那契数列。

## 1.2实验内容

1.2.1串行实现斐波那契数列

根据斐波那契数列的数学公式实现递归调用相关的函数实现，此方式时间复杂度较高。数学公式如下



1.2.2用多线程实现斐波那契数列

多线程相关知识如下

进程是程序的一个动态运行实例。而线程则是在进程的内部运行(进程的地址空间)运行的一个分支，也是调度的基本单位。进程中最少有一个线程，即主线程。其实每次运行上一关的程序，就相当于创建了一个进程，并由于程序中没有创建多个线程，所以该进程中只有一个主线程。

pthread的使用

在Linux系统下，与线程相关的函数都定义在pthread.h头文件中。所以首先需要包含这个头文件，代码如下：

#include <pthread.h>

如果想要创建一个线程，就需要用到pthread\_create函数。pthread\_create函数的函数声明如下：

int pthread\_create(pthread\_t \* thread, const pthread\_arrt\_t\* attr,void\*(\*start\_routine)(void \*), void\* arg)

若线程创建成功，则返回值为0，否则会返回错误码。该函数参数意义如下：

thread: 新线程的标识符，为一个整型数字。

attr: 用来设置线程属性，一般传NULL，即使用线程的默认设置。

start\_routine: 线程函数的地址，即想要该线程执行的函数，当该函数返回后，线程就退出了。

arg: 线程函数所需要传入的参数。

创建完线程后，线程就可能会被CPU调度，并执行线程函数。不过我们的主线程并不会自动的去等创建好的线程运行结束后再退出。若不加限制，就很可能出现子线程还没运行完毕，主线程就退出了。那么该如何是好？我们可以使用pthread\_join函数。pthread\_join函数的函数声明如下：

void pthread\_join(pthread\_t thread,void \*\* retval)

调用该函数的线程会等待变成挂起状态，只到thread所代表的线程运行结束。其中函数参数的意义如下：

thread: 线程的标识符，为一个整型数字。

retval: 线程退出的状态。

1.2.3使用OpenMP实现斐波那契数列

OpenMP是一种用于共享内存并行系统的多线程程序设计方案，支持的编程语言包括C、C++和Fortran。

OpenMP提供了对并行算法的高层抽象描述，特别适合在多核CPU机器上的并行程序设计。编译器根据程序中添加的pragma指令，自动将程序并行处理，使用OpenMP降低了并行编程的难度和复杂度。当编译器不支持OpenMP时，程序会退化成普通(串行)程序。程序中已有的OpenMP指令不会影响程序的正常编译运行。

OpenMP采用fork-join的执行模式。开始的时候只存在一个主线程，当需要进行并行计算的时候，派生出若干个分支线程来执行并行任务。当并行代码执行完成之后，分支线程会合，并把控制流程交给单独的主线程。

如果想要并行化循环，我们只需要在for之前插入一个pragma就可以实现并行化。而且，通过把这些恼人的细节都丢给编译器，你可以花费更多的时间来决定哪里需要多线程和优化数据结构。故此应该在for循环之前加上此指令

#pragma omp parallel for

在使用OpenMP的时候不能在使用之前的数列的方式来实现斐波那契数列，因为会造成冲突，这时应该使用绝对的数学公式实现

1.2.4运用MPI实现斐波那契数列

MPI相关知识如下所示

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。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。

在C语言中我们可以通过包含mpi.h来获得MPI相关的函数声明与数据结构的定义。当用mpicc编译链接好程序后，我们可以使用mpirun来运行编译链接好的MPI程序。mpirun的使用方法很简单，只要设置想要并行进程的数量和想运行哪个MPI程序即可。例如：mpirun -np 4 ./helloMPI的意思就是会跑4个helloMPI程序。

MPI常用函数

MPI调用借口的总数虽然庞大，但常用的几个函数分别为MPI\_Init、MPI\_Comm\_size、MPI\_Comm\_rank、MPI\_Send、MPI\_Recv和MPI\_Finalize。

MPI\_Init

函数声明如下：

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。除MPI\_Initialized()外， 其余所有的MPI函数应该在其后被调用。MPI系统将通过argc,argv得到命令行参数（也就是说main函数必须带参数，否则会出错）。

MPI\_Finalize函数声明如下：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(即rank为0的进程)上运行，但不能再有MPI函数(包括MPI\_Init())。

MPI\_Comm\_size函数声明如下：int MPI\_Comm\_size (MPI\_Comm comm ,int\* size)

该函数用于获得并行的进程个数。并将个数返回给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。

MPI\_Comm\_rank函数声明如下：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。

MPI\_Send函数声明如下：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。

MPI\_Recv函数声明如下：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。

1.2.5使用CUDA实现斐波那契数列

CUDA C程序和写C语言的程序差不多，只不过在写代码的时候多了几个CUDA的概念而已。

首当其冲的就是主机与设备，由于CUDA的架构设计，主机内存的指针和设备内存的指针是不能直接赋值使用的。所以CUDA C提供了分配设备内存与内存拷贝的API。

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。

## 1.3实验结果分析

经平台测试，各方案的执行时间如下：

串行计算：0.708秒

pthread并行计算：1.132秒

OpenMP并行计算：1.164秒

MPI并行计算：1.653秒

CUDA并行计算：5.52秒

从结果分析来看，串行计算的速度是最快的。由于问题较小，运算较为简单，故而串行时间较短，而创建和销毁线程，为进程分配内存，这些操作的时间开销远远大于计算斐波那契数列的时间开销。

## 1.4实验小结

在本次实验中，通过简单计算斐波那契数列，了解到，当问题规模较小的时候，简单求解的效率远大于并行等其他求解的策略，因此，在解决一个问题时，并行并一定是最快的方法，需要根据问题的实际情况来考虑是否适合并行化处理。