

课 程 实 验 报 告

**课程名称： 斐波那契数列计算**

|  |  |
| --- | --- |
| 专业班级： | CS1607 |
| 学 号： | U201614702 |
| 姓 名： | 樊俊超 |
| 指导教师： | 陆枫 |
| 报告日期： | 2019/7/29 |

目录

[实验一 串行环境下的斐波那契数列计算 3](#_Toc15421746)

[1.1 实验目的与要求 3](#_Toc15421747)

[1.2 实验内容与设计 3](#_Toc15421748)

[1.3 算法描述 3](#_Toc15421749)

[实验二 pthread环境下的斐波那契数列计算 4](#_Toc15421750)

[2.1 实验目的与要求 4](#_Toc15421751)

[2.2 实验内容 4](#_Toc15421752)

[2.3 算法描述 4](#_Toc15421753)

[实验三 OpenMP环境下的斐波那契数列计算 5](#_Toc15421754)

[3.1 实验目的与要求 5](#_Toc15421755)

[3.2 实验内容 5](#_Toc15421756)

[3.3 算法描述 5](#_Toc15421757)

[实验四 MPI环境下的斐波那契数列计算 6](#_Toc15421758)

[4.1 实验目的与要求 6](#_Toc15421759)

[4.2 实验内容 6](#_Toc15421760)

[4.3 算法描述 6](#_Toc15421761)

[实验五 CUDA环境下的斐波那契数列计算 7](#_Toc15421762)

[5.1 实验目的与要求 7](#_Toc15421763)

[5.2 实验内容 7](#_Toc15421764)

[5.3 算法描述 7](#_Toc15421765)

[斐波那契数列体会 8](#_Toc15421766)

[附录 8](#_Toc15421767)

[实验一代码 8](#_Toc15421768)

[实验二代码 9](#_Toc15421769)

[实验三代码 10](#_Toc15421770)

[实验四代码 11](#_Toc15421771)

[实验五代码 13](#_Toc15421772)

# 实验一 串行环境下的斐波那契数列计算

## 实验目的与要求

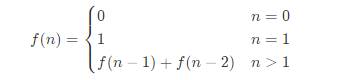
在串行环境下编写计算斐波那契数列的C语言小程序，并按要求输出对应的斐波那契数列。

## 实验内容与设计

斐波那契数列(Fibonacci sequence)，又称黄金分割数列、因数学家列昂纳多·斐波那契(Leonardoda Fibonacci)以兔子繁殖为例子引入，故又称为“兔子数列”。

该指的是这样一个数列：1、1、2、3、5、8、13、21、34、……。如果问你34后面应该是什么数字？你可能会发现数列中第i个数是数列中第i-1个数与第i-2个数的和。比如数列中的5等于2+3，数列中的13等于5+8。所以34后面的数字等于21+34=55。

找到这个规律后，我们其实就能总结出斐波那契数列的数学公式。公式如下：



## 算法描述

由于实验有严格的格式要求，所以按照所给公式分情况讨论即可。

创建FIBO函数，传入参数n，分n=0，n=1，n>1三种情况讨论。n>1时，由于n很大时得到的f(n)会很大，所以采用long long int和动态分配数组记录f(n)，得到最后结果，代码见附录。

# 实验二 pthread环境下的斐波那契数列计算

## 2.1 实验目的与要求

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。

## 2.2 实验内容

进程与线程

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即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。创建完线程后，为防止子线程未运行完毕主线程就退出的情况，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)

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)

## 2.3 算法描述

本次实验也很简单，仿照给出的线程例子，将实验一中的计算方法改为单独计算第n个斐波那契数列值，公式如下



在主函数中调用pthread\_create函数为每个i ( i >= 1 && i <= n )值，并调用pthread\_join函数防止主线程先退出，在每个子线程中单独计算一个i值对应的斐波那契数列值，代码见附录。

# 实验三 OpenMP环境下的斐波那契数列计算

## 3.1 实验目的与要求

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。

## 3.2 实验内容

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。

## 3.3 算法描述

第三次实验同样采用实验二的计算方法，即利用OpenMP函数库完成多线程调用，只需在计算第i个斐波那契数列值的for循环前使用指令#pragma omp parallel for即可，但需要注意OpenMP对可并行化for循环的要求，代码见附录。

# 实验四 MPI环境下的斐波那契数列计算

## 4.1 实验目的与要求

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。

## 4.2 实验内容

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。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定义。

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。

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。

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：

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。

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。

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。

## 4.3 算法描述

在实验中，由于本身经MPI\_Comm\_size函数获得的进程个数并不确定，所以在实验中要采用模运算来合理调度进程。在主进程中即MPI\_Comm\_rank函数获得的id为0的进程中要在获得斐波那契n值后传入各个子进程中，在子进程中通过公式计算传入数字作为下标的斐波那契数列值，并调用MPI\_Send函数返回到主进程，在主进程中利用数组存储对应下标数列值。代码见附录。

# 实验五 CUDA环境下的斐波那契数列计算

## 5.1 实验目的与要求

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。

## 5.2 实验内容

CUDA是由NVIDIA公司创立的基于他们公司生产的图形处理器GPUs的一个并行计算平台和编程模型。写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 );

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 );

cudaFree函数可以释放使用cudaMalloc分配的内存。

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。

## 5.3 算法描述

本次实验也很简单，并行计算到n为止的斐波那契数列值，并将结果存储到一个数组中，在计算完成后将设备数据拷贝到主机内存，并显示出来即可。其中，拷贝函数需要用到cudaMemcpy函数，并行执行需要用到sumOfSquares函数，代码见附录。

# 斐波那契数列体会

由于斐波那契数列比较简单，在编程方面倒是没有遇到很大的问题，实验中给出的五种环境和要求下的编程中，最难的可能就是MPI环境下的斐波那契数列计算，相对于CUDA针对设备与主机，OpenMP针对编译器，MPI是一种基于信息传递的并行编程技术，使用它时最直观的感受就是信息的交换必然用到传递函数，同时对传递对象，传递要求都有规定，在调用子线程时也要注意在通信空间中的逻辑编号。

本次实验，我对OpenMP、MPI等几种环境有了一定的了解，同时也知道了并行区别于串行的不同的地方。

# 附录

## 实验一代码

#include <stdio.h>

#include <stdlib.h>

void FIBO(int n){

if(n==0)

printf("0\n");

else if(n==1)

printf("1\n");

else{

long long int \*num = (long long int\*)malloc(sizeof(long long int)\*n);

num[0]=1;

num[1]=1;

printf("1 1 ");

for(int i=2;i<n;i++){

num[i]=num[i-1]+num[i-2];

if(i==n-1)

printf("%lld\n",num[i]);

else

printf("%lld ",num[i]);

}

}

}

int main()

{

int n;

scanf("%d",&n);

if(n>=0){

FIBO(n);

}

return 0;

}

## 实验二代码

#include <stdio.h>

#include <pthread.h>

#include <math.h>

#include <stdlib.h>

void FIBO(int n)

{

double an = (1.0 / sqrt(5.0)) \* (pow(((1 + sqrt(5.0)) / 2), n) - pow(((1 - sqrt(5.0)) / 2), n));

long long int i = an;

printf("%lld", i);

return;

}

int main()

{

int i, n;

scanf("%d", &n);

for (i = 1; i <= n; ++i) {

pthread\_t th;

int ret = pthread\_create(&th, NULL, (void \*)&FIBO, i);

if (ret != 0) {

printf("pthread create error");

return -1;

}

pthread\_join(th, NULL);

if (i != n)

printf(" ");

else

printf("\n");

}

return 0;

}

## 实验三代码

#include <stdlib.h>

#include <stdio.h>

#include <math.h>

#include <string.h>

#include <omp.h>

int main()

{

int n, i;

scanf("%d", &n);

long long int\* p = (long long int\*)malloc(sizeof(long long int) \* (n + 1));

#pragma omp parallel for

for (i = 1; i <= n; ++i) {

double ai = (1.0 / sqrt(5.0)) \* (pow(((1 + sqrt(5.0)) / 2), i) - pow(((1 - sqrt(5.0)) / 2), i));

p[i] = ai;

}

for (i = 1; i <= n; ++i) {

printf("%lld", p[i]);

if (i != n)

putchar(' ');

else

putchar('\n');

}

}

## 实验四代码

#include <stdlib.h>

#include <stdio.h>

#include <mpi.h>

#include <math.h>

#include <string.h>

int main(int argc, char\* argv[]) {

int numprocs, myid, i, n, src;

long long int buf;

MPI\_Status status;

MPI\_Request request;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myid);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &numprocs);

if (myid == 0) {

scanf("%d", &n);

for (i = 1; i < numprocs; ++i) {

MPI\_Send(&n, 1, MPI\_INT, i, 99, MPI\_COMM\_WORLD);

}

long long int\* p = (long long int\*)malloc(sizeof(long long int) \* (n + 1));

for (i = 1; i <= n; ++i) {

if (i % numprocs) {

MPI\_Send(&i, 1, MPI\_INT, (i % numprocs), 99, MPI\_COMM\_WORLD);

MPI\_Recv(&p[i], 1, MPI\_LONG\_LONG\_INT, (i % numprocs), 99, MPI\_COMM\_WORLD, &status);

}

else {

p[i] = (1.0 / sqrt(5.0)) \* (pow(((1 + sqrt(5.0)) / 2), i) - pow(((1 - sqrt(5.0)) / 2), i));

}

}

for (i = 1; i <= n; ++i) {

printf("%lld", p[i]);

if (i != n)

putchar(' ');

else

putchar('\n');

}

free(p);

}

else {

MPI\_Recv(&n, 1, MPI\_INT, 0, 99, MPI\_COMM\_WORLD, &status);

if ((n > numprocs) || (n >= myid)) {

for (i = 0; i < ((n - myid) / numprocs + 1); ++i) {

MPI\_Recv(&src, 1, MPI\_INT, 0, 99, MPI\_COMM\_WORLD, &status);

buf = (1.0 / sqrt(5.0)) \* (pow(((1 + sqrt(5.0)) / 2), src) - pow(((1 - sqrt(5.0)) / 2), src));

MPI\_Send(&buf, 1, MPI\_LONG\_LONG, 0, 99, MPI\_COMM\_WORLD);

}

}

}

MPI\_Finalize();

return 0;

}

## 实验五代码

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <cuda\_runtime.h>

#include<device\_launch\_parameters.h>

#define THREAD\_NUM 5;

\_\_global\_\_ void FIBO(long long int\*result, int n, int thread\_num)

{

int i;

int tid = threadIdx.x;

int loop\_num = (n < thread\_num) ? 1 : (n / thread\_num);

for (i = tid \* loop\_num; (i < (tid + 1) \* loop\_num) && (i < n); ++i) {

result[i + 1] = (1.00 / sqrt(5.00)) \* (pow(((1.00 + sqrt(5.00)) / 2.00), (i + 1)) - pow(((1.00 - sqrt(5.00)) / 2.00), (i + 1))) + 0.5;

}

}

int main()

{

int n, i, thread\_num = 5;

scanf("%d", &n);

long long int\*result;

long long int\*sum = (long long int\*)malloc((n + 1) \* (sizeof(long long int)));

cudaMalloc((void \*\*)&result, (n + 1) \* (sizeof(long long int)));

FIBO <<<1, thread\_num >>>(result, n, thread\_num);

cudaMemcpy(sum, result, (n + 1) \* (sizeof(long long int)), cudaMemcpyDeviceToHost);

for (i = 1; i <= n; ++i) {

printf("%lld", sum[i]);

if (i != n)

putchar(' ');

else

putchar('\n');

}

cudaFree(result);

free(sum);

return 0;

}