

**课 程 实 验 报 告**

**课程名称： 并行编程原理与实践**

**专业班级： CS1602**

**学 号： U201614545**

**姓 名： 谭胜克**

**指导教师： 金海**

**报告日期： 2019/7/22**

**计算机科学与技术学院**

**目录**

[1串行环境下斐波纳切数列计算 1](#_Toc15047810)

[1.1实验目的与要求 1](#_Toc15047811)

[1.2算法描述 1](#_Toc15047812)

[1.3实验方案 2](#_Toc15047813)

[1.4实验结果与分析 2](#_Toc15047814)

[2 pthread环境斐波纳切数列计算 3](#_Toc15047815)

[2.1实验目的与要求 3](#_Toc15047816)

[2.2算法描述 3](#_Toc15047817)

[2.3 实验方案 4](#_Toc15047818)

[2.4 实验结果与分析 4](#_Toc15047819)

[3 OpenMP环境下斐波纳切数列计算 5](#_Toc15047820)

[3.1实验目的与要求 5](#_Toc15047821)

[3.2算法描述 5](#_Toc15047822)

[3.3实验方案 5](#_Toc15047823)

[3.4实验结果与分析 6](#_Toc15047824)

[4 MPI环境下斐波纳切数列计算 7](#_Toc15047825)

[4.1 实验目的与要求 7](#_Toc15047826)

[4.2 算法描述 7](#_Toc15047827)

[4.3 实验方案 7](#_Toc15047828)

[4.4 实验结果与分析 8](#_Toc15047829)

[5 CUDA环境下斐波纳切数列计算 9](#_Toc15047830)

[5.1 实验目的与要求 9](#_Toc15047831)

[5.2 算法描述 9](#_Toc15047832)

[5.3 实验方案 9](#_Toc15047833)

[5.4 实验结果与分析 10](#_Toc15047834)

[6 实验小结 11](#_Toc15047835)

[**附件：源代码** 12](#_Toc15047836)

# 1串行环境下斐波纳切数列计算

## 1.1实验目的与要求

1. 实验目的
2. 掌握串行环境下斐波那契数列的计算方法。
3. 为后续并行环境下的实验打好基础。
4. 实验要求
5. 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。
6. 输入: 需要输出的斐波那契额数列的长度，类型为int，如4。
7. 输出: 打印出斐波那契数列，如输入为4，输出为1 1 2 3。
8. 注意: 打印完斐波那契数列之后要换行。

## 1.2算法描述

根据图1.1所示的递推公式，可以通过递归完成计算，也可以通过非递归的方式进行。我选择了非递归的方式，减少递归过程中递归的开销。通过循环实现也很简单，根据n的值分别按照递推公式进行处理即可。

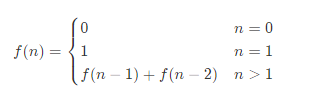


图1.1 斐波那契数列递推公式

当n=0时或n=1时，无需计算，直接返回n的值；

当n>1时，通过循环不断将前两项相加的值赋予第三项，之后在把前两项的值往后移一位，循环n-1次即可得到第n项的值。如图1.2所示，为计算部分的核心代码。

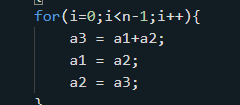


图1.2 串行计算斐波那契数列

## 1.3实验方案

1. 在本机环境下编写代码并测试正确性。
2. 将代码导入平台进行平台测试。
3. 分析代码执行的时间、空间复杂度。

## 1.4实验结果与分析

平台测试结果如图1.3所示。5个测试集均正确完成测试。



图1.3 平台测试结果

# 2 pthread环境斐波纳切数列计算

## 2.1实验目的与要求

1. 实验目的
2. 掌握pthread环境下斐波那契数列的计算方法。
3. 理解pthread并行的实现原理
4. 实验要求
5. 在pthread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。
6. 输入: 需要输出的斐波那契额数列的长度，类型为int，如4。
7. 输出: 打印出斐波那契数列，如输入为4，输出为1 1 2 3。
8. 注意: 打印完斐波那契数列之后要换行。

## 2.2算法描述

对于斐波那契额数列的计算存在一个通项公式。

pthread的作用是新申请一个线程，于是我们自然而然的就可以想到首先在程序开始运行时，首先通过启动一个新的线程去计算一定数目的斐波纳切数列，并将其存储起来，在通过主线程读取用户输入，之后根据输入选择输出的数列个数。通过多线程，将I/O与CPU计算分开，使其并行工作，减少了二者相互等待所耗费的时间。

对于斐波纳切数列的计算采用的是上述给出的通项公式，首先在子线程中完成51项斐波纳切数列的计算并存储于数组中。如图2.1所示，为计算部分的代码。

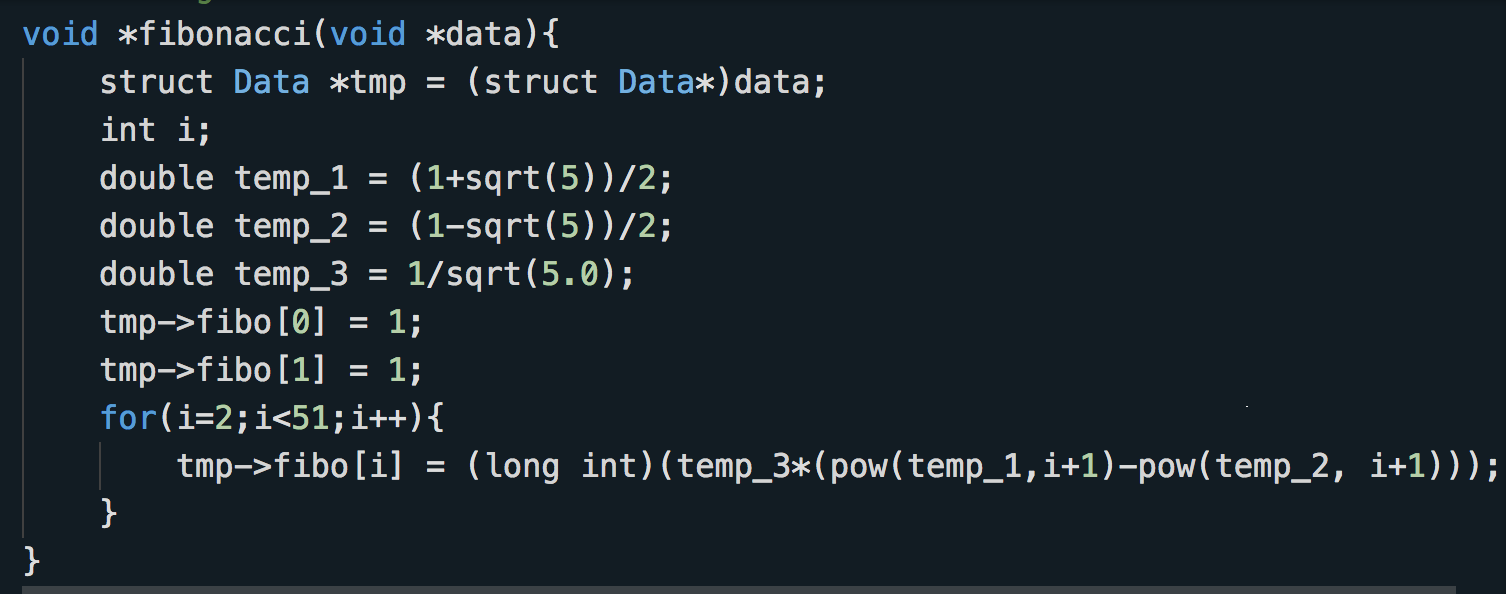


图2.1 斐波纳切计算核心代码

## 2.3 实验方案

1. 在本机环境下编写代码并测试正确性。
2. 将代码导入平台进行平台测试。
3. 分析代码执行的时间、空间复杂度。

## 2.4 实验结果与分析

如图2.2所示，在平台的5个测试集中全部正确通过。



图2.2 pthread平台测试

# 3 OpenMP环境下斐波纳切数列计算

## 3.1实验目的与要求

1. 实验目的
2. 掌握OpenMP环境下斐波那契数列的计算方法。
3. 理解OpenMP并行的实现原理
4. 实验要求
5. 在OpenMP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。
6. 输入: 需要输出的斐波那契额数列的长度，类型为int，如4。
7. 输出: 打印出斐波那契数列，如输入为4，输出为1 1 2 3。
8. 注意: 打印完斐波那契数列之后要换行。

## 3.2算法描述

OpenMP并行编程是通过编译器自动完成多线程的申请以及销毁，只需在需要并行处理的地方添加上相应的指令即可。这里不再用提前计算存储的方式，而是先读取用户的输入，之后通过OpenMP并行计算。

核心代码如图3.1所示。在利用通项公式计算斐波纳切数列时，加上#pragma omp parallel for指令，指明接下来的循环需要采用并行化的方式实现。

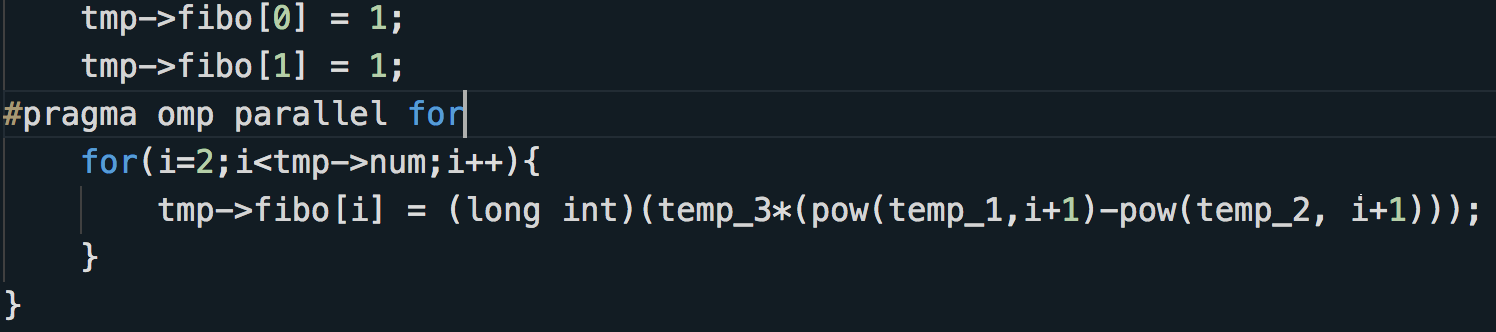


图3.1 OpenMP核心代码

## 3.3实验方案

1. 在本机环境下编写代码并测试正确性。
2. 将代码导入平台进行平台测试。
3. 分析代码执行的时间、空间复杂度。

## 3.4实验结果与分析

如图3.2所示，在平台的5个测试集中全部正确通过。



图3.2 OpenMP平台测试

# 4 MPI环境下斐波纳切数列计算

## 4.1 实验目的与要求

1. 实验目的
2. 掌握MPI环境下斐波那契数列的计算方法。
3. 理解MPI并行的实现原理
4. 实验要求
5. 在MPI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。
6. 输入: 需要输出的斐波那契额数列的长度，类型为int，如4。
7. 输出: 打印出斐波那契数列，如输入为4，输出为1 1 2 3。
8. 注意: 打印完斐波那契数列之后要换行。

## 4.2 算法描述

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。

由于MPI程序需要在运行时输入需要启动的进程个数，所以我们需要对任务进行划分。划分的依据就是进程个数。为了减少I/O等待时间，采用先计算后输出的方式。

这里首先计算的斐波纳切数列的个数为51个。首先通过MPI\_Comm\_size（）函数获取当前程序的进程总数n，对所有进程进行相应的任务划分。0号进程输入输出进程，其余进程作为计算进程。将51个斐波纳切数均分给剩余的n-1个进程，很可能会出现不能整除的现象。所以需要对最后一个进程单独处理，让最后一个进程完成剩余的所有任务的计算，就是将不能整除的部分都划分给最后一个进程去计算。

每一个计算进程通过自己的进程id获得相应的任务划分区域，开始基址为：id\*（51/（n-1）），结束地址为：id\*（51/n）。最后一个进程结束地址为51。这样以来就完成的任务的划分。

最后只需要根据用户的输入，选择计算完成的数据进行输出即可。

## 4.3 实验方案

1. 在本机环境下编写代码并测试正确性。
2. 将代码导入平台进行平台测试。
3. 分析代码执行的时间、空间复杂度。

## 4.4 实验结果与分析

如图4.1所示，在平台的5个测试集中全部正确通过。



图4.1 MPI平台测试

# 5 CUDA环境下斐波纳切数列计算

## 5.1 实验目的与要求

1. 实验目的
2. 掌握CUDA环境下斐波那契数列的计算方法。
3. 理解CUDA并行的实现原理
4. 实验要求
5. 在CUDA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。
6. 输入: 需要输出的斐波那契额数列的长度，类型为int，如4。
7. 输出: 打印出斐波那契数列，如输入为4，输出为1 1 2 3。
8. 注意: 打印完斐波那契数列之后要换行。

## 5.2 算法描述

CUDA并行编程实现斐波纳切数列计算的思想与MPI 较为相似。只不过这里需要将在设备内存中完成计算存储的数组通过特定的cudaError\_t cudaMalloc (void \*\*devPtr, size\_t size );函数拷贝到主存中才能输出。

为了便于划分，这里设定需要计算的斐波纳切数列个数为60，启动的线程个数为10。所以划分下来每个线程需要计算的斐波纳切数列个数为10。完成计算后通过拷贝函数对结果进行拷贝，等待用户输入，输出相应个数的斐波纳切数列。

## 5.3 实验方案

1. 在本机环境下编写代码并测试正确性。
2. 将代码导入平台进行平台测试。
3. 分析代码执行的时间、空间复杂度。

## 5.4 实验结果与分析

如图5.1所示，在平台的5个测试集中全部正确通过。



图5.1 CUDA平台测试

# 6 实验小结

斐波纳切数列计算的5个实验难度不大，更多的是让我们了解关于并行程序设计的基本概念、基本方法。通过对同一个问题：斐波纳切数列计算。采用5种不同的方案去实现，能够让我很好的理解不同方案实现的差别。在实验过程中，让我感触最深的就是实现模式的转变。在串行程序设计时，我只是考虑到，用户输入什么数字，我就应该去计算什么数据。到后面并行化设计时，我的设计思路转变为，首先计算多少个数据，不再根据用户输入去计算，而是根据用户输入去输出。这样一来，在用户进行I/O的时候，就可以同时完成计算，当用户I/O完成时，计算也已经完成，这样以来就大大减少了CPU与I/O进程间相互等待所耗费的时间。尽管从程序内存消耗上来看，这样做牺牲了一定的存储空间对数据进行存储，但从效果来看，确实很值得的。

同时在设计并行程序时，需要考虑到很多并发的问题，每一个线程都需要合理的划分任务，否则就很有可能会出现错误。

总之，通过这5个关于斐波纳切数列计算的实验，让我初步了解到关于并行化程序设计的一些基本概念和方法，并通过简单的操作实现了这些方法，深刻感受到了并行程序与串行程序之间思想的差别。

**附件：源代码**

1. **串行：**

//

// main.c

// fibonacci

//

// Created by Doris on 2019/7/9.

// Copyright © 2019年 tsk. All rights reserved.

//

#include <stdio.h>

long int fibonacci(int n);

/\*\*\*\*\* Begin \*\*\*\*\*/

int main()

{

int data = 0;

long int out = 0;

int i = 1;

scanf("%d",&data);

while(i <= data){

out = fibonacci(i);

if(i==data) printf("%ld\n",out);

else printf("%ld ",out);

i++;

}

return 0;

}

long int fibonacci(int n){

long int a1=0,a2=1;

long int a3=1;

int i;

if(n <= 0)

return 0;

else if(n == 1)

return 1;

else{

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

a3 = a1+a2;

a1 = a2;

a2 = a3;

}

}

return a3;

}

1. **pthread**

#include <stdio.h>

#include <pthread.h>

#include <math.h>

#include <stdlib.h>

#define MAX\_NUM 100

struct Data

{

int num;

long int fibo[MAX\_NUM];

};

/\*\*\*\*\* Begin \*\*\*\*\*/

void \*fibonacci(void \*data){

struct Data \*tmp = (struct Data\*)data;

int i;

double temp\_1 = (1+sqrt(5))/2;

double temp\_2 = (1-sqrt(5))/2;

double temp\_3 = 1/sqrt(5.0);

tmp->fibo[0] = 1;

tmp->fibo[1] = 1;

for(i=2;i<51;i++){

tmp->fibo[i] = (long int)(temp\_3\*(pow(temp\_1,i+1)-pow(temp\_2, i+1)));

}

}

int main()

{

struct Data data;

pthread\_t th;

int ret;

int num;

int i;

//create thread

ret = pthread\_create(&th, NULL, fibonacci, (void \*)&data);

pthread\_join(th, NULL);

//get input

scanf("%d",&num);

getchar();

//output

for(i = 0; i < num; i++){

if(i == num - 1){

printf("%ld", data.fibo[i]);

}

else{

printf("%ld ", data.fibo[i]);

}

}

printf("\n");

return 0;

}

/\*\*\*\*\* End \*\*\*\*\*/

1. **OpenMP**

#include <stdlib.h>

#include <stdio.h>

#include <math.h>

#include <string.h>

#define MAX\_NUM 100

struct Data

{

int num;

long int fibo[MAX\_NUM];

};

/\*\*\*\*\* Begin \*\*\*\*\*/

void fibonacci(void \*data){

struct Data \*tmp = (struct Data\*)data;

int i;

double temp\_1 = (1+sqrt(5))/2;

double temp\_2 = (1-sqrt(5))/2;

double temp\_3 = 1/sqrt(5.0);

if(tmp->num == 0){

tmp->fibo[0] = 1;

return;

}

if(tmp->num == 1){

tmp->fibo[0] = 1;

return;

}

tmp->fibo[0] = 1;

tmp->fibo[1] = 1;

#pragma omp parallel for

for(i=2;i<tmp->num;i++){

tmp->fibo[i] = (long int)(temp\_3\*(pow(temp\_1,i+1)-pow(temp\_2, i+1)));

}

}

int main()

{

struct Data data;

int num;

int i;

//get input

scanf("%d",&num);

getchar();

data.num = num;

fibonacci(&data);

//output

for(i = 0; i < num; i++){

if(i == num - 1){

printf("%ld", data.fibo[i]);

}

else{

printf("%ld ", data.fibo[i]);

}

}

printf("\n");

return 0;

}

/\*\*\*\*\* End \*\*\*\*\*/

1. **MPI**

#include <stdlib.h>

#include <stdio.h>

#include <mpi.h>

#include <math.h>

#include <string.h>

/\*\*\*\*\* Begin \*\*\*\*\*/

long int fibonacci(int n){

long int data;

double temp\_1 = (1+sqrt(5))/2;

double temp\_2 = (1-sqrt(5))/2;

double temp\_3 = 1/sqrt(5.0);

data = (long int)(temp\_3\*(pow(temp\_1,n)-pow(temp\_2, n)));

return data;

}

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

{

int numproc, myid, source;

int num;

MPI\_Status status;

MPI\_Init(&argc, &argv);

//获取进程id

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

//获取进程数

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

//按照进程数划分任务

int local\_num = 51/(numproc-1);

int local\_begin;

int local\_end;

long int\* data = (long int\*)malloc(sizeof(long int)\*51);

long int\* recv = (long int\*)malloc(sizeof(long int)\*51);

memset(data, 51, 0);

memset(recv, 51, 0);

//0号进程控制I/O

if(myid == 0){

//get input

scanf("%d",&num);

getchar();

int j;

for (source = 1; source < numproc; ++source){

MPI\_Recv(recv, 51, MPI\_LONG, source, 99,

MPI\_COMM\_WORLD, &status);

if(source == numproc - 1){ //最后一个进程传来的数据

local\_end = 51;

local\_begin = local\_num\*(source-1);

}

else{

local\_end = local\_num\*source;

local\_begin = local\_num\*(source-1);

}

printf("id:%d,begin:%d,end:%d\n",source,local\_begin,local\_end);

for(j=local\_begin;j<local\_end;j++){

data[j]=recv[j];

}

}

for(j = 1; j < num; j++){

printf("%ld ", data[j]);

}

printf("%ld\n",data[num]);

}

else if(myid == (numproc - 1)){ //最后一个进程

local\_begin = local\_num\*(numproc-2);

local\_end = 51;

int i;

for(i=local\_begin;i<local\_end;i++){

data[i] = fibonacci(i);

}

MPI\_Send(data,51,MPI\_LONG,0,99,MPI\_COMM\_WORLD);

}

else{

local\_begin = local\_num\*(myid - 1);

local\_end = local\_num\*myid;

int i;

for(i=local\_begin;i<local\_end;i++){

data[i] = fibonacci(i);

}

MPI\_Send(data,51,MPI\_LONG,0,99,MPI\_COMM\_WORLD);

}

MPI\_Finalize();

return 0;

}

/\*\*\*\*\* End \*\*\*\*\*/

1. **CUDA**

#include <stdio.h>

#include <math.h>

#define DATA\_SIZE 60

#define THREAD\_NUN 10

/\*\*\*\*\* Begin \*\*\*\*\*/

\_\_global\_\_ void fibonacci(long int\* data,double a,double b, double c){

int tid = threadIdx.x;

int i;

const int size = DATA\_SIZE/THREAD\_NUN;

for(i=tid\*size;i<(tid+1)\*size;i++){

data[i] = (long int)((c\*(pow(a,i)-pow(b,i)))+0.5);

}

}

int main()

{

int num, i;

long int data[DATA\_SIZE];

long int\* result;

cudaMalloc((long int \*\*)&result,sizeof(long int)\*DATA\_SIZE);

double temp\_1 = (1+sqrt(5))/2;

double temp\_2 = (1-sqrt(5))/2;

double temp\_3 = 1/sqrt(5.0);

fibonacci<<<1,THREAD\_NUN>>>(result,temp\_1,temp\_2,temp\_3);

cudaMemcpy(&data,result,sizeof(long int)\*DATA\_SIZE,cudaMemcpyDeviceToHost);

//get input

scanf("%d",&num);

getchar();

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

printf("%ld ", data[i]);

}

printf("%ld\n", data[num]);

return 0;

}

/\*\*\*\*\* End \*\*\*\*\*/