華中科技大學

课 程 实 验 报 告

课程名称： 并行编程原理与实践

专业班级：	**IOT1601**
学号：	**U201614898**
姓名：	**潘翔**
指导教师：	**陆枫**
报告日期：	**2019.7**

计算机科学与技术学院

目 录

1 实验一

1.1 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。

1.2 实验内容

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)以兔子繁殖为例子而引入，故又称为“兔子数列”。

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

公式 1.1 Fibonacci 递推

1.3 实验结果

代码 1.1 串行斐波那契数列

```
#include<stdio.h>
#define int long
int a[100];
int f(int i)
{
    if(a[i]!=-1)
    {
        return a[i];
    }
    else
    {
        if(i==0)
        {
            a[i]=0;
            return a[i];
        }
        if(i==1)
        {
            a[i]=1;
            return a[i];
        }
```

```
        if(i>1)
        {
            a[i]=f(i-1)+f(i-2);
            return a[i];
        }
    }

}

int main()
{
    //init
    for(int i=0;i<100;i++)
    {
        a[i]=-1;
    }

    int n;
    scanf("%ld",&n);
    for(int i=1;i<n+1;i++)
    {
        printf("%ld",f(i));
        if(i!=(n))
        {
            printf(" ");
        }
    }
    printf("\n");
    return 0;
}
```

Time:0.678s

1.4 实验总结

实验十分简单，主要是为了与并行程序进行对比

2 实验二

2.1 Thread

2.1.1 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。

2.1.2 实验环境

1) 本地环境：
2) 在线环境：educoder

2.1.3 算法描述

采用公式计算解决数据依赖，利用多线程(两线程)并行计算奇数和偶数。

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

利用多线程进行计算写入数组，最后等待计算完成，串行

2.1.4 实验方案

1) 算法过程

图 2.1 thread 算法流程图

2) 算法实现

代码 2.1 thread 并行算法

```
#include <stdio.h>
#include <pthread.h>
#include <math.h>
```

```c
#include <stdlib.h>
#define ll long long
ll a[100];
int n=0;
int thread_num=16;


void* par_fun(void * tn)
{
        // int tn=*((int *)tn);
    int par_tn=*((pthread_t *)tn);
    // printf("par_tn:%d\n",par_tn);
        for(int i=par_tn;i<=n+1;i+=thread_num)
        {

        a[i]=(long)(1.0/sqrt(5.0)*(pow((1.0+sqrt(5.0))/2.0,i)-pow((1.0-s
qrt(5.0))/2.0,i)));
        }
}



int main()
{
        scanf("%d",&n);
    // thread_num=(int)(n/2+0.5);
    pthread_t threads[thread_num];
    int tn;
    for(tn=0;tn<thread_num;tn++)
    {
        pthread_create(&threads[tn],NULL,par_fun,(void *)&tn);
        usleep(20);
    }
    for(tn=0;tn<thread_num;tn++)
    {
        pthread_join(threads[tn],NULL);
    }
        for(int i=1;i<n;i++)
        {
                printf("%lld ",a[i]);
        }
    printf("%lld\n",a[n]);
    return 0;
}
```

2.1.5 实验结果分析

educoder显示时间：1.092 秒

因为线程的创建需要时间，虽然是共享相应的变量和函数空间而不用进行拷贝，可是分配资源和函数调用需要，而测试集整体较小，所以效率上可能不如串行。

2.2 OpenMP

2.2.1 实验目的

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。

2.2.2 实验环境

本地环境:
编译器：gcc version 9.1.0 (MacPorts gcc9 9.1.0_2)
并行库：OpenMP
在线环境: educoder

2.2.3 算法描述

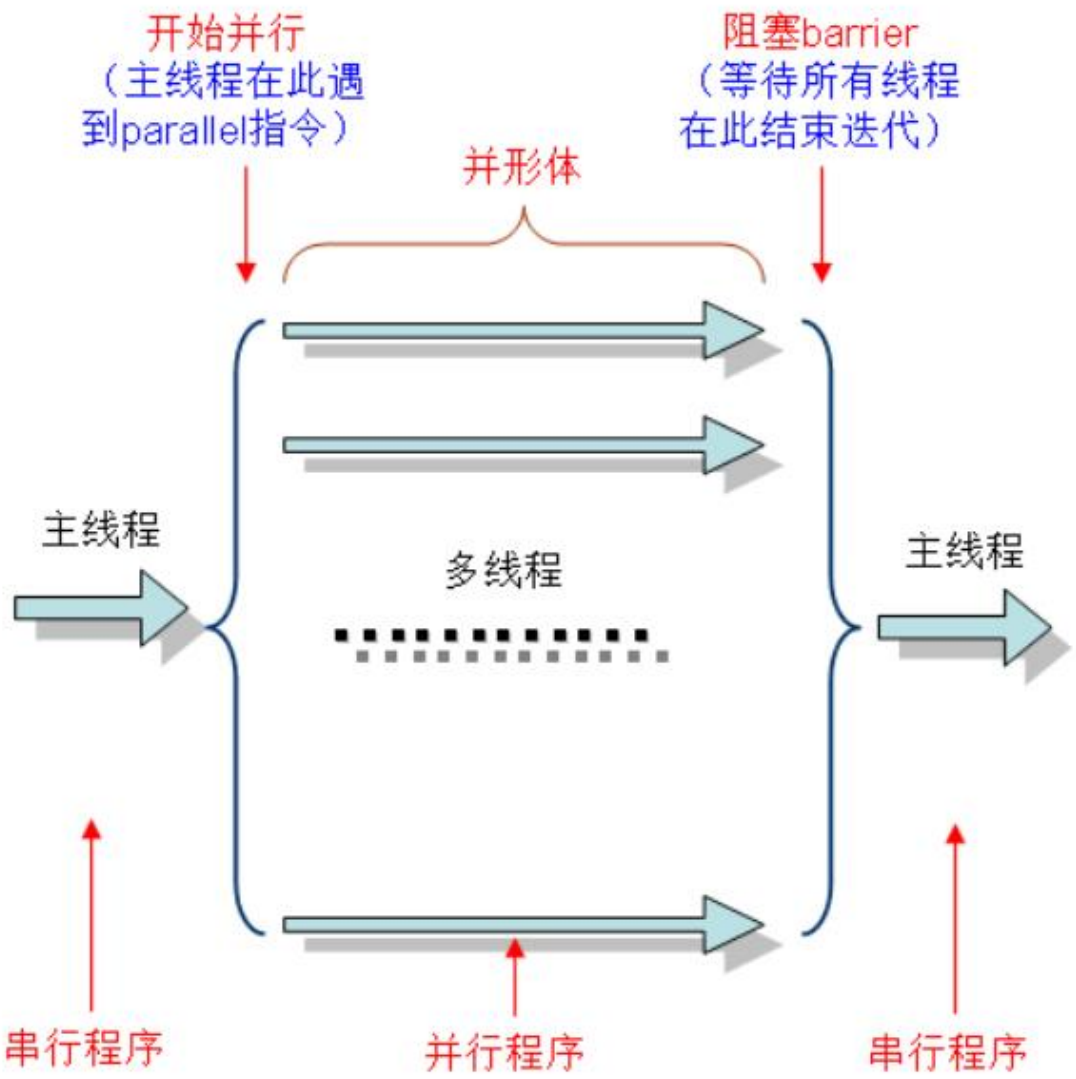

图 2.2 OpenMp 并行流程

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。

直接利用#pragma omp parallel for对条件循环部分进行并行从而OpenMP自动进行线程级别并行。

2.2.4 实验方案

代码 2.2 OpenMP 解决 fibonacci 数列

```
//OpenMp
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <math.h>

#define ll long long
ll a[100];

ll F(int n)
{
    return
(ll)(1.0/sqrt(5.0)*(pow((1.0+sqrt(5.0))/2.0,n)-pow((1.0-sqrt(5.0))/2.0,n
)));
}

int main()
{
        int n=0;
        omp_set_num_threads(2);
        scanf("%d",&n);
        getchar();
        a[0]=1;
        a[1]=1;
        #pragma omp parallel for
        for(int i=2;i<n;i++)
        {
                a[i]=F(i);
```

```
        }
        for(int i=0;i<n-1;i++)
    {
        printf("%lld ",a[i]);
    }
        printf("%lld\n",a[n-1]);
        return 0;
}
```

2.2.5 实验结果分析

educoder时间:1.01s，相较于自己手动创建线程有了一定程度的提升，可能是框架层面的优化，也可能是循环的控制导致没有进行更多的冗余计算。

2.3 MPI

2.3.1 实验目的

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。

2.3.2 实验环境

本地环境：
gcc version 9.1.0 (MacPorts gcc9 9.1.0_2)
MPI: clang-mpi(Thread model: posix)
在线环境：educoder

2.3.3 算法描述

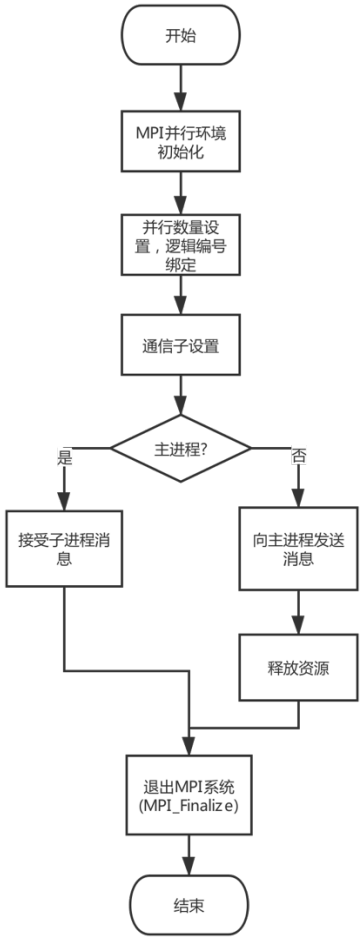

图 2.3 MPI 并行程序设计

2.3.4 实验方案

由于在实验平台中，并未指定参数输入，导致需要等待std输入流造成线程卡顿，故直接采用最大值进行计算从而进行输出，实际过程中，创建线程仍然是最大开销。

代码 2.3 MPI 解决 fibonacci

```
//MPI
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>
#include <math.h>
#include <string.h>
#define ll long long

ll F(int n)
```

```
{
    return
(ll)(1.0/sqrt(5.0)*(pow((1.0+sqrt(5.0))/2.0,n)-pow((1.0-sqrt(5.0))/2.0,n
)));
}
int main(int argv,char* argc[])
{
    int n = 0;

    int max_len=55;

    int process_id=0;
    int process_num=0;

    MPI_Status status;
    MPI_Init(&argv,&argc);
    MPI_Comm_rank(MPI_COMM_WORLD,&process_id);
    MPI_Comm_size(MPI_COMM_WORLD, &process_num);

    ll* message = (ll*)malloc(sizeof(ll)*max_len);
    ll* recv = (ll*)malloc(sizeof(ll)*max_len);
    memset(message, 0, max_len);
    memset(recv, 0, max_len);

    int loop = ceil(max_len/(process_num-1));
    if(process_id != 0) // send message
    {
        for(int i = ( process_id - 1 ) * loop; i < process_id*loop; i++)
        {
            message[i] = F(i);
        }
        MPI_Send(message, max_len, MPI_LONG_LONG_INT, 0, 0,
MPI_COMM_WORLD);
    }
    else//receive message
    {
        scanf("%d", &n);
        for(int i = 1; i < process_num; ++i)
        {
            MPI_Recv(recv, max_len, MPI_LONG_LONG_INT, i, 0,
MPI_COMM_WORLD, &status);
            for(int j = (i-1)*loop; j < i*loop; j++)
            {
                message[j] = recv[j];
```

```
            }
        }
        for(int i = 1; i < n; ++i)
        {
            printf("%lld ", message[i]);
        }
        if (n != 1)
            printf("%lld\n", message[n]);
        else
            printf("1\n");
        free(message);
        free(recv);
    }
    MPI_Finalize();
    return 0;
}
```

2.3.5 实验结果分析

Educoder时间：1.605 s

由于并未严格按照输入数据范围进行计算，直接打大表，造成一定的开销，如果允许命令行参数的话，在本地测试过程中，可以直接初始化需要计算的范围，不会造成程序卡顿，同时由于存在通信开销，故时间略长。

2.4 CUDA

2.4.1 实验目的

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。

2.4.2 实验环境

本地环境：

OS：Manjaro 18.04

GCC：9.1

CUDA：9.0

在线环境：educoder

2.4.3 算法描述

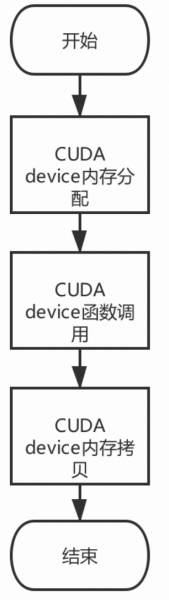

图 2.4 CUDA 流程

2.4.4 实验方案

根据输入的n值进行内存分配，然后分配线程进行计算，其中可以采用公式进行运算，也可是采用递推形式，但是采用递推形式的话不解决数据依赖并没有达到并行优化的效果。

如果采用公式的话，由于device不能使用host function，而采用内置的_powf精度较低，当n比较大的时候会出错。故采用原始函数空间的pow。此时的pow没有重载，需要使用(double,double参数)。

代码 2.4 CUDA 解决 fibonacci 问题

```
//CUDA
#include <stdio.h>
// #include <math.h>
#define ll long long

ll F(int n)
{
```

```
    return
(ll)(1.0/sqrt(5.0)*(pow((1.0+sqrt(5.0))/2.0,n)-pow((1.0-sqrt(5.0))/2.0,n
)));
}

__global__ static void fibo_cuda(ll* result){
    int tid =threadIdx.x + 1;
    ll f3 = 0;
    if(tid==0)
    {
        f3 = 0;
    }
    else if(tid==1) // print from 1
    {
        f3 = 1;
    }
    else
    {
        ll n=tid;

f3=(ll)(1.0/sqrt(5.0)*(pow((1.0+sqrt(5.0))/2.0,(double)n)-pow((1.0-sqrt(
5.0))/2.0,(double)n)));
    }
    result[tid] = f3;
}


int main()
{
    ll* p_result;
    ll result[100];
    int n;
    scanf("%d",&n);
    int N = sizeof(long)*(n+1); //add zero to the arr
    cudaMalloc((void **)&p_result,N);

    fibo_cuda<<<1,n>>>(p_result);

    cudaMemcpy(&result,p_result,N,cudaMemcpyDeviceToHost);
    for(int j = 1;j <n;j++)
    {
        printf("%lld ",result[j]);
    }
    printf("%lld\n",result[n]);
```

```
        return 0;
}
```

2.4.5 实验结果分析

Educode时间：1-2s

由于通关最初采用递推形式(后期无时间显示)，最终自行测试时间求和约在1-2s,存在拷贝空间和资源分配等高耗时操作。

2.5 串行算法复杂度分析

2.5.1 描述

分析串行算法的复杂度，并设计测试案例予以证明。

2.5.2 分析

采用递推的方式进行串行计算，其
时间复杂度：O(N)
空间复杂度：O(1)

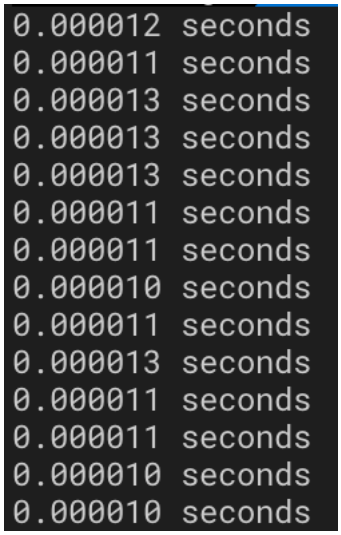

图 2.5 串行时间分析(1-100)部分

经过测试，发现程序主要的时间在IO上，即输出结果。在小数据范围内，显著差异，加上输出，则基本线性。

2.6 分析并行实现的正确性

在“斐波那契数列计算”中实现的多种并行实现中，挑选一种并分析其正确性，并请给出方案确保并行实现的正确性。

Thread形式：
分奇数和偶数采用双线程进行利用斐波拉契数列的公式进行计算，写入数组，最后串行数组，由于不存在数据依赖和输出的相关考虑，则程序结果正确。

2.7 大数场景分析

2.7.1 描述

1) 设计大数计算场景下**“斐波那契数列计算”**的测试用例并分别使用串行和并行的方式实现。
2) 分析并行实现的加速比。
3) 分析并行实现加速比的正确性。

2.7.2 分析

由于long
long的精度限制，当n=100时已经出现明显溢出，实际上算法的主要时间仍然在IO上面。

- 纯计算过程，并行加速比<并行线程数
- 整个过程中考虑初始的串行初始化过程和最后的串行输出过程
- 当 n 越大时，约趋近于并行线程数

实际过程中，由于小数据和大数据情况下具有较大的差异，实际上仍有许多书剑用于IO，所以在测量上不进行输出。对同一个程序进行多次测量的时间上可能由于缓存的原因仍有一定的差异。

对于时间的测量有两种方式：

1) 利用系统调用进行时间的测量
   此时直接采用命令行的方式进行参数给定，避免等待IO的时间误差
2) 利用 time 指令进行时间测量
   Time 指令包括三部分
   a. 实际时间(real time)：从 command 命令行开始执行到运行终止的消逝时间
   b. 用户 CPU 时间(user CPU time)：命令执行完成花费的用户 CPU 时间，即命令在用户态中执行时间总和
   c. 系统 CPU 时间(system CPU time)：命令执行完成花费的系统 CPU 时间，即命令在核心态中执行时间总和

测量过程如下：

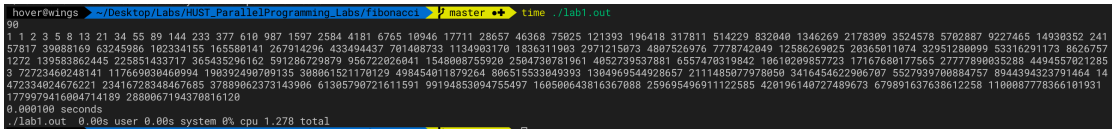

图 2.6 时间测量过程

我们发现，采用函数调用的测量精度高于time最后的时间结果。

表 2.1 时间测量结果

方法	函数调用时间
串行	0.000097 seconds
Thread	0.000694 seconds
OpenMP	0.000478 seconds
MPI	0.001041 seconds
CUDA	0.003254 seconds

2.8 并行优化方案设计

2.8.1 描述

1) 设计大数场景并行实现的优化方案
2) 测试并分析优化方案的加速比与正确性。

进行并行数的搜索，所得的并行数应该具有最高的并行加速比。

对于并行来说，对于小计算量来说，更重要的是框架的选择，因为没有太多的优化技巧，其中值得注意的就是并行量的选择，由于计算量太小，那么多开一个线程的开销可能已经远远大于并行节省的时间，此时可以考虑选择较为轻型的框架进行计算比如thread和OpenMP。

2.9 实验总结

实验过程中使用了不同的并行程序库进行同一个基础算法的比较，主要的任务是解决数据依赖和如何进行并行结果的稳定输出。由于本地 MacOS 且不想采用虚拟机，在环境方面有一段时间的折腾(Unix 与 Linux 部分 API 存在差异)，且代码迁移需要一定的修改。

实验的具体过程中，在cuda中，由于device有自己相关的函数定义，不能使用主空间的函数，所以在重载方面卡了一会，且并行编程的一个问题是参数传递，不支持命令行餐参数的情况下测试程序尚不完备，容易造成IO流卡顿。

首先并行需要解决的是数据相关性，在原始的递推形式中，由于依赖前项，从而有数据相关，当采用公式进行计算而消除了数据相关性之后，算法的性能得到了很大的提升

将串行的优化为并行，其效果分析，我们首先从串行的复杂度进行分析，串行算法的复杂度进行分析，在实际的测试过程中，发现占用性能的主要是I/O而不是计算过程，而由于本地计算性能相较于问题来说还行，CPU相应的缓存和编译的自动优化导致时间非常小，不容易测量，造成一定的麻烦。

在并行的测试上面，对于不同的框架进行了对比，从而对于不同框架的轻重程度有了直观的认识，在进行轻型并行的计算时，应该采用更加轻型化的计算框架，同时对于并行数量的选择也需要注意，注意并行造成的加速和相应的开销的对比，同时需要注意本地硬件的限制，比如本地的CPU是6核12线程，那么通常来说，并行数不应该超过12，在实际的编译过程和没有数据依赖的重型任务时候，加速比的提升和线程数的设置是线性的。

并行过程中可能涉及通信如MPI，实际上可以看到是posix:thread的高层封装，此时性能差于直接thread是可以预见的。

当设计到不同数据之间进行拷贝的时候比如CUDA可能设计显存和内存的拷贝，则其时间相对更长，同时进行设计的时候副本的拷贝也是需要考虑的。

总体上来书，软件方面：不同的框架体现了不同的特点，其瓶颈可能也不同，涉及存储和通信两个基本部分，体会了并行程序涉及需要考虑的各种因素。硬件方面：平台上所有的并行结果最后都是负优化，同时并未给明平台的硬件性能，从而说明在实际并行过程中，需要关注硬件性能和参数，从而能够获得最后的加速效果。