华中科技大学

课 程 实 验 报 告

课程名称： 并行编程原理与实践

专业班级： 计算机科学与技术（校际交流）1601 班

学　　号： U201610136

姓　　名： 朱晓光

指导教师： 陆枫

报告日期： 2019 年 7 月 11 日

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

本实验以斐波那契数列为例，主要研究和学习 C 语言实现并行编程的方法，完成实验后将掌握：

1. 掌握串行程序并行化的方法；
2. 熟悉在 Linux 的 C 语言环境下进行并行程序设计的方法；
3. 了解并行程序设计与并行算法设计的基本分析方法与途径。

1.2 实验内容

本次实验一共包含以下 5 项实验内容：

1. 在串行环境下编写计算斐波那契数列的 C 语言小程序，并按要求输入对应的斐波那契数列；
2. 编写使用 pthread 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。

1.3 算法描述

对于实验内容 1，即串行环境下计算斐波那契数列，这里使用了常见的斐波那契数列递推公式（见公式 1.1）。

$$a_n = \begin{cases} 1 & n = 1, 2 \\ a_{n-1} + a_{n-2} & n = 3, 4, 5, \dots \end{cases}$$

公式 1.1 斐波那契数列递推公式

使用该递推公式，整个数列的计算过程中仅涉及加法，适合需要顺序输出斐波那契数列的场景。但后项总是依赖前两项，若前两项还未通过计算得出，则无法计算后项。

对于后续并行环境下计算斐波那契数列的实验内容，这里使用斐波那契数列的函数直接得到（见公式 1.2）。

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

公式 1.2 斐波那契数列函数

使用该函数计算斐波那契数列的项，则后项不再依赖前项，可通过数列下标

直接计算得出。但相应的，计算过程中涉及乘除法、乘方等较加法更加耗时的运算。

1.4 实验结果与分析

在 EduCoder 平台上完成了以上所有实验内容，测试样例均通过。

其中，串行环境实验通过时平台显示耗时 2.959 秒，pthread 环境耗时 6.34 秒，OpenMP 环境耗时 5.67 秒，MPI 环境耗时 5.844 秒，CUDA 环境耗时 8.246 秒。

可以看到在使用并行方法计算斐波那契数列后，时间反而增加了。推测这可能是由于并行化时增加了多线程管理开支与使用斐波那契函数引入的计算量开支，而实现平台设定的目标斐波那契数下标过小，并行算法提供的加速比还无法抵消其增加的开销。

附录

main.c - 测试程序

通过源码开头的宏定义可以控制具体运行的算法：

- MULTITHREAD：是否使用并行算法；
- GANULARITY：并行粒度（仅 MULTITHREAD 大于 0 时有效），等于 0 时不启用粒度控制，否则按照申明的值进行控制；
- USE_FUNCTION：在串行算法中使用函数计算斐波那契数列（仅 MULTITHREAD 为 0 时有效）。

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

/**** Parameters ****/

// #define N             70
#define N            70
#define REPEAT        1000          // for parallel case
// #define REPEAT        10000000       // for serial case
#define MULTITHREAD    4
#define GRANULARITY    0
#define USE_FUNCTION   0


typedef int bool;
```

```
/**** Implementations ****/

void fibonacci_serial(int n, double *output)
{
    double a = 1.0, a_1 = 0.0;
    output[0] = a;
    for (int i = 1; i != n; ++i) {
        a += a_1;
        a_1 = a - a_1;
        output[i] = a;
    }
}

void *_fibonacci_parallel_job(void *arg)
{
    double *inout = (double *)arg;
    *inout = round(
        ( pow(1.6180339887498949025257388711906969547271728515625,
*inout)
          - pow(-0.6180339887498949025257388711906969547271728515625,
*inout)
        ) / 2.2360679774997898050514777423813939094543457031 25
    );
    return NULL;
}

void fibonacci_serial_func(int n, double *output)
{
    for (int i = 0; i != n; ++i) {
        output[i] = i + 1;
        _fibonacci_parallel_job((void *)&output[i]);
    }
}

void fibonacci_parallel_granctrl(int n, double *output)
{
    pthread_t *pool = (pthread_t *)malloc(n * sizeof(pthread_t));
    if (pool == NULL) {
        return;
    }
    for (int i = 0; i < n; i += GRANULARITY) {
        for (int j = i; j != i + GRANULARITY && j != n; ++j) {
            output[j] = j + 1;
```

```
            if (pthread_create(&pool[j], NULL, _fibonacci_parallel_job,
(void *)&output[j])) {
                free(pool);
                return;
            }
        }
        for (int j = i; j != i + GRANULARITY && j != n; ++j) {
            pthread_join(pool[j], NULL);
        }
    }
    free(pool);
}

void fibonacci_parallel_naiive(int n, double *output)
{
    pthread_t *pool = (pthread_t *)malloc(n * sizeof(pthread_t));
    if (pool == NULL) {
        return;
    }
    for (int i = 0; i != n; ++i) {
        output[i] = i + 1;
        if (pthread_create(&pool[i], NULL, _fibonacci_parallel_job,
(void *)&output[i])) {
            free(pool);
            return;
        }
    }
    for (int i = 0; i != n; ++i) {
        pthread_join(pool[i], NULL);
    }
    free(pool);
}

/**** Test Runtime ****/

int main(int argc, const char **argv)
{
    int n = N;
    double *out = (double *)malloc(n * sizeof(double));
    if (out == NULL) {
        return -1;
    }

    // Choosing method
```

```
    void (*func)(int, double *) = NULL;
    if (MULTITHREAD) {
        if (GRANULARITY <= 0) { func = fibonacci_parallel_naiive; }
        else { func = fibonacci_parallel_granctrl; }
    } else {
        if (!USE_FUNCTION) { func = fibonacci_serial; }
        else { func = fibonacci_serial_func; }
    }

    // Run
    for (int i = 0; i != REPEAT; ++i) {
        func(n, out);
    }
    // Print to console
    for (int i = 0; i != n; ++i) {
        printf("fib(%d) = %.0lf\n", i, out[i]);
    }
    free(out);
    return 0;
}
```

run.sh - 测试脚本

```
#!/bin/bash

gcc -o fib main.c -lm -pthread
/usr/bin/time -v ./fib
rm ./fib
```