斐波那契数列计算并行优化方案设计

朱晓光　计算机科学与技术（校际交流）1601 班　U201610136

项目目标

1. 分析串行算法的复杂度；
2. 设计并行算法并验证其正确性；
3. 分析大数场景下并行算法的加速比；
4. 优化大数场景下并行算法。

问题分析与假设

在本次实训中，我分别采用斐波那契数列递推公式（见公式 1）与其函数（见公式 2），分别完成了串行环境与并行环境下的斐波那契数列的计算。

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

公式 1 斐波那契数列递推公式

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

公式 2 斐波那契数列函数

若设所求斐波那契数列长度为 n，则应有串行算法时间复杂度为 O(n)，并行算法复杂度为 O(1)（仅考虑计算时间，不考虑线程初始化、共享空间分配与输出步骤）。但需要注意的是，由于并行算法引入了乘、除与乘方运算，仅有当 n 足够大时，并行算法的时间才可能优于串行算法。

具体设计与实现

为方便实验，这里在本地在 C 语言环境下实现了使用递推公式以及函数的串行算法，并使用 pthread 重新实现了使用函数的并行算法。

由于使用在使用函数时涉及浮点数运算，当输入序号过大时，函数结果不再准确。因此实验中仅取 n=70。

实验中发现，由于操作系统在线程管理上耗费了大量时间，并行算法效率极低（实际上由于线程管理耗时远大于计算耗时，并行算法总体时间复杂度更接近 O(n)），于是对并行算法加入粒度控制以进行改进。具体做法为每一批（约 4-8 个，视硬件情况而定）进行一次并行运算。

结果比较与分析

本次实验使用的硬件软件环境如下：

- 系统：Ubuntu 16.04 xenial（WSL 环境）
- 内核版本：x86_64 Linux 4.4.0-18362-Microsoft
- CPU 型号：Intel Core i5-8300H CPU @ 2.301GHz
- 内存大小：16220MiB
- GCC 版本：5.4.0 20160609

图 1 展示了使用递推公式的串行算法的时间复杂度。由于程序运行速度过快，图中“程序运行时间”为运行 1E7 次算法的总时间。

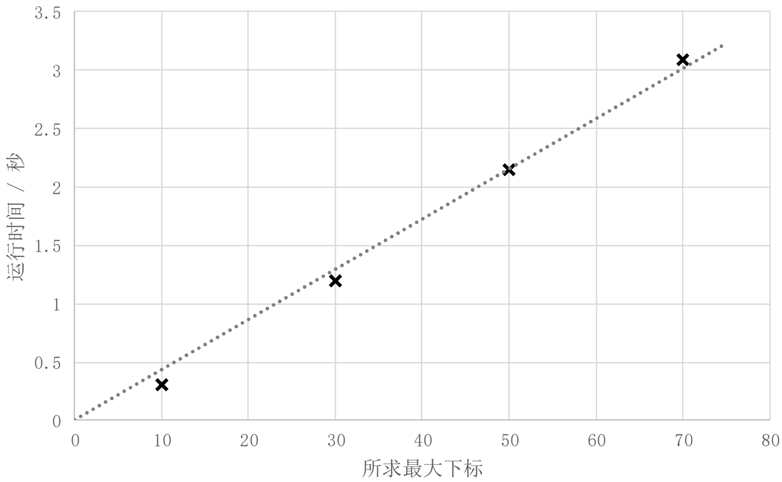

图 1 使用递推公式的串行算法的所求下标-运行时间图

可见，使用递推公式的串行算法的时间复杂度为线性时间复杂度，与假设相符。

图 2 展示了使用函数的未经优化的并行算法的时间复杂度。这里取“程序运行时间”为运行 1E3 次算法的总时间。

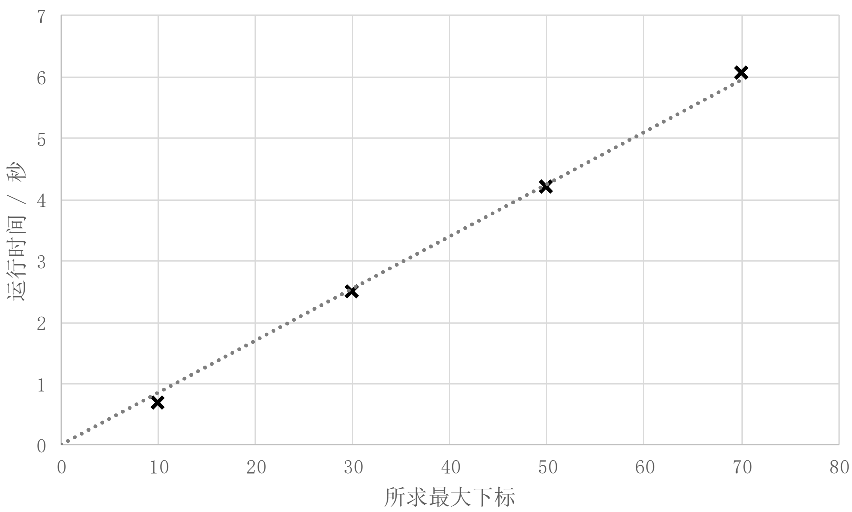

图 2 使用函数的未经优化的并行算法的所求下标-运行时间图

可以看到，并行算法的时间复杂度仍呈现 O(n)的趋势，且比串行算法要慢得多。这可能是由于系统在线程管理上耗费了大量时间，而线程的创建与回收在主线程中仍然是串行执行的，因此总时间复杂度为 O(n)。此外，由于测试机器仅有有限的物理核心可供使用（测试机器为 4 核 8 线程），而 n 又远大于这个数字，因此经过调度后可近似认为计算是一批一批并发执行的，但是在批内是并行的。

图 3 展示了几种方案的时间复杂度（部分数据因打印原因不便查看，可以参考附录中给出的原始数据）。

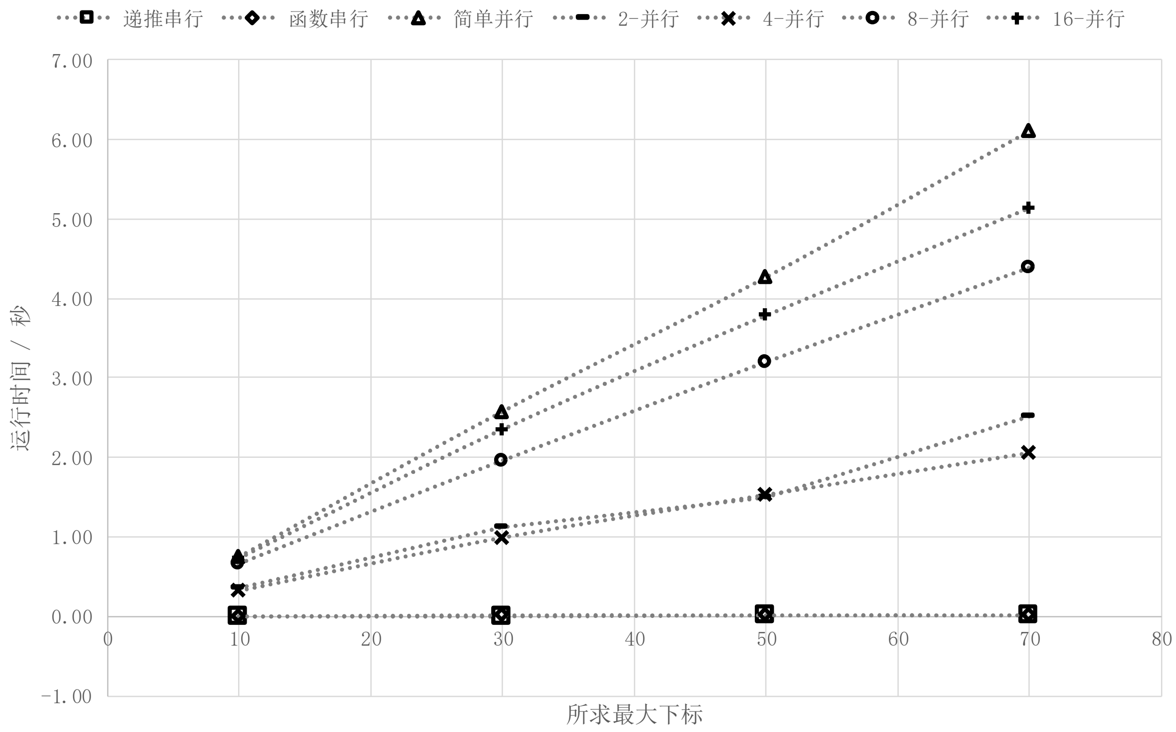

图 3 几种方案的时间复杂度

上图清楚地显示，当并行粒度超过一定阈值后，并行粒度越大算法效率越差，最后趋近无粒度控制的“简单并行”情况。

表 1 展示了以上几种方案的运行时间分配（n=70，除串行测 1E7 次运行总时间，其余均测 1000 次运行总时间）。

表 1 几种方案的运行时间分配

	用户时间	系统时间	系统-用户比值	CPU利用率
递推串行（1E7）	3.07	0.00	0.00	99%
简单并行	0.48	6.60	13.75	116%
4-并行	0.43	6.60	15.35	114%

可以看到，并行算法显著地增加了系统时间的占比，且其 CPU 利用率仅略微高出 100%。这可能说明斐波那契数列这一研究案例的运算量不够大，线程在刚被创建后即运行结束，不能够很好地使各个线程的运行时间重叠起来，以达到较高的运行时并行度。

思考总结

通过本次对斐波那契数列计算并行优化方案的研究，我了解到在实际运行环境中，可能有多方因素影响到并行算法所能提供的加速比，甚至有可能因每个线程/进程中的工作量太少，而产生小于 1 的加速比。因此，在设计并行算法时，需要充分考虑到实际运行任务的特性，合理地设置并行粒度，甚至在必要的时候回归到串行算法。

附录

图 3 原始数据

所求下标	递推串行	函数串行	简单并行	2-并行	4-并行	8-并行	16-并行
10	0.00	0.00	0.74	0.36	0.32	0.65	0.73
30	0.00	0.01	2.57	1.11	0.99	1.95	2.35
50	0.01	0.01	4.26	1.49	1.53	3.19	3.78
70	0.01	0.01	6.10	2.51	2.06	4.38	5.13

其中运行时间的单位均为秒每 1000 次。

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    0
#define GRANULARITY    4
#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)
        ) / 2.236067977499789805051477742381393909454345703125
    );
    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;
    }
```

```c
    // 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

```bash
#!/bin/bash

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