大作业 多线程编程

# 一、目的

熟悉 Linux 系统线程相关的系统调用。

# 二、内容

多线程方法计算浮点向量的点积。

# 三、实验要求

*N*

1. 在 Linux 平台上用多线程方法实现浮点向量的点积计算： *s*   *aibi* 。
2. 程序的输入命令行：

vec\_mul *thread\_num N*

参数：

*thread\_num*：线程数，从 1 到 16

*N*：向量长度，不少于 100,000

1. 输出格式：

*s=xxxx t=tttt(ms)*

说明：

xxxx: 点积计算结果

tttt: 计算所需要时间（不包括向量初始化的时间），以毫秒为单位

# 四、实验结果

|  |  |  |  |
| --- | --- | --- | --- |
| Thread\_num | N | 计算结果 | 计算时间 |
| 1 | 100,000 | 66667.0 | 2.7ms |
| 1 | 1,000,000 | 666667.0 | 11.4ms |
| 1 | 10,000,000 | 6666667.0 | 90.3ms |
| 1 | 100,000,000 | 66666667.0 | 945.7ms |
| 2 | 100,000,000 | 66666667.0 | 493.4ms |
| 4 | 100,000,000 | 66666667.0 | 280.8ms |
| 8 | 100,000,000 | 66666667.0 | 154.3ms |
| 16 | 100,000,000 | 66666667.0 | 176.8ms |

# 

# 五、所用函数

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <sys/time.h>

// 全局变量：向量、长度、线程数、结果分段存储

double \*a, \*b;

int N, thread\_num;

double \*partial\_sums; // 各线程计算的部分和

// 线程参数结构体

typedef struct {

    int start;

    int end;

    int thread\_id;

} ThreadArgs;

// 线程函数：计算指定区间的点积

void\* calculate\_partial\_sum(void \*args) {

    ThreadArgs \*t\_args = (ThreadArgs \*)args;

    double sum = 0.0;

    // 按规则计算区间内的点积

    for (int i = t\_args->start; i < t\_args->end; i++) {

        int mod = i % 3;

        double ai, bi;

        if (mod == 0) {

            ai = bi = 1.0;

        } else if (mod == 1) {

            ai = bi = -1.0;

        } else { // mod == 2

            ai = bi = 0.0;

        }

        sum += ai \* bi;

    }

    partial\_sums[t\_args->thread\_id] = sum; // 存储部分结果

    return NULL;

}

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

    if (argc != 3) {

        fprintf(stderr, "Usage: %s thread\_num N\n", argv[0]);

        return 1;

    }

    thread\_num = atoi(argv[1]);

    N = atoi(argv[2]);

    // 初始化线程相关变量

    pthread\_t \*threads = (pthread\_t \*)malloc(thread\_num \* sizeof(pthread\_t));

    partial\_sums = (double \*)calloc(thread\_num, sizeof(double));

    ThreadArgs \*args = (ThreadArgs \*)malloc(thread\_num \* sizeof(ThreadArgs));

    // 计算每个线程的任务区间

    int block\_size = N / thread\_num;

    int remainder = N % thread\_num;

    struct timeval start, end;

    gettimeofday(&start, NULL); // 开始计时（仅计算时间，不含初始化）

    // 创建线程并分配任务

    for (int i = 0; i < thread\_num; i++) {

        args[i].start = i \* block\_size;

        args[i].end = (i == thread\_num - 1) ?

                      args[i].start + block\_size + remainder :

                      args[i].start + block\_size;

        args[i].thread\_id = i;

        if (pthread\_create(&threads[i], NULL, calculate\_partial\_sum, &args[i]) != 0) {

            perror("pthread\_create failed");

            return 1;

        }

    }

    // 等待所有线程完成

    for (int i = 0; i < thread\_num; i++) {

        pthread\_join(threads[i], NULL);

    }

    // 合并部分结果

    double total\_sum = 0.0;

    for (int i = 0; i < thread\_num; i++) {

        total\_sum += partial\_sums[i];

    }

    gettimeofday(&end, NULL); // 结束计时

    // 计算时间差（毫秒）

    long time\_us = (end.tv\_sec - start.tv\_sec) \* 1000000 + (end.tv\_usec - start.tv\_usec);

    double time\_ms = (double)time\_us / 1000;

    // 输出结果

    printf("s=%.1f t=%.1f(ms)\n", total\_sum, time\_ms);

    // 释放内存

    free(threads);

    free(partial\_sums);

    free(args);

    return 0;

}

# 六、绘制图像

