中山大學
SUN YAT-SEN UNIVERSITY

并行程序设计与算法实验

Lab4-Pthreads **并行方程求解与蒙特卡洛**

姓 名 马岱

学 号 22336180

学 院 计算机学院

专 业 计算机科学与技术

2025 年 4 月 16 日

1 实验目的

- 深入理解 Pthreads 同步机制：条件变量、互斥锁
- 评估多线程加速性能

2 实验内容

- 多线程一元二次方程求解
- 多线程圆周率计算

2.1 方程求解

- 使用 Pthread 多线程及求根公式实现并行一元二次方程求解。
- 方程形式为 $x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$，其中判别式的计算、求根公式的中间值分别由不同的线程完成。
- 通过条件变量识别何时线程完成了所需计算，并分析计算任务依赖关系及程序并行性能。

2.1.1 代码思路

在 ppt 中只将求解判别式和计算 x1、x2 结果分开来计算了，但是仅仅这样划分并没有完全应用并行的思想，只是在计算最终结果的时候使用到了条件变量，因此为了能够最大化并行中间值的计算，这里我共建立了 5 个线程，分别用来计算 b^2、$4ac$、$2a$、$sqrt(b^2-4ac)$、$x1$ $x2$。之所以选择这样划分，是因为我考虑到在计算乘法的时候要花费更多的时间（相较于加法），而对于 b^2、$4ac$、$2a$ 这几个值计算的时候，他们可以完全独立计算（并行计算），$sqrt(b^2-4ac)$、$x1$ $x2$ 这几个值则需要依靠前面几个值的结果，因此需要额外加上同步条件。也就是说在计算 $sqrt(b^2-4ac)$、$x1$ $x2$ 之前我们需要先把 b^2、$4ac$、$2a$ 这几个值都计算出来，因此需要一个统一的阻塞，在他们各自的结果计算出来之后一直处于阻塞状态直到其三个的结果都计算出来，这里就使用了条件变量来进行控制。

2.1.2 代码解释

首先我们需要定义全局变量来控制各个线程，这里我没有使用结构体来统一传输的原因是对结果影响不是很大，因为本身运行时间就已经很小了，再加上结构体来传输数据很有可能 **“反其道而行之”**。

```c
#define THREAD_NUM 5 // 线程数
double a, b, c;          // 系数
double b_2, _4ac, two_a; // b^2 和 4ac 和 2a
double d;                // 判别式
double sqrtd;            // 平方根
double x1, x2;           // 根
int counter = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // 互斥锁
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;    // 条件变量
```

接着我们来实现多线程中最为重要的函数，$thread\_func()$ 函数，根据我前面叙述的代码思路，我们共定义 5 个线程，其中前面三个线程可以并行运行，三个线程分别用来计算 b^2、$4ac$、$2a$，当各自线程计算完后阻塞等待，直到 $counter = 3$（运行到第四个线程）再一起将他们传播出去（使用 $pthread\_cond\_broadcast(\&cond)$）

```c
    int id = *(int *)arg; // 线程 ID

    if (id == 0)
    {
        // 线程 0 计算 b^2
        double temp = b * b;
        pthread_mutex_lock(&mutex);
        b_2 = temp;
        counter++; // 完成一项计算
        if (counter == 3)
        { // 如果 b^2, 4ac, 2a 均计算完毕
            pthread_cond_broadcast(&cond);
        }
        pthread_mutex_unlock(&mutex);
        printf("Thread %d: b^2 = %.2f\n", id, b_2);
    }
    else if (id == 1)
    {
        // 线程 1 计算 4ac
        double temp = 4 * a * c;
        pthread_mutex_lock(&mutex);
        _4ac = temp;
        counter++;
        if (counter == 3)
        {
```

```
            pthread_cond_broadcast(&cond);
        }
        pthread_mutex_unlock(&mutex);
        printf("Thread %d: 4ac = %.2f\n", id, _4ac);
    }
    else if (id == 2)
    {
        // 线程 2 计算 2a
        double temp = 2 * a;
        pthread_mutex_lock(&mutex);
        two_a = temp;
        counter++;
        if (counter == 3)
        {
            pthread_cond_broadcast(&cond);
        }
        pthread_mutex_unlock(&mutex);
        printf("Thread %d: 2a = %.2f\n", id, two_a);
    }
```

注意 *counter* 变量在这里起到了很重要的传递（阻塞）作用（可以理解为和条件变量绑定在一起的互斥量）

当这三个线程都执行完成之后，就可以计算 $sqrt(b^2-4ac)$、$x1$ $x2$ 了，这里也没有过多需要强调的，注意一下条件变量的使用就好了。

```
    else if (id == 3)
    {
        // 线程 3 等待前三个计算完成，计算判别式 d
        pthread_mutex_lock(&mutex);
        while (counter < 3)
        {
            pthread_cond_wait(&cond, &mutex);
        }
        pthread_mutex_unlock(&mutex);
        d = b_2 - _4ac;
        printf("Thread %d: d = %.2f\n", id, d);
        // 将计数器更新为 4，表示 d 已经计算完毕，并唤醒线程 4
        pthread_mutex_lock(&mutex);
        counter++;
```

```
        if (counter == 4)
            pthread_cond_broadcast(&cond);
        pthread_mutex_unlock(&mutex);
    }
    else if (id == 4)
    {
        // 线程 4 等待判别式 d 计算完成后，计算根并输出
        pthread_mutex_lock(&mutex);
        while (counter < 4)
        {
            pthread_cond_wait(&cond, &mutex);
        }
        pthread_mutex_unlock(&mutex);
        if (d >= 0)
        {
            sqrtd = sqrt(d);
            x1 = (-b + sqrtd) / two_a;
            x2 = (-b - sqrtd) / two_a;
            printf("Thread %d: x1 = %.2f, x2 = %.2f\n", id, x1, x2);
        }
        else
        {
            printf("Thread %d: d < 0, no real roots.\n", id);
        }
    }
```

2.1.3 流水线式处理

在运行完代码之后我发现了一个很奇怪的现象，那就是使用多线程计算还不如单线程计算运行的快。仔细想一下其实确实是这样的，如果仅仅计算单个方程的话那这个并行就毫无意义，那么一个自然而然的想法就是把这个拓宽到一次性处理多个方程，简称**“流水线式批量处理模式”**。

那么该流水线具体怎么实现呢，其实总共可以分为三个阶段，我们将多组方程拆成三个阶段并行执行。对于“$ax^2 + bx + c = 0$”：

- Stage 1：计算 b^2, $4ac$, $2a$。
- Stage 2：等待 Stage 1 输出后计算判别式 $d = b^2 - 4ac$。
- Stage 3：等待 Stage 2 输出后计算 $\sqrt{d}$ 并最终求根。

对 M 个方程，我们给每个阶段配备一组线程和一个环形缓冲区（队列）。Stage 1 不断从输入队列取下一个方程，计算完中间量后放入 Stage 2 的队列；Stage 2 从自己的队列取出、中间量计算判别式并放入 Stage 3；Stage 3 取出判别式计算根并输出。这种模式能让各阶段常驻多线程，各自并行处理不同方程的不同阶段。

首先我们定义需要使用的结构体：

```
// 流水线参数
int M; // 方程总数
int num_threads; // 总线程数
int t1, t2, t3; // 三个阶段的线程数

// 任务结构体
typedef struct {
    double a, b, c;
    double b2, fourac, twoa; // 中间量 b², 4ac, 2a
    double d;   // 判别式 d = b² - 4ac
} Task;

// 环形队列
typedef struct {
    Task **buf; // 存放 Task* 的数组
    int head, tail, cap; // 头尾索引和容量
    pthread_mutex_t m;
    pthread_cond_t not_empty; // 非空条件
    pthread_cond_t not_full;  // 非满条件
} RingQueue;

static RingQueue Q1, Q2, Q3;  // 三段流水线对应的队列
```

接着我们需要实现缓冲区，这里我实现了**生产者-消费者缓冲区**，其中 push 函数向队列中插入一个任务（Task），或者传入 NULL 表示输入结束标志，pop 函数从队列中取出一个任务，如果当前是空的就阻塞等待。

```
// 向队列尾部推入元素（或 NULL 作为结束标志）
void rq_push(RingQueue *q, Task *t) {
    pthread_mutex_lock(&q->m);
    while ((q->tail + 1) % q->cap == q->head)
        pthread_cond_wait(&q->not_full, &q->m);
    q->buf[q->tail] = t;
    q->tail = (q->tail + 1) % q->cap;
```

```
    pthread_cond_signal(&q->not_empty);
    pthread_mutex_unlock(&q->m);
}

// 从队列头部弹出元素（阻塞直到非空）
Task *rq_pop(RingQueue *q) {
    pthread_mutex_lock(&q->m);
    while (q->head == q->tail)
        pthread_cond_wait(&q->not_empty, &q->m);
    Task *t = q->buf[q->head];
    q->head = (q->head + 1) % q->cap;
    pthread_cond_signal(&q->not_full);
    pthread_mutex_unlock(&q->m);
    return t;
}
```

接着分三个阶段来实现具体的运算，具体思路和之前是一样的，但是这里注意是将结果存储到了缓冲区。

```
// Stage1: 计算 b², 4ac, 2a
void *stage1(void *arg)
{
    while (1)
    {
        Task *t = rq_pop(&Q1);
        if (t == NULL)
            break; // NULL 标志结束
        t->b2 = t->b * t->b;
        t->fourac = 4 * t->a * t->c;
        t->twoa = 2 * t->a;
        rq_push(&Q2, t);
    }
    return NULL;
}

// Stage2: 计算判别式 d = b² - 4ac
void *stage2(void *arg)
{
    while (1)
    {
```

```
        Task *t = rq_pop(&Q2);
        if (t == NULL)
            break;
        t->d = t->b2 - t->fourac;
        rq_push(&Q3, t);
    }
    return NULL;
}

// Stage3: 计算根并打印
void *stage3(void *arg)
{
    while (1)
    {
        Task *t = rq_pop(&Q3);
        if (t == NULL)
            break;
        if (t->d >= 0)
        {
            double sd = sqrt(t->d);
            double x1 = (-t->b + sd) / t->twoa;
            double x2 = (-t->b - sd) / t->twoa;
            printf("Eq(%.2f,%.2f,%.2f) → x1=%.5f, x2=%.5f\n",
                   t->a, t->b, t->c, x1, x2);
        }
        else
        {
            printf("Eq(%.2f,%.2f,%.2f) → 无实根\n",
                   t->a, t->b, t->c);
        }
        free(t);
    }
    return NULL;
}
```

最后还有一个单线程，鉴于篇幅过长这里就不放上来代码解释了，和之前的大差不差。

2.2 蒙特卡洛求圆周率 pi 的近似值

- 使用 Pthread 创建多线程，并行生成正方形内的 n 个随机点。
- 统计落在正方形内切圆内点数，估算 π 的值。
- 设置线程数量（1-16）及随机点数量（1024-65536），观察对近似精度及程序并行性能的影响。

2.2.1 代码思路

蒙特卡洛这个实验其实就没有什么太多可说的了，就是很普通的划分点集，然后均分给线程执行。

2.2.2 代码解释

这里直接定义线程函数，将数据平均划分，然后每个线程执行尽量相同数量的点的估计，最后我们再做一个汇总。

```
void *monte_carlo(void *arg)
{
    int thread_id = *(int *)arg;
    long long inside = 0;
    unsigned int seed = time(NULL) + thread_id;
    for (int i = 0; i < num_points / num_threads; i++)
    {
        double x = (double)rand_r(&seed) / RAND_MAX;
        double y = (double)rand_r(&seed) / RAND_MAX;
        if (sqrt(x * x + y * y) <= 1.0)
        {
            inside++;
        }
    }
    pthread_mutex_lock(&mutex);
    total_inside += inside;
    pthread_mutex_unlock(&mutex);
    printf("Thread %d: Inside points = %lld\n", thread_id, inside);
    return NULL;
}
```

注意这里我是在线程函数里面通过上锁直接完成总的点数的相加，而不是等到回到主线程等待调用 join 之后再完成相加。这样一来便能节省不少时间。

2.3 运行脚本

为了方便运行，这里为两个任务都写了一个 shell 自动化运行脚本，这里只展示第一个任务的流水线运行脚本。

```
#!/bin/bash
# 自动化测试随机系数流水线批量处理

source="eq_slow.c"
exe="eq_slow"

# 编译
gcc -O2 -pthread -o $exe $source -lm
if [ $? -ne 0 ]; then
    echo " 编译失败，请检查 $source"
    exit 1
fi

# 测试配置
M_values=(1000 5000 1000000)
thread_counts=(3 6 9 12)

log="eq_slow_tests.log"
echo " 随机系数流水线测试" > $log
echo " 日期: $(date)" >> $log
echo "------------------------" >> $log

for M in "${M_values[@]}"; do
  echo " 方程数量: $M" | tee -a $log
  for t in "${thread_counts[@]}"; do
    echo "  线程数: $t" | tee -a $log
    result=$(.\/$exe $M $t)
    echo "$result" | tee -a $log
    echo "----------------" >> $log
  done
  echo "========================" | tee -a $log
done

echo " 测试完成，详细日志: $log"
```

3 实验结果与分析

3.1 方程求解

```
kdaier@kdaier-VirtualBox:~/lab4$ chmod +x single_test.sh
kdaier@kdaier-VirtualBox:~/lab4$ ./single_test.sh
测试方程数量: 500000
单线程运行时间 0.01380 s
测试方程数量: 1000000
单线程运行时间 0.02612 s
测试方程数量: 2000000
单线程运行时间 0.04334 s
```

图 1: 单线程运行时间

```
kdaier@kdaier-VirtualBox:~/lab4$ ./test_slow.sh
方程数量: 500000
  线程数: 3
使用 3 线程处理 500000 个方程，耗时 0.02981 秒
  线程数: 6
使用 6 线程处理 500000 个方程，耗时 0.01928 秒
  线程数: 9
使用 9 线程处理 500000 个方程，耗时 0.00702 秒
  线程数: 12
使用 12 线程处理 500000 个方程，耗时 0.03759 秒
========================
方程数量: 1000000
  线程数: 3
使用 3 线程处理 1000000 个方程，耗时 0.06083 秒
  线程数: 6
使用 6 线程处理 1000000 个方程，耗时 0.05125 秒
  线程数: 9
使用 9 线程处理 1000000 个方程，耗时 0.01907 秒
  线程数: 12
使用 12 线程处理 1000000 个方程，耗时 0.00170 秒
========================
方程数量: 2000000
  线程数: 3
使用 3 线程处理 2000000 个方程，耗时 0.11785 秒
  线程数: 6
使用 6 线程处理 2000000 个方程，耗时 0.15403 秒
  线程数: 9
使用 9 线程处理 2000000 个方程，耗时 0.00187 秒
  线程数: 12
使用 12 线程处理 2000000 个方程，耗时 0.00749 秒
```

图 2: 多线程流水线运行时间

- 分析不同线程配置下的求解时间，评估并行化带来的性能提升。

  这里由于我提前运行了单个线程和多个线程对单个方程的影响，发现了并行程序如果只针对单个方程的就没有任何意义了，因此我采用了“流水线模式”，设计了生产者-消费者缓冲区来完成对多个方程的共同求解。可以看到上图的实验结果，当我把方程的数量增加到 100w 及以上的时候，这是并行的优势就完全体现出来了。可以看到再方程数量为 50w 的时候，单线程的运行时间还是低于各多线程的运行时间的，但是当我将数据量增大到 100w 时，9 线程和 12 线程的运行时间就已经低于单线程了，直到这时，多线程的优势才真正发挥出来！这也符合并行计

算中的典型特点——在数据量较小的情况下，由于线程创建、上下文切换、同步控制等额外开销的存在，多线程程序可能反而比单线程程序更慢；但当任务量足够大时，这些开销就会被大量计算所掩盖，从而实现真正的加速效果。

- 对比单线程与多线程方案在处理相同方程时的表现，讨论可能存在的瓶颈或优化空间。

  对比单线程与多线程方案在处理相同方程时的表现，我们可以明显观察到两者在处理效率上的差异：在方程数量较小时，单线程由于结构简单资源调度开销小，表现更优；而在大规模方程求解中多线程则通过任务分摊显著缩短了总运行时间。

  不过，也存在一些需要注意的点。首先是负载不均衡的问题，若各线程处理的数据量差异大，会出现某些线程忙碌而其他线程空闲的情况，影响整体效率；其次是线程同步与共享数据访问所导致的阻塞开销，在访问共享缓冲区或结果容器时导致频繁发生锁竞争；此外，线程数设置不合理（超过 CPU 核数）会造成频繁的上下文切换，反而拖慢整体进程。

  因此，为进一步优化程序，可采用动态任务分配以提升负载均衡性，使用无锁队列或最小化锁定粒度降低同步开销，引入线程池管理复用线程资源。

3.2 蒙特卡洛方法求圆周率

表 1: 蒙特卡洛　估计运行时间

采样点数	1 线程	2 线程	4 线程	8 线程	16 线程
1024	0.0013	0.0009	0.0024	0.0057	0.0078
4096	0.0013	0.0007	0.0006	0.0016	0.0009
16384	0.0013	0.0009	0.0026	0.0014	0.0030
65536	0.0026	0.0079	0.0030	0.0023	0.0023

表 2: 蒙特卡洛　估计值

采样点数	1 线程	2 线程	4 线程	8 线程	16 线程
1024	3.125000	3.140625	3.195313	3.046875	3.070313
4096	3.154297	3.173828	3.158203	3.154785	3.157715
16384	3.142944	3.141846	3.141113	3.142151	3.142456
65536	3.141205	3.141326	3.141724	3.141357	3.141541

表 3: 蒙特卡洛　估计并行效率

采样点数	1 线程	2 线程	4 线程	8 线程	16 线程
1024	1.0000	1.4643	0.5325	0.2265	0.1667
4096	1.0000	1.9362	2.0682	0.7913	1.4219
16384	1.0000	1.4000	0.5045	0.9492	0.4375
65536	1.0000	0.3278	0.8676	1.1132	1.1346

- 比较不同线程数量和随机点数量下圆周率估计的准确性和计算速度。

  首先从估计准确性的角度来看，随着随机点数量的增加，pi 的估计值逐渐趋近于真实值 3.1415926，误差不断减小。这是蒙特卡洛方法收敛性的体现：样本量越大，统计误差越小，估计结果越接近真实值。同时我们也观察到，线程数量的变化对估计值的波动影响并不大，准确性主要受随机点数量支配，这说明并行化处理虽然加快了计算速度，但在估计值的统计性质上仍保持一致性。

  再从计算速度来看，线程数量对运行时间有显著影响。在随机点数量较小时，线程数增加带来的加速效果并不明显，甚至由于线程调度和同步开销，可能出现运行时间增加的情况。但当随机点数量足够大时，线程数的增加能有效缩短计算时间，呈现出较为明显的加速比提升。。

- 讨论增加线程数量是否总能提高计算效率，以及其对圆周率估计精度的影响。

  （这一小问和下一小问一起回答，放在下一小问的答案处）

- 提供实验数据图表，展示随着线程数和随机点数的变化，计算效率和精度的趋势。

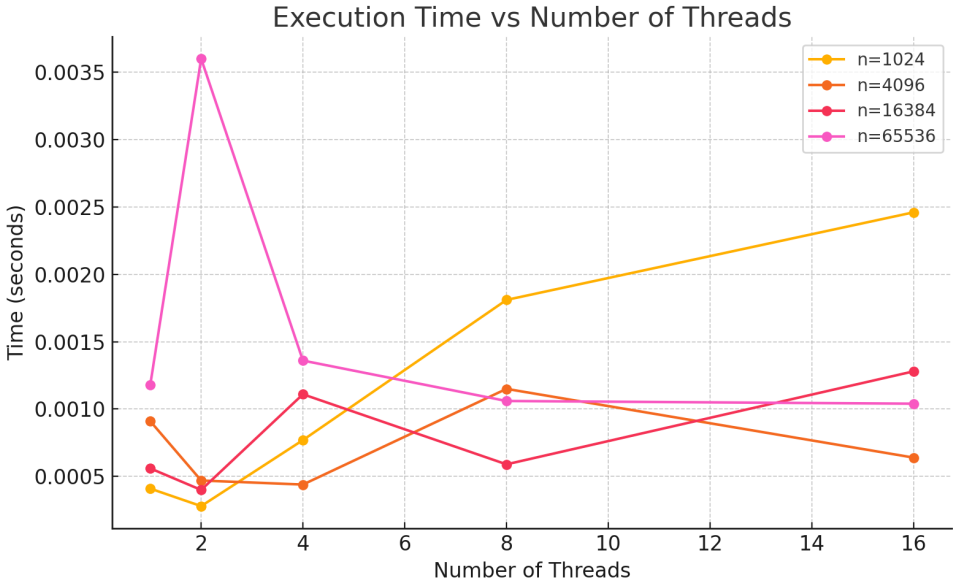

图 3: 线程数-计算效率

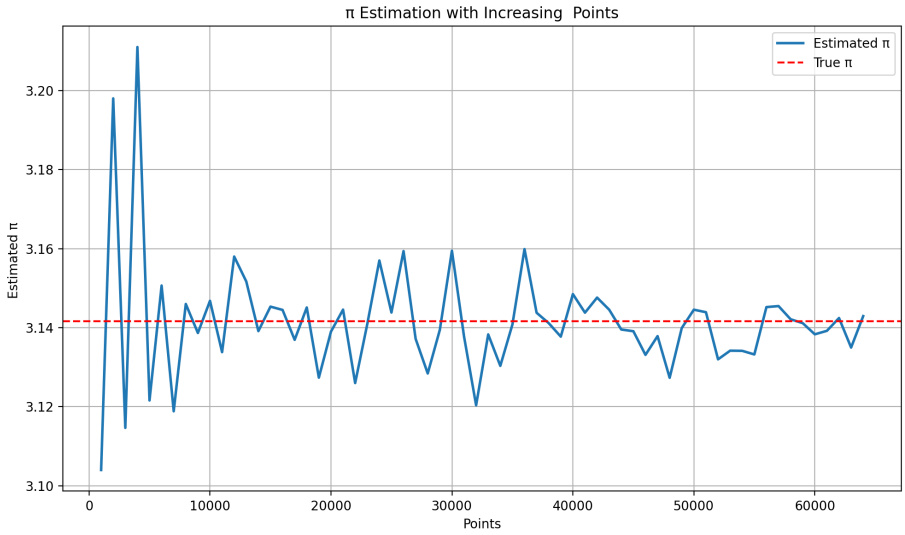

图 4: 随机点数-精度

这里为了能够更直观地观察随着线程数和随机点数的变化，计算效率和精度的趋势。我编辑脚本将数据全部存储在 csv 文件中，然后使用 python 代码来生成可视化的结果图。从图三和图四可以清楚地看到具体的计算效率和精度。

我们可以看到，随着线程数的增加，在固定采样点数的情况下，运行时间整体呈下降趋势，说明多线程确实在一定程度上提升了计算效率。但同时我们也观察到，在较小的采样点数下（如 1024 或 4096），线程数增加带来的时间缩短并不明显，1024 数据甚至存在较大的波动或回升，这主要是由于线程调度、线程创建和合并等额外开销在小规模计算中所占比例较大，影响了整体性能。而在大规模采样点数（如 16384 和 65536）下，线程数增加带来的效率提升更加显著，尤其在 8 线程内效果较好，但超过 8 线程后加速比逐渐趋于平缓，说明出现了并行计算的边际效益递减现象。

从估计 pi 值的变化趋势来看，随着采样点数的增加，pi 的估计值逐渐收敛到真实值 3.1415926 附近；而线程数的变化对估计值的影响并不显著，表明多线程并未改变算法本身的统计性质，仅是提升了样本处理速度。因此提高精度更关键的是足够大的采样点数，而不是过多的线程数。

- 分析实验过程中遇到的问题，如同步问题、负载不均等，并提出相应的解决策略。

  同步问题。由于每个线程需要将自己的局部计数结果汇总到全局变量中，如果不加以控制，会产生 “竞态条件”，导致数据冲突和结果不正确。为了解决这一问题，我在代码中采用了互斥锁机制（‘pthread_mutex_lock‘ 和 ‘pthread_mutex_unlock‘），确保在对全局变量进行写操作时只有一个线程可以访问。

  负载不均问题。理论上将采样点均匀地分配给各线程处理，但在实际运行中，由于线程调度机制、CPU 核心利用率差异或硬件超线程的干扰，部分线程运行时间

可能更长，导致整体运行时间受限于最慢的线程，降低了并行效率。为了解决这一问题，可以引入动态任务分配策略，即将任务划分为更多的小块，由线程动态领取任务执行，从而提升负载均衡性。