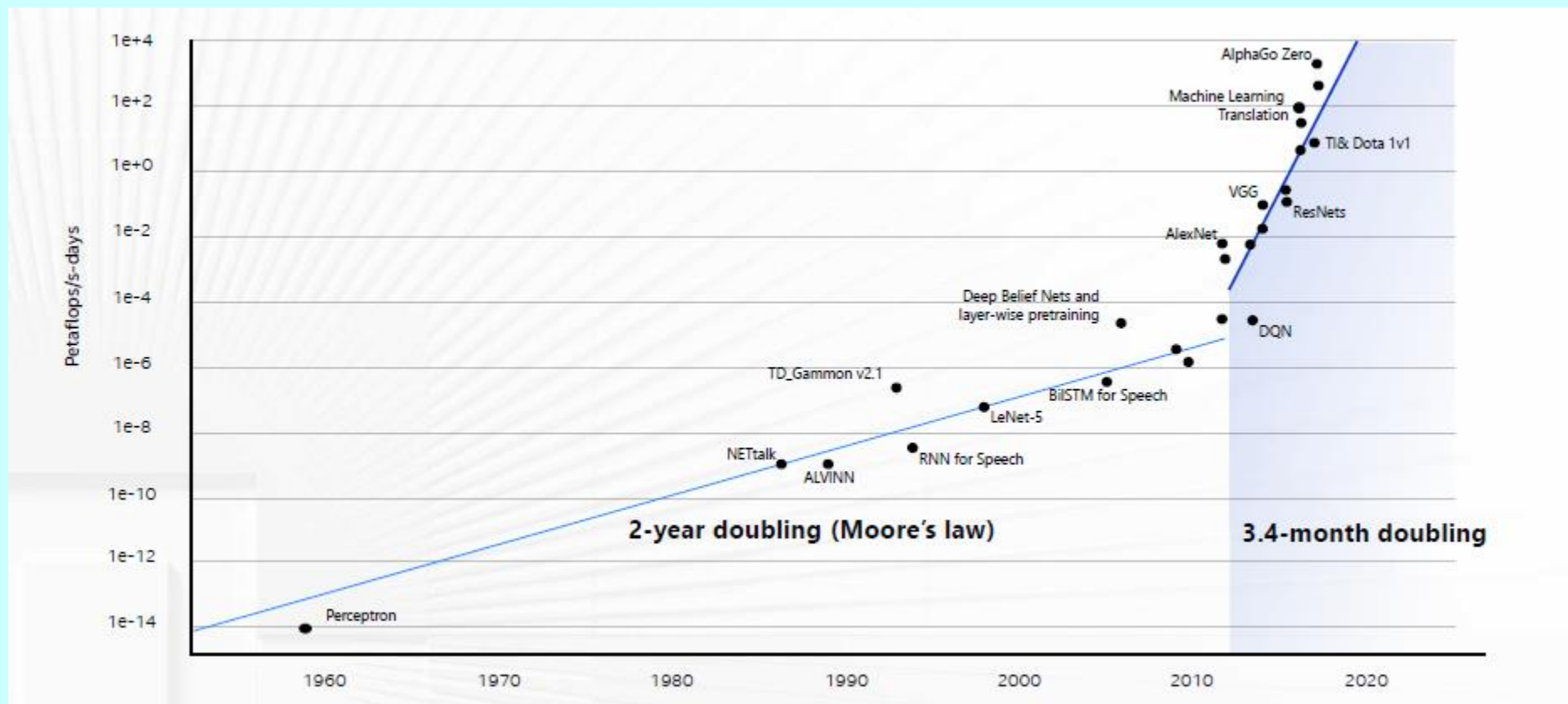


并行编程

软件学院《数据结构》讲义
内部使用



算力瓶颈



Source: AI and Compute • November 7, 2019 • Dario Amodei & Danny Hernandez

动画渲染

```
Frame frames[N]; // 一共有N帧画面

renderFilm():
  for i ← 0 to N-1:
    frames[i].render() // 对每一帧进行渲染
```

以迪士尼的动画电影《汽车总动员2》为例，平均渲染一帧需要11.5小时（最长需要90小时），一共约有152 640帧。如果是串行进行渲染，那么将会需要1 755 360小时（约200年）的时间才能完成。

```
class Frame {  
public:  
    Frame() { flag = false; }  
    void render() { flag = true; }  
    bool isRendered() { return flag; }  
  
private:  
    bool flag;  
};  
  
const int N = 512;  
Frame frames[N];  
  
bool check() {  
    for (int i = 0; i < N; i++) {  
        if (!frames[i].isRendered()) return false;  
    }  
    return true;  
}
```

代码8.1. Frame类、数组及其辅助函数

```
#include <iostream>

using namespace std;

void renderFilm() {
    for (int i = 0; i < N; i++)
        frames[i].render();
}

int main() {
    renderFilm();
    if (check())
        cout << "动画渲染成功\n";
    else
        cout << "动画渲染失败\n";
    return 0;
}
```

代码8.2. 串行的动画渲染

并行动画渲染

```
Frame frames[N]; // 一共有N帧画面
```

```
renderFilm():
```

```
    如果角色是master:
```

```
        将N帧画面等分为M组
```

```
        分配M个slave机器, 每个机器渲染N/M帧
```

```
        等待所有slave机器渲染完毕
```

```
    如果角色是slave:
```

```
        获取机器的id (id假设从0开始到M-1)
```

```
        start ← id * (N/M)
```

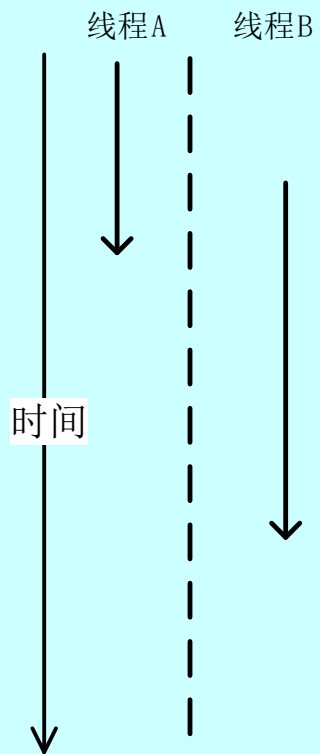
```
        end ← (id+1) * (N/M) // 计算每一台机器渲染的范围[start, end)
```

```
        for i ← start to end-1:
```

```
            frames[i].render() // 对每一帧进行渲染
```

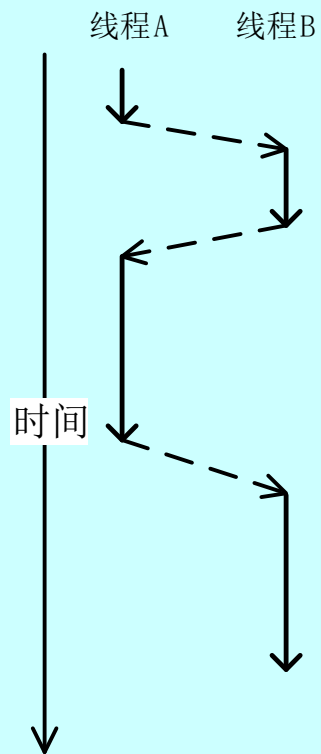
```
        通知master执行完成
```

线程的两种并发模式



(a)

(a) 多个核上的并发（并行）



(b)

(b) 单核上的并发

```
#include <iostream>

#include <thread>           // C++11的多线程标准库

void foo() { std::cout << "thread 1 ...\n"; }

void bar(int x) { std::cout << "thread 2 ... " << x << '\n'; }

int main() {
    std::thread first(foo);    // 创建一个线程first并调用foo()
    std::thread second(bar,0); // 创建一个线程second并调用bar(0)

    std::cout << "thread main ...\n";
    // 线程回收

    first.join();              // 等待first线程运行完
    second.join();             // 等待second线程运行完

    std::cout << "foo and bar completed.\n";
    return 0;
}
```

代码8.3. 简单的C++11 多线程程序

C++ 11 多线程库

```
thread() noexcept; // 1. 默认构造器

template <class Fn, class... Args>
explicit thread (Fn&& fn, Args&&... args); // 2. 初始化构造器

thread (const thread&) = delete; // 3. 复制构造器[删除]

thread (thread&& x) noexcept; // 4. move构造器


void thread::detach();
void thread::join();
bool thread::joinable() const noexcept;
```

(a) 多个核上的并发 (并行) (b) 单核上的并发

```

const int M = 8;    // 线程数

void slaveRenderFilm(int id) {    // 参数是自定义的线程序号

    int start = id * (N / M);

    int end = (id + 1) * (N / M);    // 确定各个线程的负责范围

    for (int i = start; i < end; i++)

        frames[i].render();

    cout << "线程" << id << "完成\n";

}

void renderFilm() {

    vector<thread> threads;

    for (int i = 0; i < M; i++)

        threads.emplace_back(slaveRenderFilm, i);    // 传入线程序号

    for (int i = 0; i < M; i++)

        threads[i].join();

}

```

```

qizhwei@DESKTOP-5VKIBV2:~/test$ ./parallelrender
线程线程0完成
1完成
线程3完成
线程5完成
线程6完成
线程4完成
线程7完成
线程2完成
qizhwei@DESKTOP-5VKIBV2:~/test$ ./parallelrender
线程线程3完成
线程2完成
线程1完成
线程5完成
线程7完成
线程4完成
0完成
线程6完成

```

代码8.4. 并行的动画渲染

数据竞争

```
qizhwei@DESKTOP-5VKIBV2:~/test$ ./datarace
count: 135509
qizhwei@DESKTOP-5VKIBV2:~/test$ ./datarace
count: 130065
qizhwei@DESKTOP-5VKIBV2:~/test$ ./datarace
count: 100019
qizhwei@DESKTOP-5VKIBV2:~/test$ ./datarace
count: 135969
qizhwei@DESKTOP-5VKIBV2:~/test$ ./datarace
count: 138829
qizhwei@DESKTOP-5VKIBV2:~/test$ ./datarace
count: 100053
```

```
#include <iostream>

#include <thread>

using namespace std; // 使用了std名字空间

int countNum = 0; // 全局变量

// 处理100000次
void counter() {
    for(int i = 0; i < 100000; i++)
        countNum++;
}

int main() {
    // 创建两个线程，各自调用counter进行一些处理
    thread t1(counter), t2(counter);

    // 回收
    t1.join();
    t2.join();

    cout << "count: " << countNum << endl;
    return 0;
}
```

代码8.5. 多线程计数程序（共享全局变量）

共享变量存在数据竞争的原因

- 数据竞争产生的原因是由于对于共享变量操作的“非原子性”。非原子性操作是指，这样的操作在执行过程中是可能被其他线程打断的。
 - 比如，在代码8.5中的对共享变量的非原子操作是第10行。当一个线程还没有完成对于数据的操作，另一个线程拿到旧值进行了重复操作，导致结果与预期不符。
- 除非操作本身是原子的（即不可打断的，C++11中也提供<atomic>库，该标准库中提供了一系列原子操作），否则对于共享变量（全局变量、静态变量、共享指针）的操作就会产生数据竞争的现象。
- 此外函数内部的非静态局部变量则不会产生数据竞争现象，因为它们是不共享的。

```

int counts[2];

void counter(int id) {
    static int countNum = 0;          // 静态变量
    for(int i = 0; i < 100000; i++)    // 处理100000次
        countNum++;
    counts[id] = countNum;
}

int main() {
    thread t1(counter, 0), t2(counter, 1);
    t1.join(); t2.join();

    int realCount = (counts[0] > counts[1])? counts[0] : counts[1];
    cout << "count: " << realCount << endl;
    return 0;
}

```

```

count: 146112
count: 165962
count: 144445
count: 164086
.....

```

代码8.6. 多线程计数程序（共享静态变量）

```
// 传入计数器count的指针
void counter(int *cp) {
    for(int i = 0; i < 100000; i++)
        (*cp)++;
}

int main() {
    int countNum = 0;
    thread t1(counter, &countNum), t2(counter, &countNum);
    t1.join(); t2.join();

    cout << "count: " << countNum << endl;
    return 0;
}
```

代码8.7. 多线程计数程序（共享同一个变量的指针）

线程同步-互斥锁

```
// lock() 的操作不可被打断  
void lock() {  
    while(flag) ;  
    flag = 1;  
}
```

```
void unlock() {  
    flag = 0;  
}
```

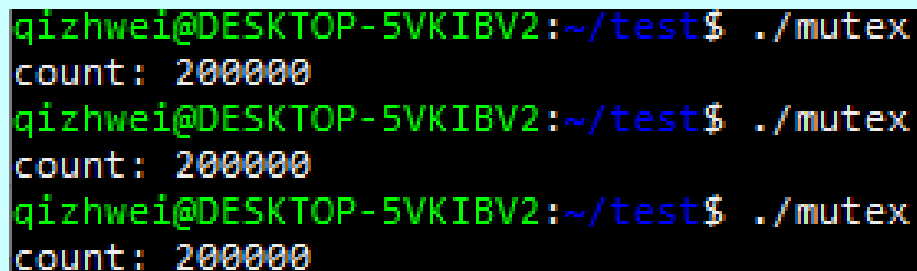
线程同步-互斥锁

```
int countNum = 0;    // 全局变量
mutex mtx;           // 互斥锁，需要include <mutex>

void counter() {
    mtx.lock();

    for(int i = 0; i < 100000; i++)
        countNum++;

    mtx.unlock();
}
```



```
qizhwei@DESKTOP-5VKIBV2:~/test$ ./mutex
count: 200000
qizhwei@DESKTOP-5VKIBV2:~/test$ ./mutex
count: 200000
qizhwei@DESKTOP-5VKIBV2:~/test$ ./mutex
count: 200000
```

代码8.8. 加入mutex互斥锁的计数程序

线程同步-细粒度互斥锁

```
void counter() {  
    for(int i = 0; i < 100000; i++) {  
        mtx.lock();  
        countNum++;  
        mtx.unlock();  
    }  
}
```

代码8.9. 细粒度mutex互斥锁的计数程序

加锁对于程序性能的影响

```
for(int i = 0; i < nthreads; i++)
    threads.emplace_back(sum_mutex, i);

for(int i = 0; i < nthreads; i++)
    threads[i].join();

if(gsum != (nelems * (nelems - 1)) / 2)
    cerr << "Error result " << gsum << endl;

return 0;
}
```

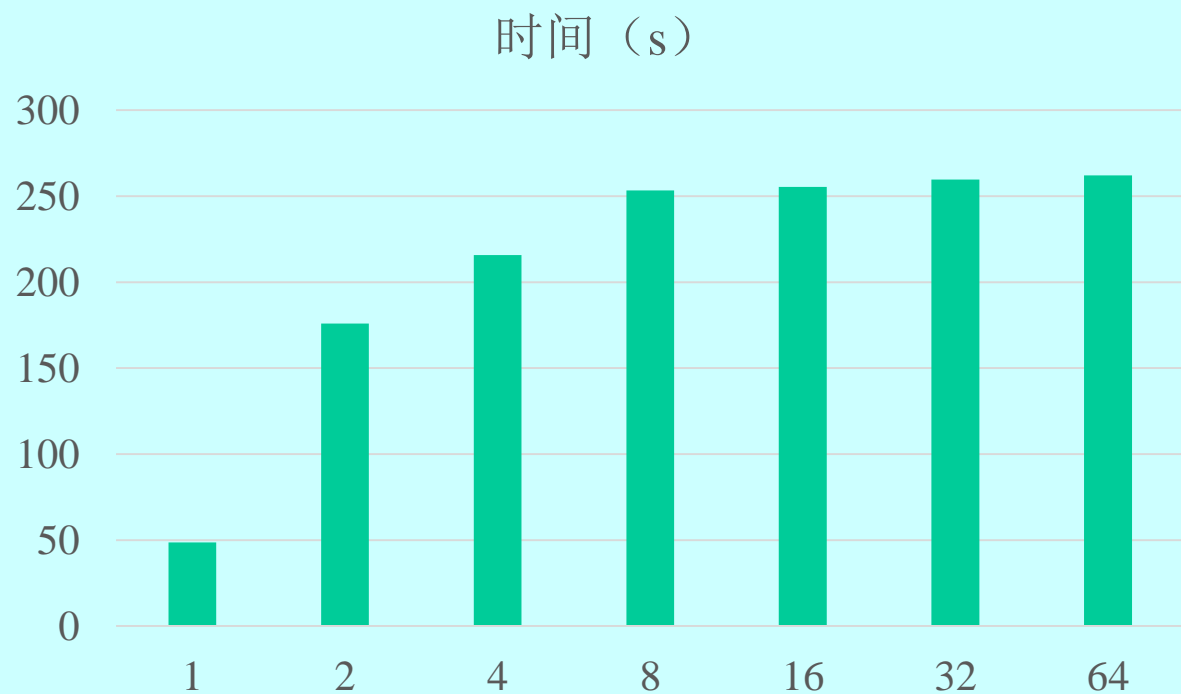
代码8.10. 多线程求和主程序

```
void sum_mutex(int id) {
    long start = id * nelems_per_thread;
    long end = start + nelems_per_thread;

    for(long i = start; i < end; i++) {
        mtx.lock();
        gsum += i;
        mtx.unlock();
    }
}
```

代码8.11. 加锁求和的sum_mutex函数

加锁对于程序性能的影响



线程数	1	2	4	8	16	32	64
时间 (s)	48.576	176.021	215.823	253.385	255.358	259.631	262.065

Lock-free 优化

```
long psum[MAXTHREADS] = { 0 };  
  
void sum_local(int id) {  
    long sum = 0;  
  
    long start = id * nelems_per_thread;  
    long end = start + nelems_per_thread;  
  
    for(long i = start; i < end; i++)  
        sum += i;  
  
    psum[id] = sum;  
}
```

代码8.12. 无锁的多线程加法

Lock-free 优化



线程数	1	2	4	8	16	32	64
时间 (s)	5.160	2.636	1.445	0.846	0.849	0.851	0.853

线程数	1	2	4	8	16	32	64
时间 (s)	48.576	176.021	215.823	253.385	255.358	259.631	262.065

C++11中的lock_guard锁

```
explicit lock_guard (mutex_type& m);
```

在构造器中，传入一个mutex互斥锁的引用，自构造完成之后，互斥锁的便上了锁，直到lock_guard类生命周期结束（退出作用域）调用析构器时，传入的互斥锁便解锁。这样，代码8.9中的counter()函数可以改写如代码8.13给出的形式。

```
void counter() {  
    for(int i = 0; i < 100000; i++) {  
        // lck的生命周期在这个作用域内  
        lock_guard<mutex> lck (mtx);  
        countNum++;  
    }  
}
```

并发List-线程安全

```
class List {  
public:  
    List() { head = NULL; }  
  
    bool insert(int key) {  
        try {  
            Node *newHead = new Node;  
            newHead->key = key;  
            {  
                lock_guard<mutex> lck(mtx);  
                newHead->next = head;  
                head = newHead;  
            }  
            return true;  
        }  
        catch (bad_alloc &e) {  
            cerr << "bad_alloc caught: " << e.what() << endl;  
            return false;  
        }  
    }  
}
```

```
bool lookup(int key) {  
    lock_guard<mutex> lck(mtx);  
    for (Node *curr = head; curr; curr = curr->next) {  
        if (curr->key == key) return true;  
    }  
    return false;  
}  
  
private:  
    struct Node {  
        int key;  
        Node *next;  
    };  
    Node *head;  
    mutex mtx;  
};
```

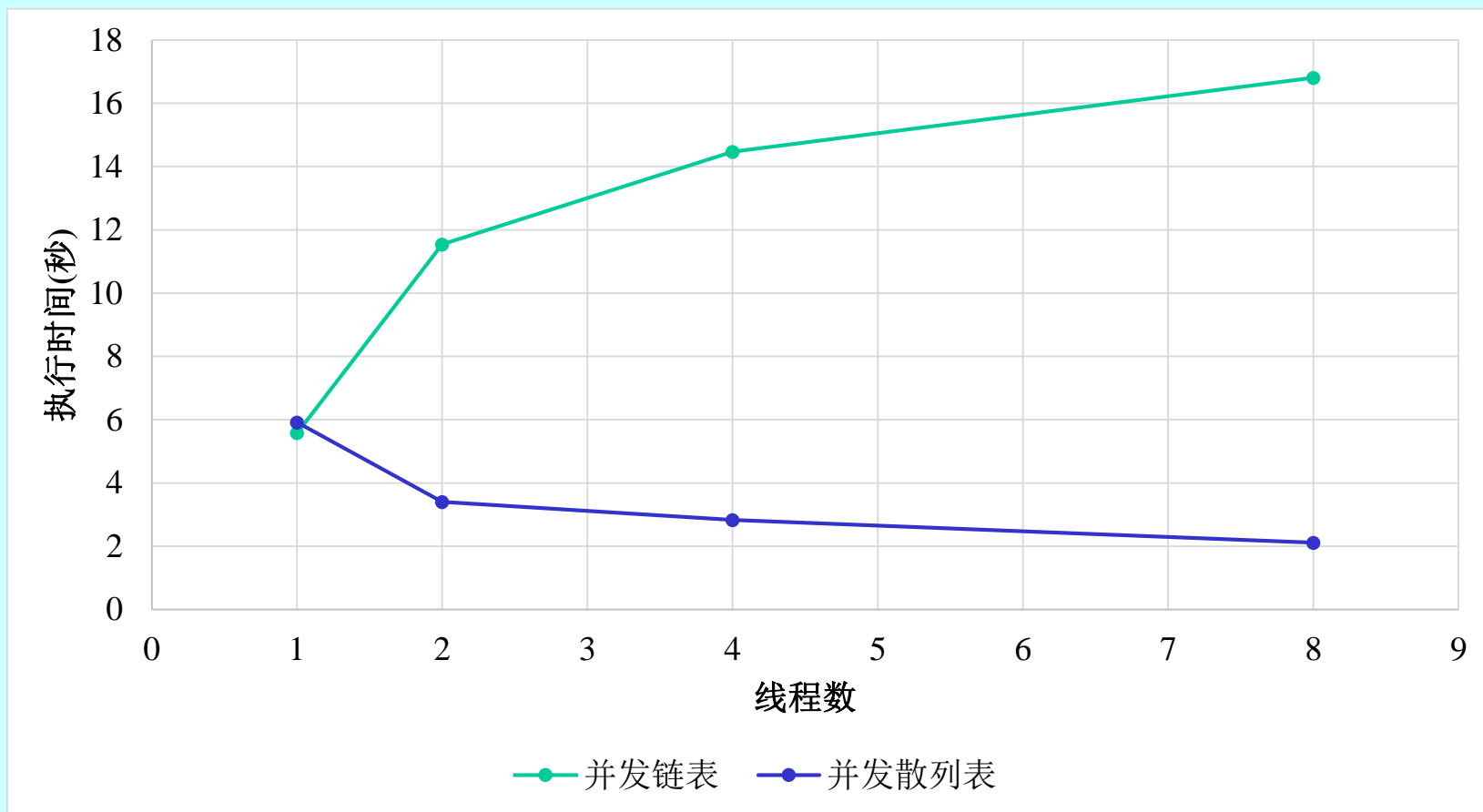
并发散列表-线程安全?

```
const int BUCKET = 101;

class Hash {
public:
    bool insert(int key) {
        int bucket = key % BUCKET;
        return lists[bucket].insert(key);
    }

    bool lookup(int key) {
        int bucket = key % BUCKET;
        return lists[bucket].lookup(key);
    }
private:
    List lists[BUCKET];
};
```


性能比较



并发链表与并发散列表的性能比较

并发队列

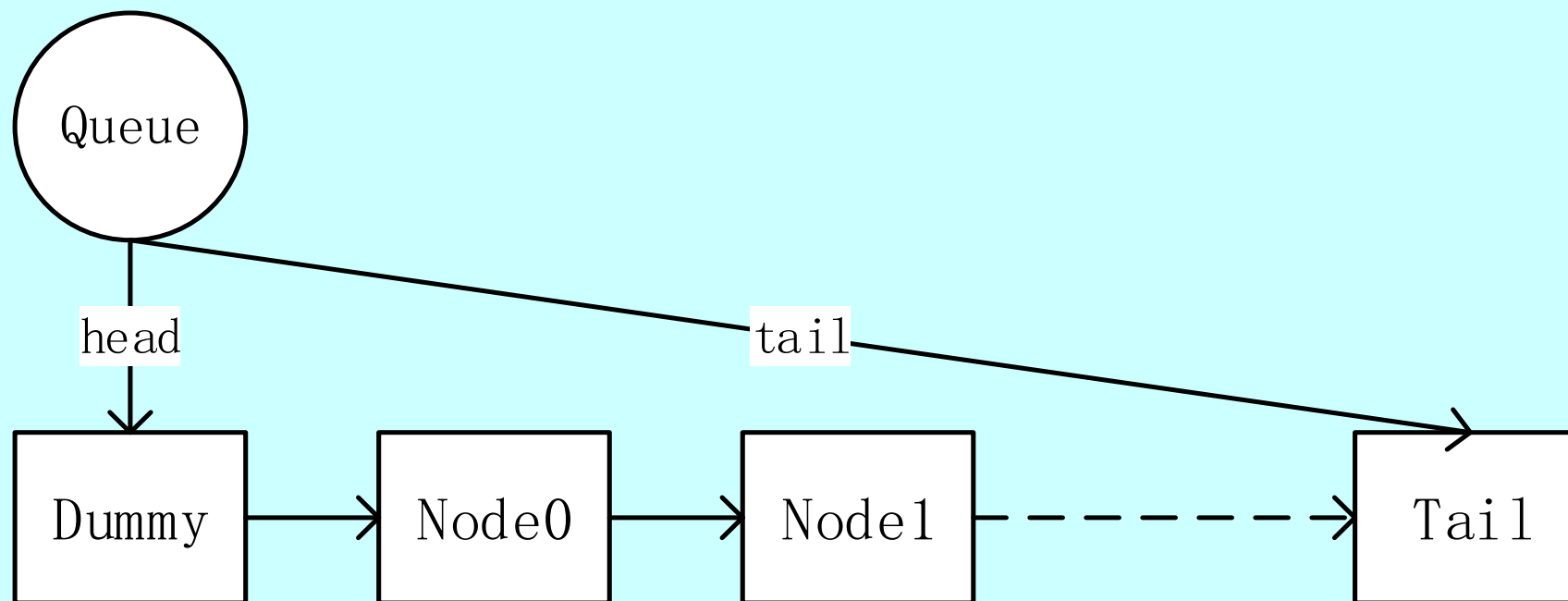


图8.2. Michael & Scott Queue

Michael和Scott两人在1996年设计出，其精要之处在于设计了一个dummy节点。目的是使对于tail和head之间的处理不会出现竞争，所以出队和入队之间不会冲突

```
class Queue {
public:
    Queue() {
        Node *dummy = new Node{ 0, NULL };
        head = tail = dummy;
    }

    void enqueue(int key) {
        Node *tmp = new Node{ key, NULL };
        lock_guard<mutex> lock(tailMtx);
        tail->next = tmp;
        tail = tmp;
    }
};
```

```
bool dequeue(int *value) {
    lock_guard<mutex> lock(headMtx);
    Node *tmp = head->next;
    if (tmp == NULL) return false;

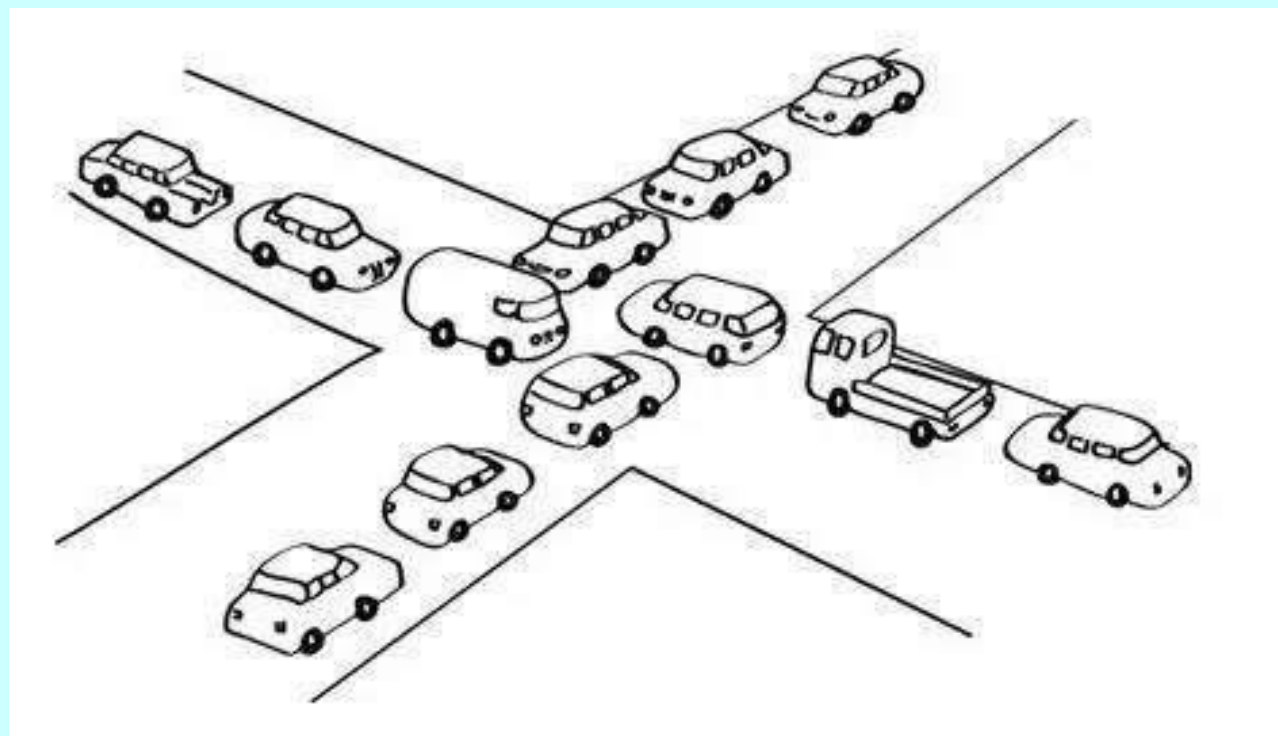
    *value = tmp->key;
    delete head;
    head = tmp;
    return true;
}

private:
    struct Node {
        int key;
        Node *next;
    };

    Node *head, *tail;
    mutex headMtx, tailMtx;
};
```

执行模式	串行	并行
执行时间（秒）	3.737	2.028

死锁



死锁

```
#include <iostream>

#include <mutex>

#include <thread>

using namespace std;

mutex mtx1;    // 互斥锁1
mutex mtx2;    // 互斥锁2

void foo() {
    mtx1.lock();
    mtx2.lock();

    cout << "foo" << endl;

    mtx1.unlock();
    mtx2.unlock();
}

void bar() {
    mtx2.lock();
    mtx1.lock();

    cout << "bar" << endl;

    mtx2.unlock();
    mtx1.unlock();
}
```

```
int main() {
    // 两个线程分别调用foo和bar并回收

    thread t1(foo), t2(bar);

    t1.join();
    t2.join();

    cout << "Finish!" << endl;

    return 0;
}
```

代码8.17. 一个会产生“死锁”程序

```
foo
bar
Finish!
foo
bar
Finish!
foo
bar
Finish!
foo
bar
Finish!
```

seq 10000 | xargs -i ./deadlock

Next

- Cuckoo hash
- 数据结构讲义