Lock-Based Data Structures

03/25/2021
Professor Amanda Bienz
Textbook pages 289-294

Lock-Based Concurrent Data Structure

- Race conditions happen around shared state
- Good programming practice generally encompasses state in a data structure or object
- Common goal : add locks to a data structure to make the structure thread safe
- Considerations:
  - Correctness : does data structure still do what we want?
  - Scalability : more threads shouldn't slow down operations

Lock-Based Concurrent Data Structure

- Race conditions happen around shared state
- Good programming practice generally encompasses state in a data structure or object
- Common goal : add locks to a data structure to make the structure thread safe
- Considerations:
  - Correctness : does data structure still do what we want?
  - Scalability : more threads shouldn't slow down operations
  - **These two goals are often in conflict**

Example : Concurrent Counters without Locks

- Simple, but not correct with multiple updaters

```
typedef struct __counter_t {
        int value;
} counter_t;

void init(counter_t *c) {
        c->value = 0;
}

void increment(counter_t *c) {
        c->value++;
}

void decrement(counter_t *c) {
        c->value--;
}

int get(counter_t *c) {
        return c->value;
}
```

Example : Concurrent Counters with Locks

- Add a single lock acquired when calling a routine that manipulates the data structure
- Reminder : Pthread_XXX is a wrapper around pthread_XXX with an error check

```
typedef struct __counter_t {
        int value;
        pthread_lock_t lock;
} counter_t;

void init(counter_t *c) {
        c->value = 0;
        Pthread_mutex_init(&c->lock, NULL);
}

void increment(counter_t *c) {
        Pthread_mutex_lock(&c->lock);
        c->value++;
        Pthread_mutex_unlock(&c->lock);
}
```

Example : Concurrent Counters with Locks (Cont.)

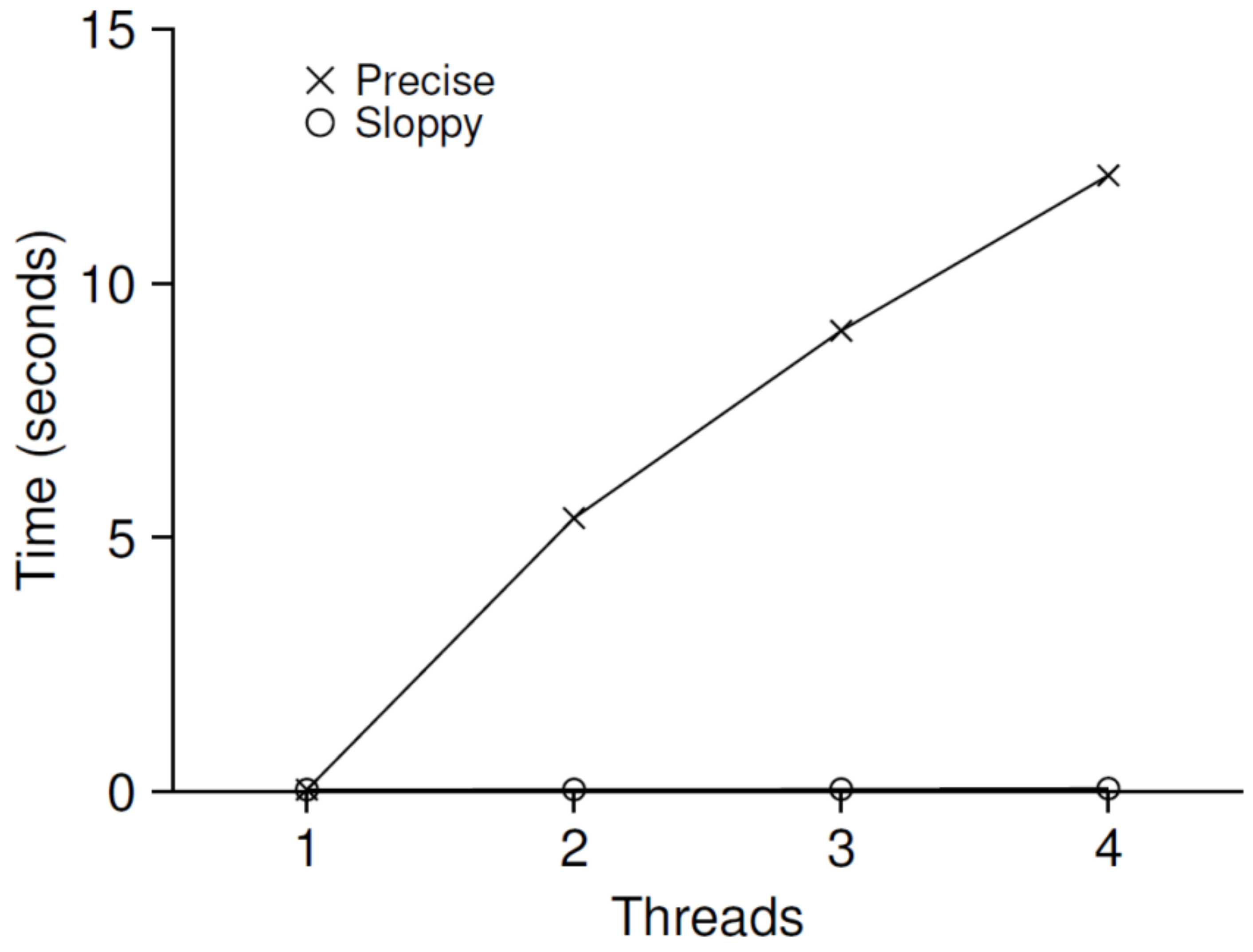
- Add a single lock acquired when calling a routine that manipulates the data structure
- Reminder : Pthread_XXX is a wrapper around pthread_XXX with an error check

```
(Cont.)
    void decrement(counter_t *c) {
        Pthread_mutex_lock(&c->lock);
        c->value--;
        Pthread_mutex_unlock(&c->lock);
    }

    int get(counter_t *c) {
        Pthread_mutex_lock(&c->lock);
        int rc = c->value;
        Pthread_mutex_unlock(&c->lock);
        return rc;
    }
```

The performance costs of the simple approach

- Each thread updates a single shared counter
  - Each thread updates the counter one million times
  - iMac with four Intel 2.7GHz i5 CPUs



Synchronized counter scales poorly.

Goal : Perfect Scaling

- Even though more work is done, it is done in parallel
- The time taken to complete the task is *not increased*
- For the counters example :
  - Perfect scaling with N threads is N times the updates in the same time
  - Our example when from less than 0.1 seconds with 1 thread to about 12 seconds with 4 threads
- Contending on locks and data structures is **very expensive** due to architectural reasons
  - Cross-CPU communication (e.g. for locking)
  - Cross-CPU cache interferences (counter moves between CPU caches, so things that hit with 1 CPU miss all the time with 2)

Sloppy Counter

- A common approach : redefine the problem by relaxing consistency
    - Often don’t need perfect count at any given time
    - Don’t want to lose any counts (eventually see all increments)
- Sloppy counter works by representing…
    - A single **logical counter** via numerous local physical counters, one per CPU core
    - A single **global counter**
    - There are **locks** : one for each local counter and one for global
- Example : machine with 4 CPUs would have 4 local counters and one global counter

Basic Idea of Sloppy Counting

- When a thread running on a core wishes to increment the counter:
  - It increments its local counter
  - Each CPU has its own local counter
    - Threads across CPUs can update local counters *without contention*
    - Thus, counter updates are scalable
  - Local values are periodically transferred to global counter
    - Acquire global lock, increment by local counter’s value, reset local counter to zero

Basic Idea of Sloppy Counting (Cont.)

- How often the local-to-global transfer occurs is determined by a threshold S (sloppiness)
  - Smaller S : counter behaves like the non-scalable counter
  - Larger S : more scalable counter, but further off global value might be from actual count
- Note : it is not a counter per thread, just per CPU
  - This is why we have a lock per local counter : multiple threads could update the counter on a single CPU
  - A counter per thread would eliminate this, but result in a lot of state if you have a lot of threads

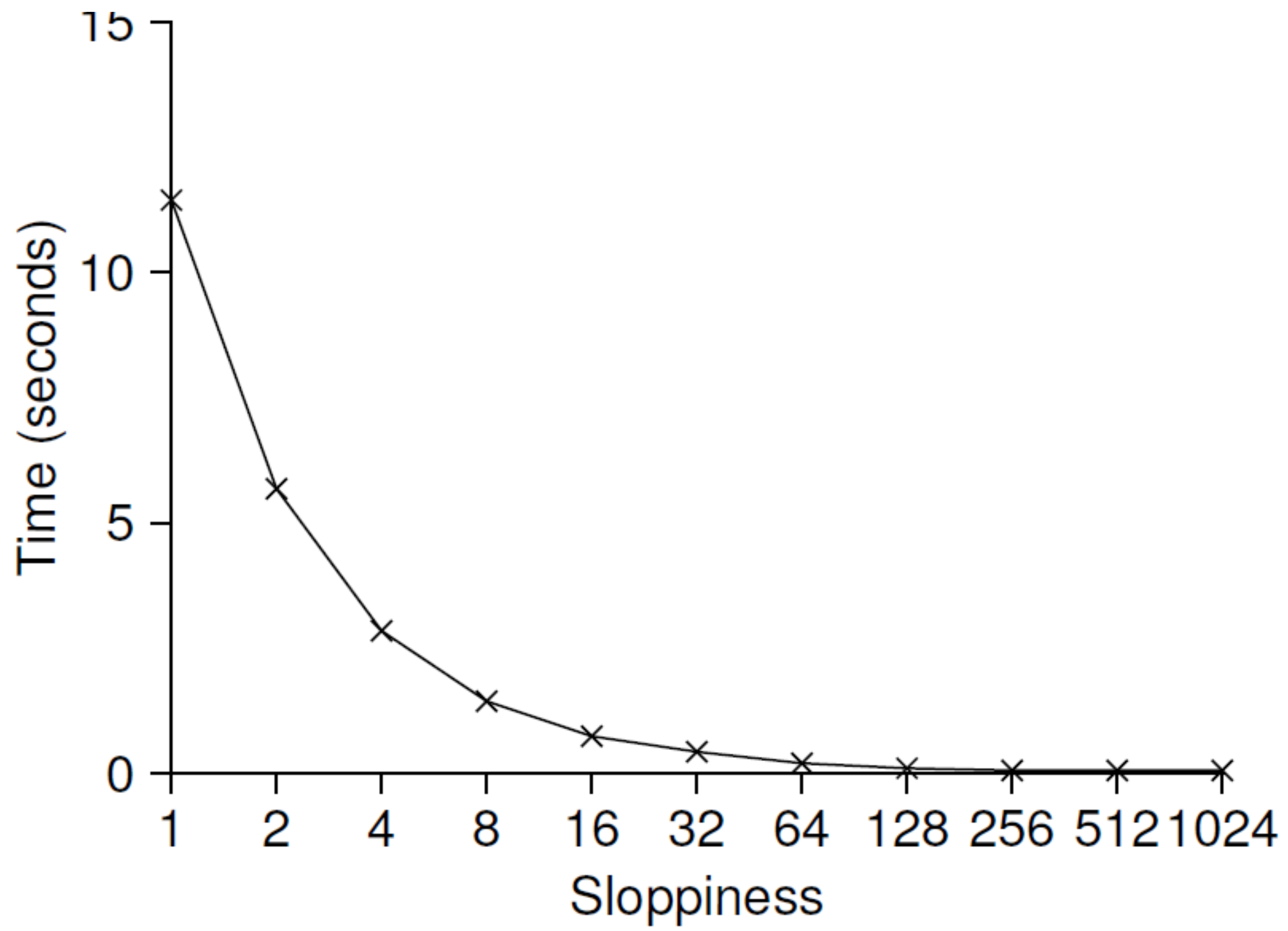
Sloppy Counter Example

- Tracing the Sloppy Counters
  - Threshold S is 5, four CPUs (on thread per CPU)
  - Each thread updates their local counters (L1 to L4)

Time	L_1	L_2	L_3	L_4	G
0	0	0	0	0	0
1	0	0	1	1	0
2	1	0	2	1	0
3	2	0	3	1	0
4	3	0	3	2	0
5	4	1	3	3	0
6	5 → 0	1	3	4	5 (from L_1)
7	0	2	4	5 → 0	10 (from L_4)

Importance of the Threshold Value S

- Each four threads increments a counter 1 million times on four CPUs
- Low S : poor performance, global count always accurate
- High S : better performance, global count is not always accurate



Scaling Sloppy Counters

Sloppy Counter Implementation

```c
typedef struct __counter_t {
    int global;                // global count
    pthread_mutex_t glock;  // global lock
    int local[NUMCPUS];     // local count (per cpu)
    pthread_mutex_t llock[NUMCPUS]; // ... and locks
    int threshold;             // update frequency
} counter_t;

// init: record threshold, init locks, init values
//       of all local counts and global count
void init(counter_t *c, int threshold) {
    c->thres hold = threshold;

    c->global = 0;
    pthread_mutex_init(&c->glock, NULL);

    int i;
    for (i = 0; i < NUMCPUS; i++) {
        c->local[i] = 0;
        pthread_mutex_init(&c->llock[i], NULL);
    }
}
```

Sloppy Counter Implementation

(Cont.)

```c
// update: usually, just grab local lock and update local amount
//         once local count has risen by 'threshold', grab global
//         lock and transfer local values to it
void update(counter_t *c, int threadID, int amt) {
    pthread_mutex_lock(&c->llock[threadID]);
    c->local[threadID] += amt;      // assumes amt > 0
    if (c->local[threadID] >= c->threshold) { // transfer to global
        pthread_mutex_lock(&c->glock);
        c->global += c->local[threadID];
        pthread_mutex_unlock(&c->glock);
        c->local[threadID] = 0;
    }
    pthread_mutex_unlock(&c->llock[threadID]);
}

// get: just return global amount (which may not be perfect)
int get(counter_t *c) {
    pthread_mutex_lock(&c->glock);
    int val = c->global;
    pthread_mutex_unlock(&c->glock);
    return val;    // only approximate!
}
```

Concurrent Linked Lists

```
// basic node structure
typedef struct __node_t {
        int key;
        struct __node_t *next;
} node_t;

// basic list structure (one used per list)
typedef struct __list_t {
        node_t *head;
        pthread_mutex_t lock;
} list_t;

void List_Init(list_t *L) {
        L->head = NULL;
        pthread_mutex_init(&L->lock, NULL);
}

```

(Cont.)

Concurrent Linked Lists (Cont.)

```
(Cont.)
    int List_Insert(list_t *L, int key) {
        pthread_mutex_lock(&L->lock);
        node_t *new = malloc(sizeof(node_t));
        if (new == NULL) {
            perror("malloc");
            pthread_mutex_unlock(&L->lock);
            return -1; // fail
        }
        new->key = key;
        new->next = L->head;
        L->head = new;
        pthread_mutex_unlock(&L->lock);
        return 0; // success
    }
(Cont.)
```

Concurrent Linked Lists (Cont.)

```
(Cont.)

    int List_Lookup(list_t *L, int key) {
        pthread_mutex_lock(&L->lock);
        node_t *curr = L->head;
        while (curr) {
            if (curr->key == key) {
                pthread_mutex_unlock(&L->lock);
                return 0; // success
            }
            curr = curr->next;
        }
        pthread_mutex_unlock(&L->lock);
        return -1; // failure
    }
```

Concurrent Linked Lists (Cont.)

- The code acquires a lock in the insert routine upon entry
- The code releases the lock upon exit
  - If malloc() happens to fail, the code must also release the lock before failing the insert
  - This kind of exceptional control flow has been shown to be quite error prone
    - You have to release a single lock in multiple places
    - Changes to how you lock/unlock have to propagate to multiple places in the code (and it is easy to miss one)
  - **Solution : the lock and release only surround the actual critical section in the insert code**

Concurrent Linked List Insert : Rewritten

```
void List_Init(list_t *L) {
        L->head = NULL;
        pthread_mutex_init(&L->lock, NULL);
}

void List_Insert(list_t *L, int key) {
        // synchronization not needed
        node_t *new = malloc(sizeof(node_t));
        if (new == NULL) {
                perror("malloc");
                return;
        }
        new->key = key;

        // just lock critical section
        pthread_mutex_lock(&L->lock);
        new->next = L->head;
        L->head = new;
        pthread_mutex_unlock(&L->lock);
}
```

Scaling Linked List

- Current linked list has poor scalability : lock the entire list while you walk it
- Hand-over-hand locking (lock coupling)
  - Add a lock per node of the list instead of having a single lock for the entire list
  - While traversing the list: 1. Grab next node's lock and 2. Release current node's lock
  - Enable a high degree of concurrency in list operations
    - However, in practice, the overheads of acquiring and releasing locks for each node of a list traversal is prohibitive
- **Scaling arbitrary linked lists is difficult because the sheer amount of state to be protected**

Michael and Scott Concurrent Queues

- What if all we want is a queue?
- There are two locks
  - One for the **head** of the queue
  - One for the **tail**
  - **The goal of these two locks is to enable concurrency of enqueue and dequeue operations**
- Add a dummy node
  - Allocated in the queue initialization code
  - Enable the separation of head and tail operations

Concurrent Queues (Cont.)

```
typedef struct __node_t {
        int value;
        struct __node_t *next;
} node_t;

typedef struct __queue_t {
        node_t *head;
        node_t *tail;
        pthread_mutex_t headLock;
        pthread_mutex_t tailLock;
} queue_t;

void Queue_Init(queue_t *q) {
        node_t *tmp = malloc(sizeof(node_t));
        tmp->next = NULL;
        q->head = q->tail = tmp;
        pthread_mutex_init(&q->headLock, NULL);
        pthread_mutex_init(&q->tailLock, NULL);
}

(Cont.)
```

Concurrent Queues (Cont.)

```
(Cont.)
    void Queue_Enqueue(queue_t *q, int value) {
        node_t *tmp = malloc(sizeof(node_t));
        assert(tmp != NULL);

        tmp->value = value;
        tmp->next = NULL;

        pthread_mutex_lock(&q->tailLock);
        q->tail->next = tmp;
        q->tail = tmp;
        pthread_mutex_unlock(&q->tailLock);
    }
(Cont.)
```

Concurrent Queues (Cont.)

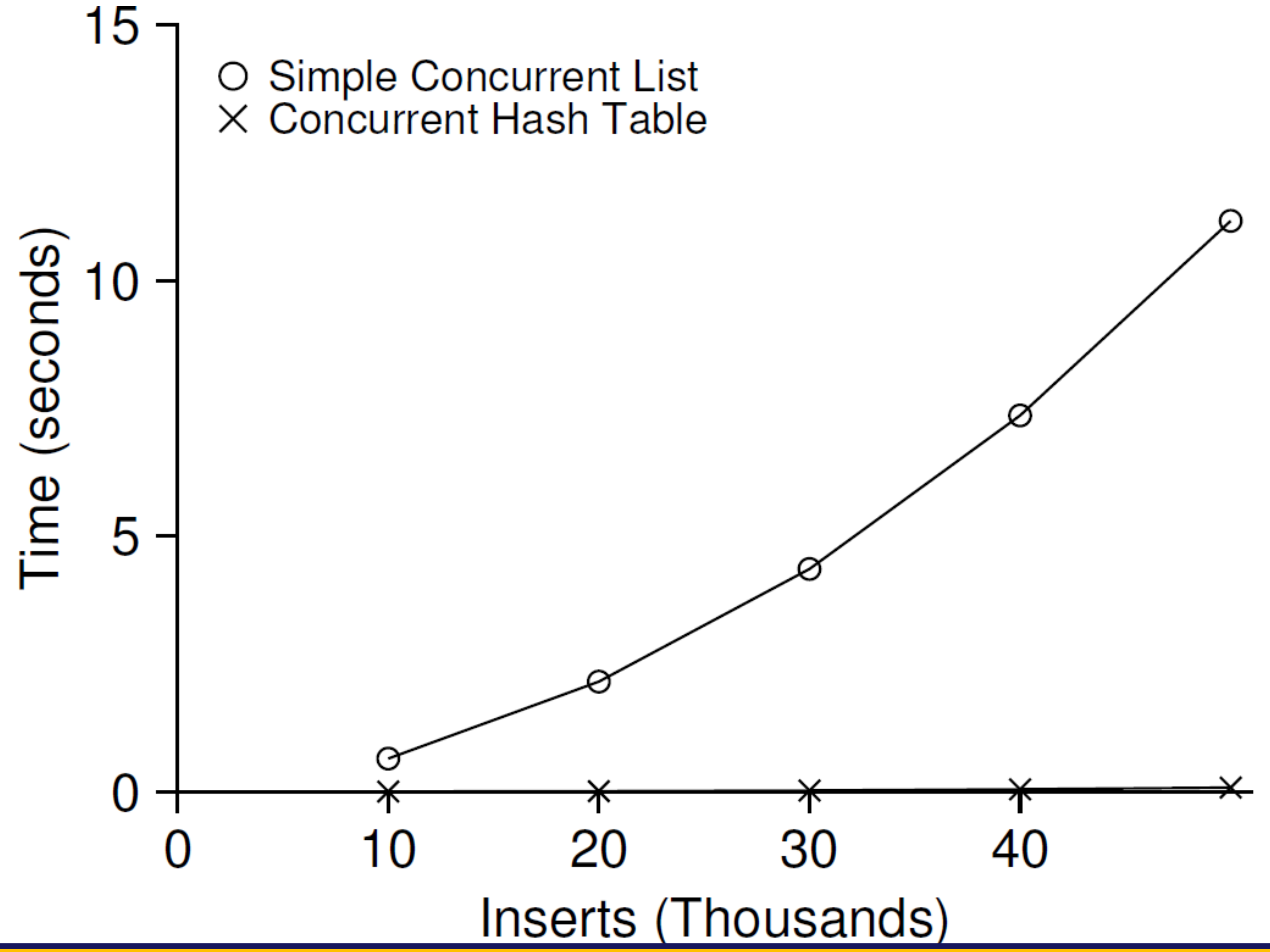
```
(Cont.)
        int Queue_Dequeue(queue_t *q, int *value) {
                pthread_mutex_lock(&q->headLock);
                node_t *tmp = q->head;
                node_t *newHead = tmp->next;
                if (newHead == NULL) {
                        pthread_mutex_unlock(&q->headLock);
                        return -1; // queue was empty
                }
                *value = newHead->value;
                q->head = newHead;
                pthread_mutex_unlock(&q->headLock);
                free(tmp);
                return 0;
        }
```

Concurrent Hash Table

- Focus on a simple hash table
  - The hash table does not resize
  - Build using the concurrent lists
  - It uses a **lock per hash bucket** each of which is represented by a list

Performance of Concurrent Hash Table

- From 10,000 to 50,000 concurrent updates forom each of four threads
  - iMac with four Intel 2.7 GHz i5 CPUs



- **The simple concurrent hash table scales magnificently.**
- **With few threads << buckets, threads are generally on independent lists!**

Concurrent Hash Table

```
#define BUCKETS (101)

typedef struct __hash_t {
        list_t lists[BUCKETS];
} hash_t;

void Hash_Init(hash_t *H) {
        int i;
        for (i = 0; i < BUCKETS; i++) {
                List_Init(&H->lists[i]);
        }
}

int Hash_Insert(hash_t *H, int key) {
        int bucket = key % BUCKETS;
        return List_Insert(&H->lists[bucket], key);
}

int Hash_Lookup(hash_t *H, int key) {
        int bucket = key % BUCKETS;
        return List_Lookup(&H->lists[bucket], key);
}
```

Reading

- Deadlocks : pg 283
- Synchronization problems : pg 289-294
- POSIX : pg 299-303
- Alternative approaches : pg 311-313