13 Concurrent Hashing and Natural Parallelism

13.1 Introduction

In earlier chapters, we studied how to extract parallelism from data structures like queues, stacks, and counters, that seemed to provide few opportunities for parallelism. In this chapter we take the opposite approach. We study *concurrent hashing*, a problem that seems to be "naturally parallelizable" or, using a more technical term, *disjoint–access–parallel*, meaning that concurrent method calls are likely to access disjoint locations, implying that there is little need for synchronization.

Hashing is a technique commonly used in sequential `Set` implementations to ensure that `contains`(), `add`(), and `remove`() calls take constant average time. The concurrent `Set` implementations studied in Chapter 9 required time linear in the size of the set. In this chapter, we study ways to make hashing concurrent, sometimes using locks and sometimes not. Even though hashing seems naturally parallelizable, devising an effective concurrent hash algorithm is far from trivial.

As in earlier chapters, the `Set` interface provides the following methods, which return Boolean values:

- `add`(x) adds x to the set. Returns *true* if x was absent, and *false* otherwise,
- `remove`(x) removes x from the set. Returns *true* if x was present, and *false* otherwise, and
- `contains`(x) returns *true* if x is present, and *false* otherwise.

When designing set implementations, we need to keep the following principle in mind: we can buy more memory, but we cannot buy more time. Given a choice between an algorithm that runs faster but consumes more memory, and a slower algorithm that consumes less memory, we tend to prefer the faster algorithm (within reason).

A *hash set* (sometimes called a *hash table*) is an efficient way to implement a set. A hash set is typically implemented as an array, called the *table*. Each table

entry is a reference to one or more *items*. A *hash function* maps items to integers so that distinct items usually map to distinct values. (Java provides each object with a `hashCode()` method that serves this purpose.) To add, remove, or test an item for membership, apply the hash function to the item (modulo the table size) to identify the table entry associated with that item. (We call this step *hashing* the item.)

In some hash-based set algorithms, each table entry refers to a single item, an approach known as *open addressing*. In others, each table entry refers to a set of items, traditionally called a *bucket*, an approach known as *closed addressing*.

Any hash set algorithm must deal with *collisions*: what to do when two distinct items hash to the same table entry. Open-addressing algorithms typically resolve collisions by applying alternative hash functions to test alternative table entries. Closed-addressing algorithms place colliding items in the same bucket, until that bucket becomes too full. In both kinds of algorithms, it is sometimes necessary to *resize* the table. In open-addressing algorithms, the table may become too full to find alternative table entries, and in closed-addressing algorithms, buckets may become too large to search efficiently.

Anecdotal evidence suggests that in most applications, sets are subject to the following distribution of method calls: 90% `contains()`, 9% `add()`, and 1% `remove()` calls. As a practical matter, sets are more likely to grow than to shrink, so we focus here on *extensible hashing* in which hash sets only grow (shrinking them is a problem for the exercises).

It is easier to make closed-addressing hash set algorithms parallel, so we consider them first.

13.2 Closed-Address Hash Sets

Pragma 13.2.1. Here and elsewhere, we use the standard Java `List<T>` interface (in package java.util.List). A `List<T>` is an ordered collection of `T` objects, where `T` is a type. Here, we make use of the following List methods: `add(`x`)` appends x to the end of the list, `get(`i`)` returns (but does not remove) the item at position i, `contains(`x`)` returns *true* if the list contains x. There are many more.

The `List` interface can be implemented by a number of classes. Here, it is convenient to use the `ArrayList` class.

We start by defining a *base* hash set implementation common to all the concurrent closed-addressing hash sets we consider here. The `BaseHashSet<T>` class is an *abstract* class, that is, it does not implement all its methods. Later, we look at three alternative synchronization techniques: one using a single coarse-grained lock, one using a fixed-size array of locks, and one using a resizable array of locks.

```
1  public abstract class BaseHashSet<T> {
2    protected List<T>[] table;
3    protected int setSize;
4    public BaseHashSet(int capacity) {
5      setSize = 0;
6      table = (List<T>[]) new List[capacity];
7      for (int i = 0; i < capacity; i++) {
8        table[i] = new ArrayList<T>();
9      }
10   }
11   ...
12 }
```

Figure 13.1 BaseHashSet<T> class: fields and constructor.

Fig. 13.1 shows the base hash set's fields and constructor. The table[] field is an array of buckets, each of which is a set implemented as a list (Line 2). We use ArrayList<T> lists for convenience, supporting the standard sequential add(), remove(), and contains() methods. The setSize field is the number of items in the table (Line 3). We sometimes refer to the length of the table[] array, that is, the number of buckets in it, as its *capacity*.

The BaseHashSet<T> class does not implement the following *abstract* methods: acquire(x) acquires the locks necessary to manipulate item x, release(x) releases them, policy() decides whether to resize the set, and resize() doubles the capacity of the table[] array. The acquire(x) method must be *reentrant* (Chapter 8, Section 8.4), meaning that if a thread that has already called acquire(x) makes the same call, then it will proceed without deadlocking with itself.

Fig. 13.2 shows the contains(x) and add(x) methods of the BaseHashSet<T> class. Each method first calls acquire(x) to perform the necessary synchronization, then enters a **try** block whose **finally** block calls release(x). The contains(x) method simply tests whether x is present in the associated bucket (Line 17), while add(x) adds x to the list if it is not already present (Line 27).

How big should the bucket array be to ensure that method calls take constant expected time? Consider an add(x) call. The first step, hashing x, takes constant time. The second step, adding the item to the bucket, requires traversing a linked list. This traversal takes constant expected time only if the lists have constant expected length, so the table capacity should be proportional to the number of items in the table. This number may vary unpredictably over time, so to ensure that method call times remain (more-or-less) constant, we must *resize* the table every now and then to ensure that list lengths remain (more-or-less) constant.

We still need to decide *when* to resize the hash set, and how the resize() method synchronizes with the others. There are many reasonable alternatives. For closed-addressing algorithms, one simple strategy is to resize the set when the average bucket size exceeds a fixed threshold. An alternative policy employs two fixed integer quantities: the *bucket threshold* and the *global threshold*.

```
13    public boolean contains(T x) {
14      acquire(x);
15      try {
16        int myBucket = x.hashCode() % table.length;
17        return table[myBucket].contains(x);
18      } finally {
19        release(x);
20      }
21    }
22    public boolean add(T x) {
23      boolean result = false;
24      acquire(x);
25      try {
26        int myBucket = x.hashCode() % table.length;
27        result = table[myBucket].add(x);
28        setSize = result ? setSize + 1 : setSize;
29      } finally {
30        release(x);
31      }
32      if (policy())
33        resize();
34      return result;
35    }
```

Figure 13.2 `BaseHashSet<T>` class: the `contains()` and `add()` methods hash the item to choose a bucket.

- If more than, say, 1/4 of the buckets exceed the bucket threshold, then double the table capacity, or
- If any single bucket exceeds the global threshold, then double the table capacity.

Both these strategies work well in practice, as do others. Open-addressing algorithms are slightly more complicated, and are discussed later.

13.2.1 A Coarse-Grained Hash Set

Fig. 13.3 shows the `CoarseHashSet<T>` class's fields, constructor, `acquire`(x), and `release`(x) methods. The constructor first initializes its superclass (Line 4). Synchronization is provided by a single reentrant lock (Line 2), acquired by `acquire`(x) (Line 8) and released by `release`(x) (Line 11).

Fig. 13.4 shows the `CoarseHashSet<T>` class's `policy`() and `resize`() methods. We use a simple policy: we resize when the average bucket length exceeds 4 (Line 16). The `resize`() method locks the set (Line 20), and checks that no other thread has resized the table in the meantime (Line 23). It then allocates and initializes a new table with double the capacity (Lines 25–29) and transfers items from the old to the new buckets (Lines 30–34). Finally, it unlocks the set (Line 36).

```
1  public class CoarseHashSet<T> extends BaseHashSet<T>{
2    final Lock lock;
3    CoarseHashSet(int capacity) {
4      super(capacity);
5      lock = new ReentrantLock();
6    }
7    public final void acquire(T x) {
8      lock.lock();
9    }
10   public void release(T x) {
11     lock.unlock();
12   }
13   ...
14 }
```

Figure 13.3 `CoarseHashSet<T>` class: fields, constructor, `acquire()`, and `release()` methods.

```
15   public boolean policy() {
16     return setSize / table.length > 4;
17   }
18   public void resize() {
19     int oldCapacity = table.length;
20     lock.lock();
21     try {
22       if (oldCapacity != table.length) {
23         return; // someone beat us to it
24       }
25       int newCapacity = 2 * oldCapacity;
26       List<T>[] oldTable = table;
27       table = (List<T>[]) new List[newCapacity];
28       for (int i = 0; i < newCapacity; i++)
29         table[i] = new ArrayList<T>();
30       for (List<T> bucket : oldTable) {
31         for (T x : bucket) {
32           table[x.hashCode() % table.length].add(x);
33         }
34       }
35     } finally {
36       lock.unlock();
37     }
38   }
```

Figure 13.4 `CoarseHashSet<T>` class: the `policy()` and `resize()` methods.

13.2.2 A Striped Hash Set

Like the coarse-grained list studied in Chapter 9, the coarse-grained hash set shown in the last section is easy to understand and easy to implement. Unfortunately, it is also a sequential bottleneck. Method calls take effect in a one-at-a-time order, even when there is no logical reason for them to do so.

We now present a closed address hash table with greater parallelism and less lock contention. Instead of using a single lock to synchronize the entire set, we split the set into independently synchronized pieces. We introduce a technique called *lock striping*, which will be useful for other data structures as well. Fig. 13.5 shows the fields and constructor for the `StripedHashSet<T>` class. The set is initialized with an array `locks[]` of L locks, and an array `table[]` of $N = L$ buckets, where each bucket is an unsynchronized `List<T>`. Although these arrays are initially of the same capacity, `table[]` will grow when the set is resized, but `lock[]` will not. Every now and then, we double the table capacity N without changing the lock array size L, so that lock i eventually protects each table entry j, where $j = i \pmod{L}$. The `acquire`(x) and `release`(x) methods use x's hash code to pick which lock to acquire or release. An example illustrating how a `StripedHashSet<T>` is resized appears in Fig. 13.6.

There are two reasons not to grow the lock array every time we grow the table:

- Associating a lock with every table entry could consume too much space, especially when tables are large and contention is low.
- While resizing the table is straightforward, resizing the lock array (while in use) is more complex, as discussed in Section 13.2.3.

Resizing a `StripedHashSet` (Fig. 13.7) is almost identical to resizing a `CoarseHashSet`. One difference is that `resize`() acquires the locks in `lock[]` in ascending order (Lines 18–20). It cannot deadlock with a `contains`(), `add`(), or `remove`() call because these methods acquire only a single lock. A `resize`() call cannot deadlock with another `resize`() call because both calls start without holding any locks, and acquire the locks in the same order. What if two or more threads try to resize at the same time? As in the `CoarseHashSet<T>`, when a thread starts to resize the table, it records the current table capacity. If, after it has acquired all the locks, it discovers that some other thread has changed the table

```
1  public class StripedHashSet<T> extends BaseHashSet<T>{
2    final ReentrantLock[] locks;
3    public StripedHashSet(int capacity) {
4      super(capacity);
5      locks = new Lock[capacity];
6      for (int j = 0; j < locks.length; j++) {
7        locks[j] = new ReentrantLock();
8      }
9    }
10   public final void acquire(T x) {
11     locks[x.hashCode() % locks.length].lock();
12   }
13   public void release(T x) {
14     locks[x.hashCode() % locks.length].unlock();
15   }
```

Figure 13.5 `StripedHashSet<T>` class: fields, constructor, `acquire()`, and `release()` methods.

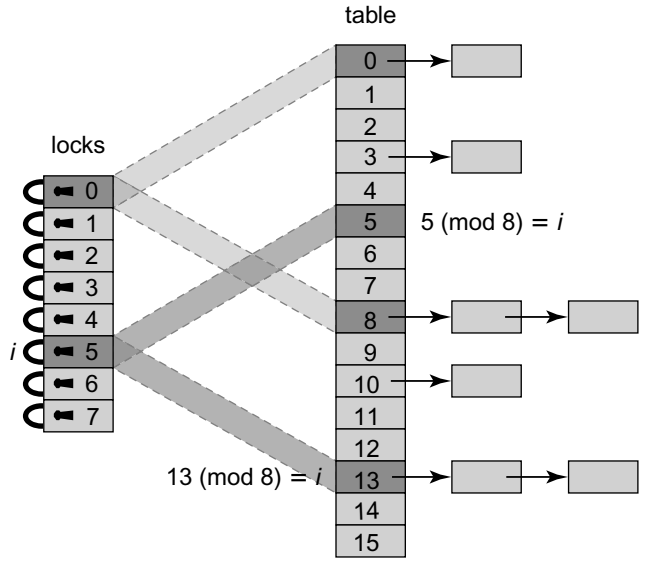

Figure 13.6 Resizing a `StripedHashSet` lock-based hash table. As the table grows, the striping is adjusted to ensure that each lock covers $2^{N/L}$ entries. In the figure above, $N = 16$ and $L = 8$. When N is doubled from 8 to 16, the memory is striped so that lock $i = 5$ for example covers both locations that are equal to 5 modulo L.

capacity (Line 23), then it releases the locks and gives up. (It could just double the table size anyway, since it already holds all the locks.)

Otherwise, it creates a new `table[]` array with twice the capacity (Line 25), and transfer items from the old table to the new (Line 30). Finally, it releases the locks (Line 36). Because the `initializeFrom()` method calls `add()`, it may trigger nested calls to `resize()`. We leave it as an exercise to check that nested resizing works correctly in this and later hash set implementations.

To summarize, striped locking permits more concurrency than a single coarse-grained lock because method calls whose items hash to different locks can proceed in parallel. The `add()`, `contains()`, and `remove()` methods take constant expected time, but `resize()` takes linear time and is a "stop-the-world" operation: it halts all concurrent method calls while it increases the table's capacity.

13.2.3 A Refinable Hash Set

What if we want to refine the granularity of locking as the table size grows, so that the number of locations in a stripe does not continuously grow? Clearly, if we want to resize the lock array, then we need to rely on another form of synchronization. Resizing is rare, so our principal goal is to devise a way to permit the lock array to be resized without substantially increasing the cost of normal method calls.

```
16    public void resize() {
17      int oldCapacity = table.length;
18      for (Lock lock : locks) {
19        lock.lock();
20      }
21      try {
22        if (oldCapacity != table.length) {
23          return; // someone beat us to it
24        }
25        int newCapacity = 2 * oldCapacity;
26        List<T>[] oldTable = table;
27        table = (List<T>[]) new List[newCapacity];
28        for (int i = 0; i < newCapacity; i++)
29          table[i] = new ArrayList<T>();
30        for (List<T> bucket : oldTable) {
31          for (T x : bucket) {
32            table[x.hashCode() % table.length].add(x);
33          }
34        }
35      } finally {
36        for (Lock lock : locks) {
37          lock.unlock();
38        }
39      }
40    }
```

Figure 13.7 `StripedHashSet<T>` class: to resize the set, lock each lock in order, then check that no other thread has resized the table in the meantime.

```
1  public class RefinableHashSet<T> extends BaseHashSet<T>{
2    AtomicMarkableReference<Thread> owner;
3    volatile ReentrantLock[] locks;
4    public RefinableHashSet(int capacity) {
5      super(capacity);
6      locks = new ReentrantLock[capacity];
7      for (int i = 0; i < capacity; i++) {
8        locks[i] = new ReentrantLock();
9      }
10     owner = new AtomicMarkableReference<Thread>(null, false);
11   }
12   ...
13 }
```

Figure 13.8 `RefinableHashSet<T>` class: fields and constructor.

Fig. 13.8 shows the fields and constructor for the `RefinableHashSet<T>` class. To add a higher level of synchronization, we introduce a globally shared `owner` field that combines a Boolean value with a reference to a thread. Normally, the Boolean value is *false*, meaning that the set is not in the middle of resizing. While a resizing is in progress, however, the Boolean value is *true*, and the associated reference indicates the thread that is in charge of resizing. These

two values are combined in an AtomicMarkableReference<Thread> to allow them to be modified atomically (see Pragma 9.8.1 in Chapter 9). We use the owner as a mutual exclusion flag between the resize() method and any of the add() methods, so that while resizing, there will be no successful updates, and while updating, there will be no successful resizes. Every add() call must read the owner field. Because resizing is rare, the value of owner should usually be cached.

Each method locks the bucket for x by calling acquire(x), shown in Fig. 13.9. It spins until no other thread is resizing the set (Lines 19–21), and then reads the lock array (Line 22). It then acquires the item's lock (Line 24), and checks again, this time while holding the locks (Line 26), to make sure no other thread is resizing, and that no resizing took place between Lines 21 and 26.

If it passes this test, the thread can proceed. Otherwise, the locks it has acquired could be out-of-date because of an ongoing update, so it releases them and starts over. When starting over, it will first spin until the current resize completes (Lines 19–21) before attempting to acquire the locks again. The release(x) method releases the locks acquired by acquire(x).

The resize() method is almost identical to the resize() method for the StripedHashSet class. The one difference appears on Line 46: instead of acquiring all the locks in lock[], the method calls quiesce() (Fig. 13.10) to ensure that no other thread is in the middle of an add(), remove(), or contains() call. The quiesce() method visits each lock and waits until it is unlocked.

```
14    public void acquire(T x) {
15      boolean[] mark = {true};
16      Thread me = Thread.currentThread();
17      Thread who;
18      while (true) {
19        do {
20          who = owner.get(mark);
21        } while (mark[0] && who != me);
22        ReentrantLock[] oldLocks = locks;
23        ReentrantLock oldLock = oldLocks[x.hashCode() % oldLocks.length];
24        oldLock.lock();
25        who = owner.get(mark);
26        if ((!mark[0] || who == me) && locks == oldLocks) {
27          return;
28        } else {
29          oldLock.unlock();
30        }
31      }
32    }
33    public void release(T x) {
34      locks[x.hashCode() % locks.length].unlock();
35    }
```

Figure 13.9 RefinableHashSet<T> class: acquire() and release() methods.

```
36   public void resize() {
37     int oldCapacity = table.length;
38     boolean[] mark = {false};
39     int newCapacity = 2 * oldCapacity;
40     Thread me = Thread.currentThread();
41     if (owner.compareAndSet(null, me, false, true)) {
42       try {
43         if (table.length != oldCapacity) { // someone else resized first
44           return;
45         }
46         quiesce();
47         List<T>[] oldTable = table;
48         table = (List<T>[]) new List[newCapacity];
49         for (int i = 0; i < newCapacity; i++)
50           table[i] = new ArrayList<T>();
51         locks = new ReentrantLock[newCapacity];
52         for (int j = 0; j < locks.length; j++) {
53           locks[j] = new ReentrantLock();
54         }
55         initializeFrom(oldTable);
56       } finally {
57         owner.set(null, false);
58       }
59     }
60   }
```

Figure 13.10 `RefinableHashSet<T>` class: `resize()` method.

```
61   protected void quiesce() {
62     for (ReentrantLock lock : locks) {
63       while (lock.isLocked()) {}
64     }
65   }
```

Figure 13.11 `RefinableHashSet<T>` class: `quiesce()` method.

The `acquire()` and the `resize()` methods guarantee mutually exclusive access via the flag principle using the `mark` field of the `owner` flag and the table's locks array: `acquire()` first acquires its locks and then reads the `mark` field, while `resize()` first sets `mark` and then reads the locks during the `quiesce()` call. This ordering ensures that any thread that acquires the locks after `quiesce()` has completed will see that the set is in the processes of being resized, and will back off until the resizing is complete. Similarly, `resize()` will first set the `mark` field, then read the locks, and will not proceed while any `add()`, `remove()`, or `contains()` call's lock is set.

To summarize, we have seen that one can design a hash table in which both the number of buckets and the number of locks can be continuously resized. One limitation of this algorithm is that threads cannot access the items in the table during a resize.