An Implementation of Concurrent Union-find Data Structure in Java

Yu Wang (wangy22@mcmaster.ca)

Contents

1 Introduction 1
2 Class: DisjointSet 1
3 Testing 4
4 Test Result 8
A Defined Chunks 8
B Index 9

1 Introduction

Written in Java, DisjointSet implements an concurrent Union-Find data structure discussed in [1] using disjoint forests. Concurrency is implemented using threads in Java and is guaranteed calling lock and unlock methods before and after the critical sections. We call such data structure thread safe.

To allow any object to be contained in the disjoint set, our implementation uses generics[2], which is represented by T. For testing, we use long integer type, Long for the disjoint set. Testing results from both serial code and concurrent code are compared.

The Java code is to be compiled by Sun JDK 1.5.

2 Class: DisjointSet

Following is the root chunk of DisjointSet class. It imports java.util.* for Map interface and its implementation TreeMap; and java.util.concurrent.locks.* for Lock interface and its implementation ReentrantLock.

In class DisjointSet of type T, a tree map Parent is defined, for mapping each element in to its parent element in the set. lock is defined to locking and unlocking the critical section, in order to prevent concurrent writing to the same resource.
import java.util.*;
import java.util.concurrent.locks.*;

class DisjointSet <T> {
    private Map <T, T> Parent = new TreeMap <T, T> ();
    private final Lock lock = new ReentrantLock();

    public void makeSet (T x) {
        lock.lock();
        try {
            Parent.put(x, x);
        } finally {
            lock.unlock();
        }
    }
}

Defines:
    Parent, used in chunks 1–5.
    lock, used in chunks 1–3.
    Root chunk (not used in this document).

Major operations to a disjoint set are makeSet, union and findSet. In makeSet method, new mapping from x to x is put into Parent, which means an element x is the parent of itself, for identifying itself as a set. Since this operation writes to the shared resource, it is a critical section.
In union method, all elements whose parent is the same as element \( y \) will be assigned to parent of \( x \), which means two sets containing \( x \) and \( y \) respectively are joined. Since this operation writes to the shared resource, it is a critical section too.

```java
3 ⟨union 3⟩≡
public void union (T x, T y) {
    lock.lock();
    try {
        for ( T z : Parent.keySet() ) {
            if ( findSet(z).equals( findSet(y) ) ) {
                Parent.put(z, findSet(x));
            }
        }
    } finally {
        lock.unlock();
    }
}
```

Uses Parent 1 and lock 1. This code is used in chunk 1.

In findSet method, Parent of \( x \) is recursively searched. The return element is the representative of the disjoint set to which \( x \) belongs. During the process, no information is written to the resource, thus locking and unlocking is unnecessary.

```java
4 ⟨findSet 4⟩≡
public T findSet (T x) {
    if ( Parent.get(x).equals(x) ) {
        return x;
    } else {
        return findSet (Parent.get(x));
    }
}
```

Uses Parent 1. This code is used in chunk 1.

For testing purpose, getMap method returns Parent to allow internal information being retrieved.

```java
5 ⟨getMap 5⟩≡
public Map <T, T> getMap () {
    return Parent;
}
```

Uses Parent 1. This code is used in chunk 1.
3 Testing

For testing, we want both serial test and concurrent test, and see if the results are the same. In main body, serialTest is firstly called, such that only one thread exists and no two operations are performed at the same time. After that, concurrentTest is called, which creates SIZE threads running simultaneously performing operations that write to the shared resource.

The test cases are Long integers randomly generated. They are to be grouped by modulus with respect to 2, which are -1, 0 and 1; or negative odd numbers, even numbers and positive odd numbers. The initial disjoint sets are -1, 0 and 1, so that new numbers can be grouped respectively by union method. At the end, both tests print the contents of disjoint sets, which suppose to be identical.

```java
import java.util.*;

public class Test implements Runnable {
    static final int SIZE = 20;
    static Long data[] = new Long [SIZE];
    static DisjointSet <Long> cs = new DisjointSet <Long> ();

    public void run() {
        serialTest();
        concurrentTest();
        main();
    }

    private void serialTest() {
        // Serial test code
    }

    private void concurrentTest() {
        // Concurrent test code
    }

    private void main() {
        // Main code
    }
}
```

Defines:
- SIZE, used in chunks 6–10.
- cs, used in chunks 9 and 10.
- data, used in chunks 6–8 and 10.

Root chunk (not used in this document).
In the main entrance, random generator $r$ takes the ID of current thread as its seed, in order to generate different set of random numbers at each run. Generated test cases are saved in array $data$ of $Long$ integers. After that, $serialTest()$ and $concurrentTest()$ are called.

```java
⟨main⟩
≡
public static void main(String[] args) {
    Random r = new Random(Thread.currentThread().getId());
    for (int i = 0; i < SIZE; i++) {
        data[i] = r.nextLong();
    }
    serialTest();
    concurrentTest();
}
```

Uses $SIZE$ 6 and $data$ 6.
This code is used in chunk 6.

In $serialTest$, DisjointSet $ss$ is created. It makes -1, 0 and 1 into disjoint sets first by calling $makeSet$ method. Then it traverse through each number in $data$, make it a set and union it with its modulus. At the end, $ss$ is printed.

```java
⟨serialTest⟩
≡
public static void serialTest () {
    //Serial verison
    DisjointSet <Long> ss = new DisjointSet <Long> ();
    System.out.println ("Serial Test");
    System.out.println ("==========");
    ss.makeSet((long)-1);
    ss.makeSet((long)0);
    ss.makeSet((long)1);
    for (int i = 0; i < SIZE; i++) {
        ss.makeSet(data[i]);
        ss.union(data[i] % 2, data[i]);
    }
    print (ss);
}
```

Uses $SIZE$ 6 and $data$ 6.
This code is used in chunk 6.
In concurrentTest, it does the same task as in serialTest, except that each makeSet and union methods are called inside a thread, so that they are running concurrently. Array t stores references to all the threads created.

\[
\text{⟨concurrentTest 9⟩ ≡ } \text{public static void concurrentTest () {}
    \text{Test test = new Test();}
    \text{Thread t[] = new Thread[SIZE];}
    \text{System.out.println ("Concurrent Test");}
    \text{System.out.println ("=“=”");}
    \text{cs.makeSet((long)-1);}
    \text{cs.makeSet((long)0);}
    \text{cs.makeSet((long)1);}
    \text{for (int i = 0; i < SIZE; i++) {}
        \text{t[i] = new Thread (test);}
        \text{t[i].start();}
    \} }
    \text{for (int i = 0; i < SIZE; i++) {}
        \text{while (t[i].isAlive()) {}}
    \} }
    \text{print (cs);}
\text{}}
\]

Uses SIZE 6 and cs 6.
This code is used in chunk 6.

Once the method start() is called, a thread is created. According to the interface of Thread class, run() method is called automatically once started. Note that each thread is delayed 1 milisecond after union(), so that other threads are allowed to be executed at this point, or guarantee their concurrency.

\[
\text{⟨run 10⟩ ≡ } \text{public void run() {}
    \text{for (int i = 0; i < SIZE; i++) {}
        \text{cs.makeSet(data[i]);}
        \text{cs.union(data[i] % 2, data[i]);}
        \text{try {}}
            \text{Thread.sleep(1);}
            \text{//Sleep for 1 miliseconds to allow other threads continue.}
        \text{catch (InterruptedException e) {}}
    \} }
\text{}}
\]

Uses SIZE 6, cs 6, and data 6.
This code is used in chunk 6.
Both `serialTest` and `concurrentTest` print their `DisjointSet` object at the end.

```java
public static void print (DisjointSet <Long> s) {
    System.out.println("Group 1 - Negative Odd Numbers:");
    for (long l : s.getMap().keySet()) {
        if (s.findSet(l) == -1) System.out.print(l + " ");
    }
    System.out.println("\n");

    System.out.println("Group 2 - Even Numbers:");
    for (long l : s.getMap().keySet()) {
        if (s.findSet(l) == 0) System.out.print(l + " ");
    }
    System.out.println("\n");

    System.out.println("Group 3 - Positive Odd Numbers:");
    for (long l : s.getMap().keySet()) {
        if (s.findSet(l) == 1) System.out.print(l + " ");
    }
    System.out.println("\n");
}
```

This code is used in chunk 6.
4 Test Result

By setting \( \text{SIZE} \) to 20, 20 \text{Long} \ integers are randomly generated and to be used in both tests.

\[
\text{[wangy22@gaiden q2]@ $ \text{java Test Serial Test

\begin{verbatim}
Group 1 - Negative Odd Numbers: 
-9103770306483490189 -4232865876030345843 -1160629452687687109 -1
\end{verbatim}
\]

Group 2 - Even Numbers:
-6237872167485304708 -4964420948893066024 -1109287713991315740 -974081879987450628
-669528114487223426 -594798593157429144 0 112842269129291794 2139215297105423308
2891469594365336806 3831662765844904176 5424394867226112926 6410576364588137014
6976596177944619528 7326573195622447256 7564655870752979346

Group 3 - Positive Odd Numbers:
1 2578166436595196069 6137546356583794141

\begin{verbatim}
Group 1 - Negative Odd Numbers: 
-9103770306483490189 -4232865876030345843 -1160629452687687109 -1
\end{verbatim}
\]

Group 2 - Even Numbers:
-6237872167485304708 -4964420948893066024 -1109287713991315740 -974081879987450628
-669528114487223426 -594798593157429144 0 112842269129291794 2139215297105423308
2891469594365336806 3831662765844904176 5424394867226112926 6410576364588137014
6976596177944619528 7326573195622447256 7564655870752979346

Group 3 - Positive Odd Numbers:
1 2578166436595196069 6137546356583794141

As we can see, random \text{Long} \ integers are grouped correctly in \text{concurrentTest}, with respect to the output of \text{serialTest}.

APPENDICES

A Defined Chunks

\( (\text{DisjointSet 1}) \ 1 \)
\( (\text{Test 6}) \ 6 \)
\( (\text{concurrentTest 9}) \ 6, 9 \)
\( (\text{findSet 4}) \ 1, 4 \)
B Index

Parent: 1, 1, 2, 3, 4, 1, 5
SIZE: 6, 6, 7, 8, 9, 10, 6
cs: 6, 9, 10
data: 6, 6, 7, 6, 8, 10
lock: 1, 1, 2, 3

References
