HW3 : SUBHAJIT SAHU : 2018801013

BITONIC COUNTING NETWORK

Bitonic network is a balanced counting network that allows processes to decompose operations, like counting, and reduce memory contention.

Balancer:

A Balancer is a simple switch with two input and two output wires. Tokens arrive on the Balancer's input wires at arbitrary times, and emerge on their output wires, at some later. Given a stream of input tokens, it sends one token to the top output wire, and the next to the bottom, and so on, effectively balancing the number of tokens between the two wires.

Balancer.traverse():

1. Flip balancer state.

2. Return the updated state.

MergerNetwork:

Each Merger[2K] consists of 2 Merger[K] networks followed by a layer of Balancers. The top Merger[K] takes even inputs from top-half and odd inputs from bottom-half. The bottom Merger[K] takes odd inputs from top-half and even inputs form bottom half. The respective output of each Merger[K] is then combined with a Balancer. When K is 1, Merger[2K] is a single Balancer.

MergerNetwork.traverse():

1. Find connected Merger[K].

2. Traverse connected Merger[K].

3. Traverse connected balancer.

BitonicNetwork:

Each Bitonic[2K] consists of 2 Bitonic[K] networks followed by a Merger[2K]. The top Bitonic[K] is connected to top-half inputs, and bottom Bitonic[K] is connected to bottom-half inputs. Outputs of both Bitonic[K] are connected directly to Merger[2K]. Bitonic[2] networks consists of a single Balancer.

BitonicNetwork.traverse():

1. Find connected Bitonic[K].
2. Traverse connected Bitonic[K].
3. Traverse connected Merger[2K].

CODE: https://repl.it/@wolfram77/bitonic-network#CountingNetwork.java

```
interface CountingNetwork {
int traverse(int x);
}
```

CODE: https://repl.it/@wolfram77/bitonic-network#Balancer.java

```
// A Balancer is a simple switch with two input
// and two output wires. Tokens arrive on the
// Balancer's input wires at arbitrary times,
// and emerge on their output wires, at some later
// Given a stream of input tokens, it sends one
// token to the top output wire, and the next to
// the bottom, and so on, effectively balancing the
// number of tokens between the two wires.
class Balancer implements CountingNetwork {
int state = 1;
// state: previous state (0=up, 1=down)
// 1. Flip balancer state.
// 2. Return the updated state.
@Override
public synchronized int traverse(int x) {
state = 1-state; // 1
return state; // 2
}
}
```

CODE: https://repl.it/@wolfram77/bitonic-network#MergerNetwork.java

```
// Each Merger[2K] consists of 2 Merger[K] networks
// followed by a layer of Balancers. The top
// Merger[K] takes even inputs from top-half and
// odd inputs from bottom-half. The bottom
// Merger[K] takes odd inputs from top-half and
// even inputs form bottom half. The respective
// output of each Merger[K] is then combined with
// a Balancer. When K is 1, Merger[2K] is a single
// Balancer.
class MergerNetwork implements CountingNetwork {
CountingNetwork[] halves;
CountingNetwork[] balancers;
final int width;
// halves: top, bottom Merger[K]
// balancers: layers of Balancers
// width: number of inputs/outputs


public MergerNetwork(int w) {
if (w > 2) halves = new MergerNetwork[] {
new MergerNetwork(w/2),
new MergerNetwork(w/2)
};
balancers = new Balancer[w/2];
for (int i=0; i<w/2; i++)
balancers[i] = new Balancer();
width = w;
}


// 1. Find connected Merger[K].
// 2. Traverse connected Merger[K].
// 3. Traverse connected balancer.
@Override
public int traverse(int x) {
int w = width;
int h = x < w/2? x%2 : 1 - x%2; // 1
if (w <= 2) x = x/2; // 1
else x = halves[h].traverse(x/2); // 2
return x*2 + balancers[x].traverse(0); // 3
}
}
```

CODE: https://repl.it/@wolfram77/bitonic-network#BitonicNetwork.java

```
// Each Bitonic[2K] consists of 2 Bitonic[K]
// networks followed by a Merger[2K]. The top
// Bitonic[K] is connected to top-half inputs, and
// bottom Bitonic[K] is connected to bottom-half
// inputs. Outputs of both Bitonic[K] are connected
// directly to Merger[2K]. Bitonic[2] networks
// consists of a single Balancer.
class BitonicNetwork implements CountingNetwork {
CountingNetwork[] halves;
CountingNetwork merger;
final int width;
// halves: top, bottom Bitonic[K]
// merger: Merger[2K] connected to both Bitonic[K]
// width: number of inputs/outputs


public BitonicNetwork(int w) {
if (w > 2) halves = new BitonicNetwork[] {
new BitonicNetwork(w/2),
new BitonicNetwork(w/2)
};
merger = new MergerNetwork(w);
width = w;
}


// 1. Find connected Bitonic[K].
// 2. Traverse connected Bitonic[K].
// 3. Traverse connected Merger[2K].
@Override
public int traverse(int x) {
int w = width;
int h = x / (w/2); // 1
if (w > 2) { // 2
x = halves[h].traverse(x % (w/2)); // 2
x = x + h * (w/2); // 2
}
return merger.traverse(x); // 3
}
}
```

CODE: https://repl.it/@wolfram77/bitonic-network#Main.java

```
import java.util.*;
import java.util.concurrent.atomic.*;
```

```
class Main {
static CountingNetwork bitonic;
static AtomicInteger[] counts;
static int WIDTH = 16;
static int OPS = 1000;
// bitonic: bitonic counting network of WIDTH
// counts: atomic integers incremented by threads
// WIDTH: number of threads / width of network
// OPS: number of increments


// Each unbalanced thread tries to increment a
// random count. At the end, the counts would
// not be balanced.
//
// Each balanced thread tries to increment a
// random count, but this time, selected through
// a bitonic network. At the end, the counts
// should be balanced.
static Thread thread(int id, boolean balance) {
return new Thread(() -> {
for (int i=0; i<OPS; i++) {
int c = (int) (WIDTH * Math.random());
if (balance) c = bitonic.traverse(c);
counts[c].incrementAndGet();
Thread.yield();
}
log(id+": done");
});
}


// Initialize bitonic network and counts.
static void setup() {
bitonic = new BitonicNetwork(WIDTH);
counts = new AtomicInteger[WIDTH];
for (int i=0; i<WIDTH; i++)
counts[i] = new AtomicInteger(0);
}


// Test either unbalanced or balanced threads.
static void testThreads(boolean balance) {
```

```
setup();
Thread[] t = new Thread[WIDTH];
for (int i=0; i<WIDTH; i++) {
t[i] = thread(i, balance);
t[i].start();
}
try {
for (int i=0; i<WIDTH; i++)
t[i].join();
}
catch(InterruptedException e) {}
}


// Check if counts are balanced. At maximum
// counts should be separated by 1.
static boolean isBalanced() {
int v = counts[0].get();
for (int i=0; i<WIDTH; i++)
if (v-counts[i].get() > 1) return false;
return true;
}


// Test both unbalanced and balanced threads
// to check if counts stay balanced after they
// run their increments.
public static void main(String[] args) {
log("Starting unbalanced threads ...");
testThreads(false);
log(Arrays.deepToString(counts));
log("Counts balanced? "+isBalanced());
log("");
log("Starting balanced threads ...");
testThreads(true);
log(Arrays.deepToString(counts));
log("Counts balanced? "+isBalanced());
}


static void log(String x) {
System.out.println(x);
}
}
```

OUTPUT: https://bitonic-network.wolfram77.repl.run

```
Starting unbalanced threads ...
0: done
14: done
6: done
4: done
8: done
2: done
12: done
11: done
10: done
1: done
15: done
13: done
9: done
7: done
3: done
5: done
[1004, 1030, 985, 992, 989, 1007, 1008, 985, 976, 968, 1046, 1005, 1014, 1017, 994, 980]
Counts balanced? false

Starting balanced threads ...
9: done
7: done
15: done
1: done
4: done
2: done
10: done
0: done
8: done
6: done
5: done
14: done
12: done
13: done
3: done
11: done
[1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000]
Counts balanced? true
```