javaf / combining-tree

A Combining Tree is an N-ary tree of nodes, that follows software combining to reduce memory contention while updating a shared value.

repl.it/@wolfram77/combining-tree#readme.md

MIT License

0 stars 0 forks

Star Unwatch

Code Issues Pull requests Actions Projects Wiki Security

master

wolfram77 easy README ... 28 minutes ago 8

View code

README.md

A Combining Tree is an N-ary tree of nodes, that follows software combining to reduce memory contention while updating a shared value.

The shared value is placed at the root of the tree, and threads perfrom `getAndOp()` at the leaf nodes. Each leaf node handles N threads. The combined value is then propagated up the tree by an active thread. There can be several active threads, but eventually one active thread updates the root node. It then shares this message to all other threads behind it.

This was contention at a single memory location is avoided. However, i guess that such a design is normally useful only in hardware, as it becomes too slow in software. Useful for educational purposes.

```
CombiningTree.getAndOp(x, op):
Gets current value, and then updates it.
x: value to OP with, op: binary op
1. Select leaf index using thread id.
2. Perform get & op at desired leaf.
```

```
CombiningTree.getAndOp(x, op, i):
Gets current value, and then updates it.
x: value to OP with, op: binary op, i: leaf index
1. Perform get & op at desired leaf (ensure in limit).
```

```
Node.getAndOp(x, op)
Gets current value, and then updates it.
x: value to OP (accumulate), op: binary operator
1. Wait until node is free.
2. Perform get & op based on 3 possible cases.
2a. Root node
2b. Active thread (first to visit node)
2c. Passive thread (visits later)
```

```
Node.getAndOpRoot(x, op)
Performs get & op for root node.
x: value to OP (accumulate), op: binary operator
1. Get old value, by combining (a).
2. Empty the node.
3. Insert a OP x
3. Return old value.
```

```
Node.getAndOpActive(x, op)
Performs get & op for active thread.
x: value to OP (accumulate), op: binary operator
1. Insert value.
2. Wait until node is full, or timeout.
3. We have the values, so start pushing.
4. Combine values into one with OP.
5. Push combined value to parent.
6. Distribute recieved value for all threads.
7. Start the pulling process.
8. Decrement count (we have our pulled value).
9. Return pulled value.
```

```
Node.getAndOpPassive(x, op)
Performs get & op for passive thread.
x: value to OP (accumulate), op: binary operator
1. Insert value.
2. Wait until active thread has pulled value.
3. Decrement count, one pulled value processed.
4. If count is 0, the node is free.
5. Return value of this thread.
```

```
## OUTPUT
3-ary 3-depth Combining tree.
Starting 25 threads doing increments ...
30: done in 51ms
14: done in 60ms
13: done in 68ms
23: done in 53ms
12: done in 56ms
21: done in 54ms
32: done in 56ms
31: done in 56ms
22: done in 54ms
28: done in 7831ms
11: done in 7836ms
10: done in 7838ms
19: done in 7833ms
20: done in 7833ms
29: done in 7831ms
24: done in 8165ms
33: done in 8160ms
25: done in 8165ms
16: done in 8167ms
34: done in 8159ms
15: done in 8167ms
26: done in 11970ms
17: done in 11972ms
27: done in 12239ms
18: done in 12241ms
Total: 2500

Was valid? true
```

See CombiningTree.java, Node.java for code, Main.java for test, and repl.it for output.

references

- The Art of Multiprocessor Programming :: Maurice Herlihy, Nir Shavit

Languages

- Java 100.0%

javaf / combining-tree

Code Issues Pull requests Actions Projects Wiki Security

master

combining-tree / Node.java / Jump to

wolfram77 programming done :) History

1 contributor

Raw Blame

213 lines (191 sloc) 6.21 KB

```
1   import java.util.function.*;
2
3   // N-ary Node is the basic block of an N-ary Combining tree.
4   // It has N places for values, allowing N threads to put in
5   // their value. The first thread to put its value in the node
6   // becomes the "active thread", while all others become
7   // "passive threads". Once a node is full (or timeout), the
8   // active thread combines the node values and pushes it to
9   // the parent node. The value it recieves back from parent
10  // node is the distributed among all threads with proper
11  // values. Passive threads just wait for these values to be
12  // distributed, and take the one for them.
13
14  class Node<T> {
15    static final int FREE = 0;
16    static final int PUSHING = 1;
17    static final int PULLING = 2;
18    static final long TIMEOUT = 100;
19    Node<T> parent;
20    int state;
21    int count;
22    T[] value;
23    int size;
24    // FREE: indicates that values are still being inserted
25    // PUSHING: node is full, combined value being pushed
26    // PULLED: pulled value from parent being given to threads
27    // TIMEOUT: max time active thread waits for node to fill
```

```
28  // parent: parent node
29  // state: either FREE, PUSHING, or PULLING
30  // count: number of values in node
31  // value: storage place for values
32  // size: max. no. of values allowed (arity of node, eg 2)
33
34
35  public Node() {
36    this(2);
37  }
38
39  @SuppressWarnings("unchecked")
40  public Node(int n) {
41    value = (T[]) new Object[n];
42    parent = null;
43    state = FREE;
44    count = 0;
45    size = n;
46  }
47
48
49  // Gets node value (only for root node).
50  public synchronized T get() {
51    return value[0];
52  }
53
54  // Sets node value (only for root node).
55  public synchronized void set(T x) {
56    value[0] = x;
57    count = 1;
58  }
59
60
61  // Gets current value, and then updates it.
62  // x: value to OP (accumulate), op: binary operator
63  // 1. Wait until node is free.
64  // 2. Perform get & op based on 3 possible cases.
65  // 2a. Root node
66  // 2b. Active thread (first to visit node)
67  // 2c. Passive thread (visits later)
68  public synchronized T getAndOp(T x,
69  BinaryOperator<T> op)
70  throws InterruptedException {
71    while (state!=FREE || count==size) wait(); // 1
72    if (parent==null) return getAndOpRoot(x, op); // 2a
73    if (count==0) return getAndOpActive(x, op);   // 2b
74    return getAndOpPassive(x, op); // 2c
75  }
```

```
76
77    // Performs get & op for root node.
78    // x: value to OP (accumulate), op: binary operator
79    // 1. Get old value, by combining (a).
80    // 2. Empty the node.
81    // 3. Insert a OP x
82    // 3. Return old value.
83    private synchronized T getAndOpRoot(T x,
84    BinaryOperator<T> op)
85    throws InterruptedException {
86      T a = combine(op);
87      count = 0;
88      insert(op.apply(a, x));
89      return a;
90    }
91
92    // Performs get & op for active thread.
93    // x: value to OP (accumulate), op: binary operator
94    // 1. Insert value.
95    // 2. Wait until node is full, or timeout.
96    // 3. We have the values, so start pushing.
97    // 4. Combine values into one with OP.
98    // 5. Push combined value to parent.
99    // 6. Distribute recieved value for all threads.
100   // 7. Start the pulling process.
101   // 8. Decrement count (we have our pulled value).
102   // 9. Return pulled value.
103   private synchronized T getAndOpActive(T x,
104   BinaryOperator<T> op)
105   throws InterruptedException {
106     insert(x); // 1
107     waitUntilFull(TIMEOUT); // 2
108     state = PUSHING;   // 3
109     T a = combine(op); // 4
110     T r = parent.getAndOp(a, op); // 5
111     distribute(r, op); // 6
112     state = PULLING;   // 7
113     notifyAll();       // 7
114     decrementCount();  // 8
115     return r; // 9
116   }
117
118   // Performs get & op for passive thread.
119   // x: value to OP (accumulate), op: binary operator
120   // 1. Insert value.
121   // 2. Wait until active thread has pulled value.
122   // 3. Decrement count, one pulled value processed.
123   // 4. If count is 0, the node is free.
```

```
124   // 5. Return value of this thread.
125   private synchronized T getAndOpPassive(T x,
126   BinaryOperator<T> op)
127   throws InterruptedException {
128     int i = insert(x); // 1
129     while (state!=PULLING) wait(); // 2
130     decrementCount(); // 3, 4
131     return value[i]; // 5
132   }
133
134
135   // Inserts a value in the node (for a thread).
136   // x: value to insert
137   // 1. Wait unit node is free.
138   // 2. Get index to place value in.
139   // 3. Place the value.
140   // 4. Increment number of values in node.
141   // 5. If node is full, notify active thread.
142   // 6. Return index where value was placed.
143   public synchronized int insert(T x)
144   throws InterruptedException {
145     while (state!=FREE) wait(); // 1
146     int i = count;    // 2
147     value[i] = x;     // 3
148     incrementCount(); // 4, 5
149     return i; // 6
150   }
151
152
153   // Combine all values in the node (put by threads).
154   // op: binary operator
155   // 1. If OP is "+", combine will return sum.
156   public synchronized T combine(BinaryOperator<T> op) {
157     T a = value[0]; // 1
158     for (int i=1; i<count; i++)  // 1
159       a = op.apply(a, value[i]); // 1
160     return a; // 1
161   }
162
163
164   // Distribute pulled value to all threads.
165   // r: pulled value, op: binary operator
166   // T0: active thread, T1...: passive threads
167   // 1. T0 receives r
168   // 2. T1 recieves r OP v0.
169   // 3. T2 recieves r OP v0 OP v1 ...
170   public synchronized void distribute(T r,
171   BinaryOperator<T> op) {
```

```
172       for (int i=0; i<count; i++) { // 1
173         T x = value[i];             // 1
174         value[i] = r;               // 1
175         r = op.apply(r, x);  // 2
176       } // 3
177     }
178
179     // Increment count once done with insertion.
180     // 1. Increment count.
181     // 2. If node is full, notify active thread.
182     private synchronized void incrementCount() {
183       if (++count<size) return; // 1
184       notifyAll(); // 2
185     }
186
187     // Decrement count once done with distribution.
188     // 1. Decrement count.
189     // 2. If count is zero, node is free.
190     private synchronized void decrementCount() {
191       if (--count>0) return; // 1
192       state = FREE; // 2
193       notifyAll();  // 2
194     }
195
196     // Wait until node is full, or timeout.
197     // 1. Get start time.
198     // 2. If node is full, exit.
199     // 3. Otherwise, wait with timeout.
200     // 4. On waking up, check current time.
201     // 5. Reduce timeout by the elapsed time.
202     // 6. If timeout done, exit (else retry 2).
203     private void waitUntilFull(long w)
204     throws InterruptedException {
205       long t0 = System.currentTimeMillis(); //1
206       while (count < size) { // 2
207         wait(w);             // 3
208         long t = System.currentTimeMillis(); // 4
209         w -= t - t0;     // 5
210         if (w<=0) break; // 6
211       }
212     }
213   }
```

javaf / combining-tree

Code Issues Pull requests Actions Projects Wiki Security

master

combining-tree / CombiningTree.java / Jump to

wolfram77 programming done :) History

1 contributor

Raw Blame

82 lines (69 sloc) 2.25 KB

```
1  import java.util.function.*;
2
3
4  // A Combining Tree is an N-ary tree of nodes, that follows
5  // software combining to reduce memory contention while
6  // updating a shared value.
7  //
8  // The shared value is placed at the root of the tree, and
9  // threads perfrom getAndOp() at the leaf nodes. Each leaf
10 // node handles N threads. The combined value is then
11 // propagated up the tree by an active thread. There can be
12 // several active threads, but eventually one active thread
13 // updates the root node. It then shares this message to all
14 // other threads behind it.
15 //
16 // This was contention at a single memory location is avoided.
17 // However, i guess that such a design is normally useful
18 // only in hardware, as it becomes too slow in software.
19 // Useful for educational purposes.
20
21 class CombiningTree<T> {
22   Node<T> root;
23   Node<T>[] leaf;
24   // root: root node
25   // leaf: leaf nodes
26
27
```

```
28  public CombiningTree(int depth) {
29    this(depth, 2);
30  }
31
32  public CombiningTree(int depth, int ary) {
33    Node<T>[] parent = createNodes(1, ary);
34    root = parent[0];
35    leaf = parent;
36    for (int i=1; i<depth; i++) {
37      int n = (int) Math.pow(ary, i);
38      leaf = createNodes(n, ary);
39      for (int j=0; j<n; j++)
40        leaf[j].parent = parent[j/ary];
41      parent = leaf;
42    }
43  }
44
45  @SuppressWarnings("unchecked")
46  private Node<T>[] createNodes(int n, int ary) {
47    Node<T>[] a = (Node<T>[]) new Node[n];
48    for (int i=0; i<n; i++)
49      a[i] = new Node<>(ary);
50    return a;
51  }
52
53
54  // Get value of tree (at root).
55  public T get() {
56    return root.get();
57  }
58
59  // Set value of tree (at root).
60  public void set(T x) {
61    root.set(x);
62  }
63
64
65  // Gets current value, and then updates it.
66  // x: value to OP with, op: binary op
67  // 1. Select leaf index using thread id.
68  // 2. Perform get & op at desired leaf.
69  public T getAndOp(T x, BinaryOperator<T> op)
70  throws InterruptedException {
71    int i = (int) Thread.currentThread().getId(); // 1
72    return getAndOp(x, op, i); // 2
73  }
74
75  // Gets current value, and then updates it.
```

```
76    // x: value to OP with, op: binary op, i: leaf index
77    // 1. Perform get & op at desired leaf (ensure in limit).
78    public T getAndOp(T x, BinaryOperator<T> op, int i)
79    throws InterruptedException {
80      return leaf[i % leaf.length].getAndOp(x, op); // 1
81    }
82  }
```

javaf / combining-tree

Code Issues Pull requests Actions Projects Wiki Security

master

combining-tree / Main.java / Jump to

wolfram77 programming done :)

History

1 contributor

Raw Blame

100 lines (85 sloc) 2.49 KB

```
1  import java.util.*;
2  import java.util.concurrent.*;
3
4
5  class Main {
6    static Queue<Integer> queue;
7    static CombiningTree<Integer> tree;
8    static int TH = 25, ARY = 3, NUM = 100;
9    // queue: used to store old (get) values (unique check)
10   // tree: combining tree where threads do increment ops
11   // TH: number of threads
12   // ARY: arity of combining tree
13   // NUM: number of increment ops each thread performs
14
15
16   // Each thread performs increment op using the combining
17   // tree, and saves old (get) values in the queue, for
18   // uniqueness check later.
19   static Thread thread(int id) {
20     return new Thread(() -> {
21       try {
22       long start = System.currentTimeMillis();
23       for (int i=0; i<NUM; i++) {
24         Integer r = tree.getAndOp(1,
25           (Integer x, Integer y) -> x + y);
26         queue.add(r);
27         Thread.yield();
```

```
      }
      long stop = System.currentTimeMillis();
      log(id()+": done in "+(stop-start)+"ms");
      } catch (InterruptedException e) {}
    });
  }

  // Check if total sum is as expected.
  // Check if all old values are unique.
  static boolean wasValid() {
    int a = tree.get().intValue();
    if (a != TH*NUM) return false;
    Set<Integer> s = new HashSet<>();
    while (queue.size()>0) {
      Integer n = queue.remove();
      if (s.contains(n)) return false;
      s.add(n);
    }
    return true;
  }

  // Setup the combining tree for threads.
  static void setupTreeAndQueue() {
    int depth = (int) Math.ceil(Math.log(TH)/Math.log(ARY));
    tree = new CombiningTree<>(depth, ARY);
    tree.set(0);
  }

  // Setup the queue for storing old values.
  static void setupQueue() {
    queue = new ConcurrentLinkedQueue<>();
  }


  // Start threads doing increments using tree.
  static Thread[] startOps() {
    Thread[] t = new Thread[TH];
    for (int i=0; i<TH; i++)
      t[i] = thread(i);
    for (int i=0; i<TH; i++)
      t[i].start();
    return t;
  }

  // Wait until all threads done with increments.
  static void awaitOps(Thread[] t) {
    try {
    for (int i=0; i<TH; i++)
```

```
76        t[i].join();
77      } catch (InterruptedException e) {}
78    }
79
80
81    public static void main(String[] args) {
82      setupTree();
83      setupQueue();
84      log(ARY+"-ary "+depth+"-depth Combining tree.");
85      log("Starting "+TH+" threads doing increments ...");
86      Thread[] t = startOps();
87      awaitOps(t);
88      log("Total: "+tree.get());
89      log("\nWas valid? "+wasValid());
90    }
91
92
93    static void log(String x) {
94      System.out.println(x);
95    }
96
97    static long id() {
98      return Thread.currentThread().getId();
99    }
100 }
```