Worksheet: Sequential->Parallel Spanning Tree Algorithm

Insert finish, async, and atomic (includes a compareAndSet) constructs (pseudocode is fine) to convert the sequential spanning tree algorithm to a parallel algorithm

```
1.  class V  {
2.    V [] neighbors; // adjacency list for input graph
3.    V parent; // output value of parent in spanning tree
4.
5.    boolean makeParent(V n) {
6.      if (parent == null) { parent = n; return true; }
7.      else return false; // return true if n became parent
8.    } // makeParent
9.
10.   void compute() {
11.     for (int i=0; i<neighbors.length; i++) {
12.       final V child = neighbors[i];
13.       if (child.makeParent(this))
14.         child.compute(); // recursive call
15.      }
16.   } // compute
17. } // class V
18. . . . // main program
19. root.parent = root; // Use self-cycle to identify root
20. root.compute();
21. . . .
```


Atomic Variables represent a special (and more efficient) case of object-based isolation

```
1.  class V  {
2.    V [] neighbors; // adjacency list for input graph
3.    AtomicReference<V> parent; // output value of parent in spanning tree
4.    boolean makeParent(final V n) {
5.      // compareAndSet() is a more efficient implementation of
6.      // object-based isolation
7.      return parent.compareAndSet(null, n);
8.    } // makeParent
9.    void compute() {
10.     for (int i=0; i<neighbors.length; i++) {
11.       final V child = neighbors[i];
12.       if (child.makeParent(this))
13.         async(() -> { child.compute(); }); // escaping async
14.     }
15.   } // compute
16.} // class V
17.. . .
18.root.parent = root; // Use self-cycle to identify root
19.finish(() -> { root.compute(); });
20.. . .
```