Lecture 33: Nonblocking Linked Lists

COSC 273: Parallel and Distributed Computing

Spring 2023

Annoucements

1. Quiz on concurrent linked lists due Today @ 5:00 pm
2. Next leaderboard submission on Monday

Primes Leaderboard

Baseline: 58810ms

1. Deadlock Dodgers (4358 ms)

No others were significantly faster than the baseline.

Tips from Deadlock Dodgers?

Read Documentation

→ thread pools

→ Callable (not runnable)

↪ return something

Block method → find primes in range

Task = 1 block

→ wait for task compete

→ Future interface

Notes on Space Usage

PrimeBlock : Boolean Array isPrime
starts at value start

isPrime[i] == true
if start + i is Prime

2 B. Booleans ⟹ ~ 2 B Bytes
not Bits

Idea: use bytes (8 bits)
use a single byte to encode
8 bits
byte b

first bit = b&1, second b&2,
third b&4...

Arrays.sort — "dual pivot quicksort"

Sorting Leaderboard

Baseline: 8034ms

1. Sunny Day (511ms)
2. MRC (1580ms)
3. Team 2 (2214ms)

Parallel Sort Java Built (In future, don't use — use as benchmark)

recursive quicksort, parallelize recursive calls.

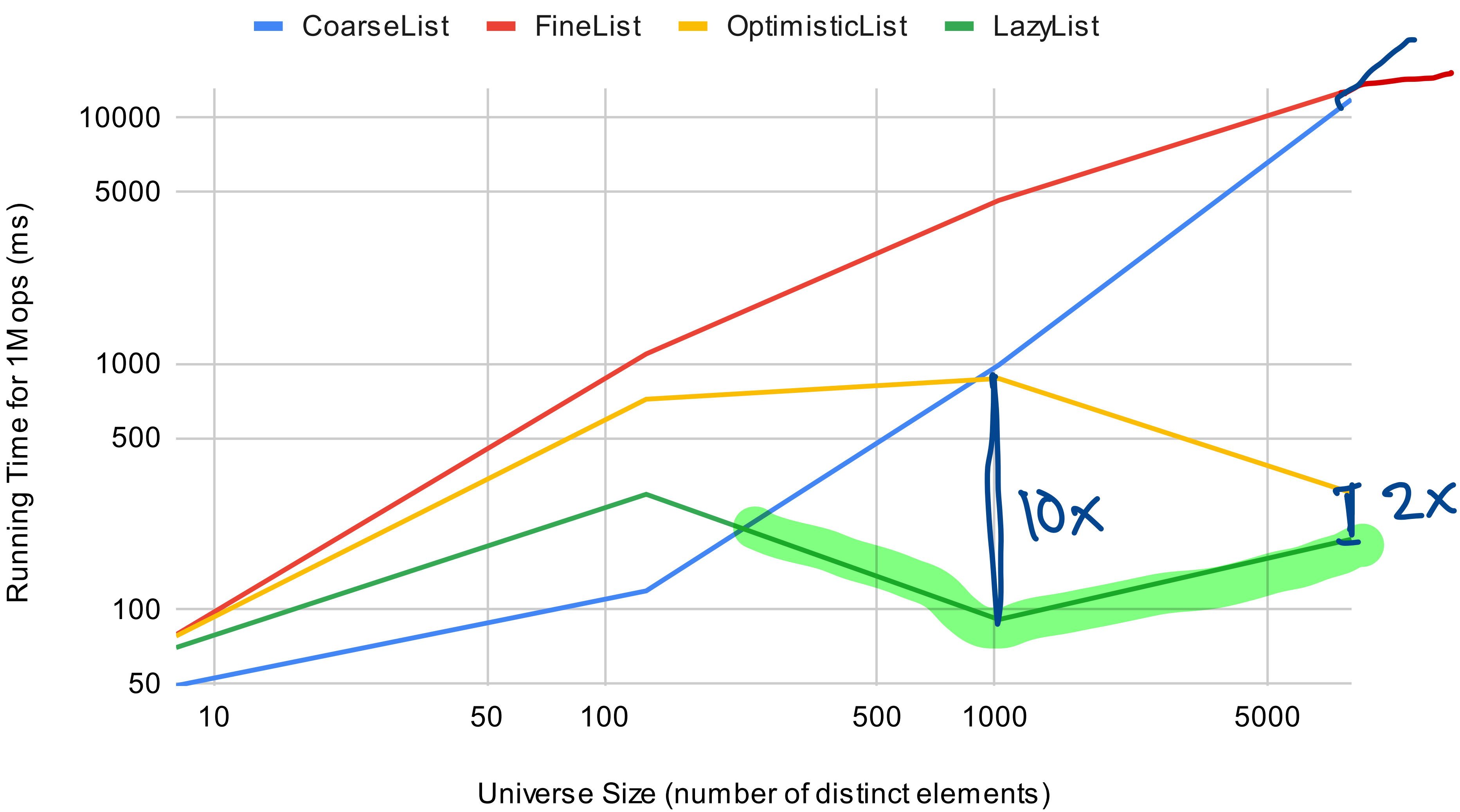
Tips from Sunny Day?

Previously

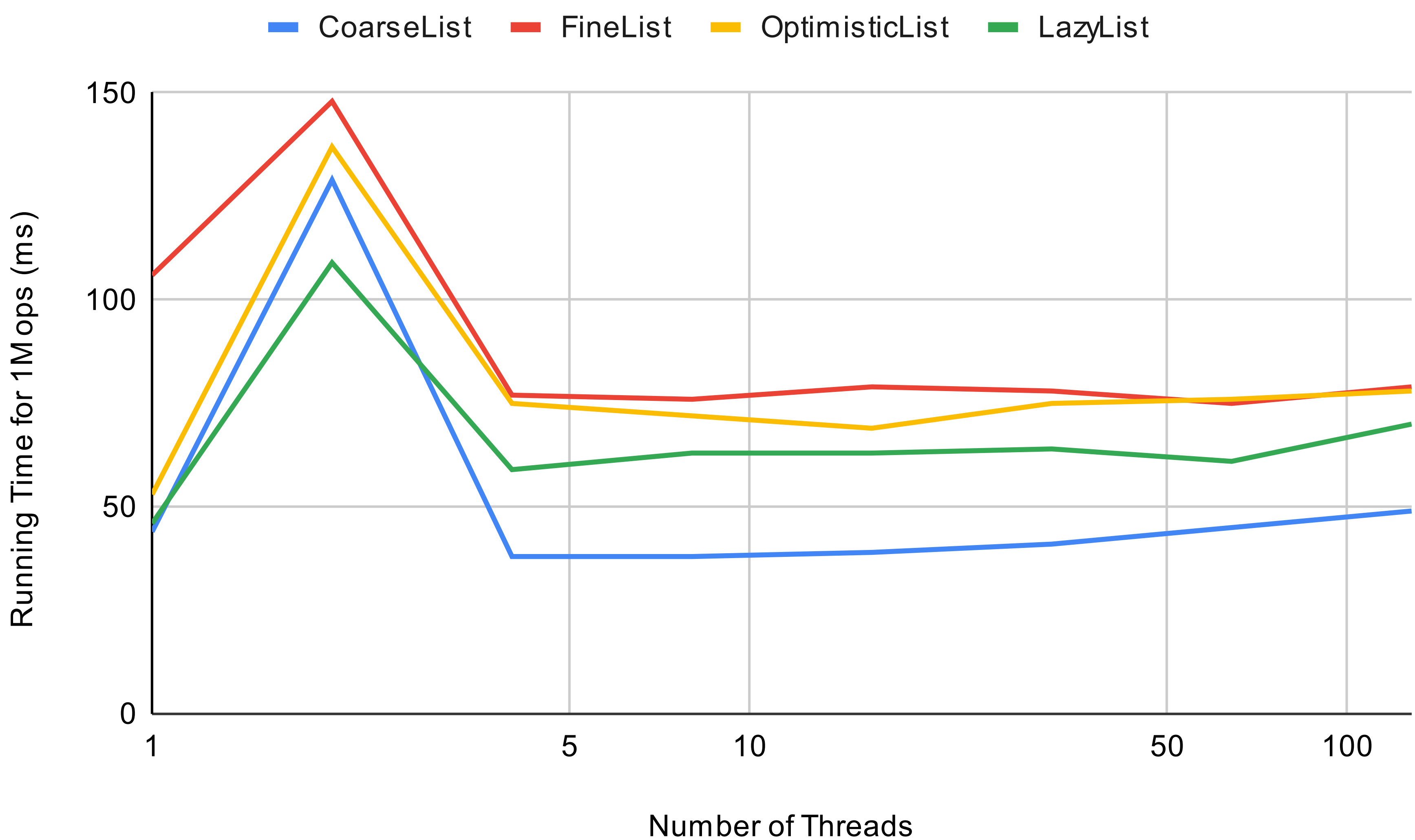
Concurrent Linked Lists, Four Ways:

1. Coarse locking
   - lock the whole data structure for every operation
2. Fine-grained locking
   - lock individual nodes to avoid conflicts
3. Optimistic locking
   - search without locks, lock on find, then validate
4. Lazy removal
   - like optimistic, but with logical removal
   - wait-free `contains` implementation!

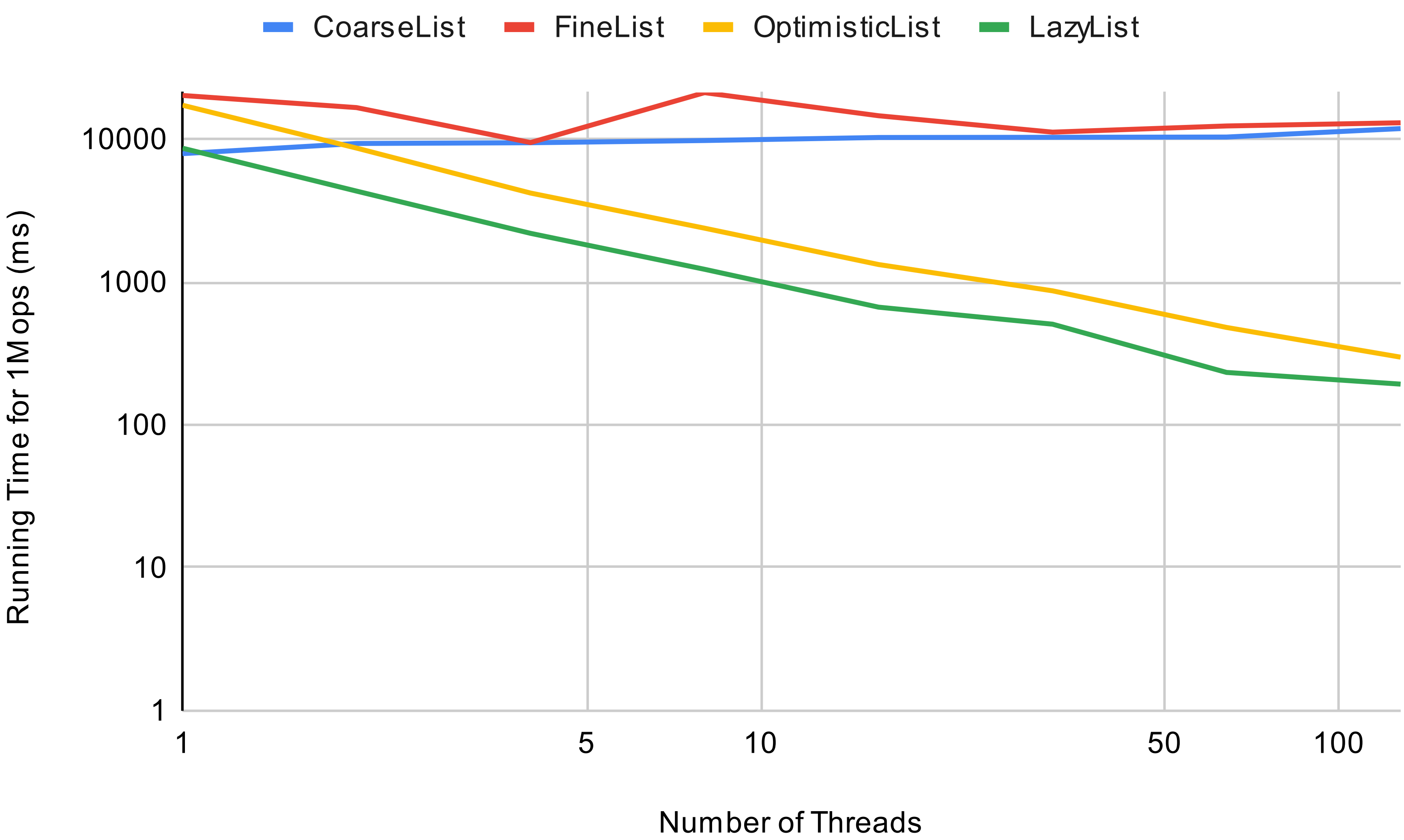
Performance v. Size, 128 Threads



Time v. Threads, 8 Elements



Time v. Threads, 8,192 Elements



Today

Nonblocking linked lists!

Question. Can we avoid locks entirely?

Lazy List and Locks

1. Traverse without locking
2. Lock relevant nodes
3. Validate list
4. Perform operation
5. Unlock nodes

C.S.

Why Does LazyList Need Locks?

Validataion:

```
private boolean validate (Node pred, Node curr) {
  return !pred.marked && !curr.marked && pred.next == curr;
}
```

Modification (e.g., add):

```
Node node = new Node(item);
node.next = curr;
pred.next = node; // this is the only step that modifies list!
```

Only modification step.

Why Does LazyList Need Locks?

Validataion:

```
private boolean validate (Node pred, Node curr) {
  return !pred.marked && !curr.marked && pred.next == curr;
}
```

Modification (e.g., add):

```
Node node = new Node(item);
node.next = curr;
pred.next = node; // this is the only step that modifies list!
```

The issue:

- Validation and modification are separate steps
- Must enforce that nodes are unchanged between validation and mod

An Idea

If we can

1. combine validation and modification steps
2. perform this operation atomically

then maybe we can avoid locking?

A Tool

Better living with atomics!

- AtomicMarkableReference<T>
- Stores
  1. a reference to a T
  2. a boolean marked
- Atomic operations
  1. boolean compareAndSet(T expectedRef, T newRef, boolean expectedMark, boolean newMark)
  2. T get(boolean[] marked)
  3. T getReference()
  4. boolean isMarked()

if(ref == expected
&& mark == expected)
ref ← newRef
marked ← newMark
return true
else
return false

An Algorithm?

Use `AtomicMarkableReference<Node>` for `Node` references

- `mark` indicates logical removal —

For `add`/`remove`:

1. Find location
2. Validate and modify
   - (first logically remove if `remove`)
   - use `compareAndSet` to atomically
     1. check that predecessor not removed (marked)
     2. update `next` field of predecessor

For `contains`:

- Just traverse the list!

if "next" is marked, then cur node is logically removed

NonblockingList Design

See NonblockingList.java

1. For Node class, AtomicMarkableReference<Node> next is marked if this Node is logically removed
    - separate logical/physical removal as in LazyList
2. Separate Window class stores two Nodes: prev, curr
3. NonblockingList method find returns a Window
    - find also removes any marked nodes encountered

find pred and curr node for a given key/value

`NonblockingList` Design

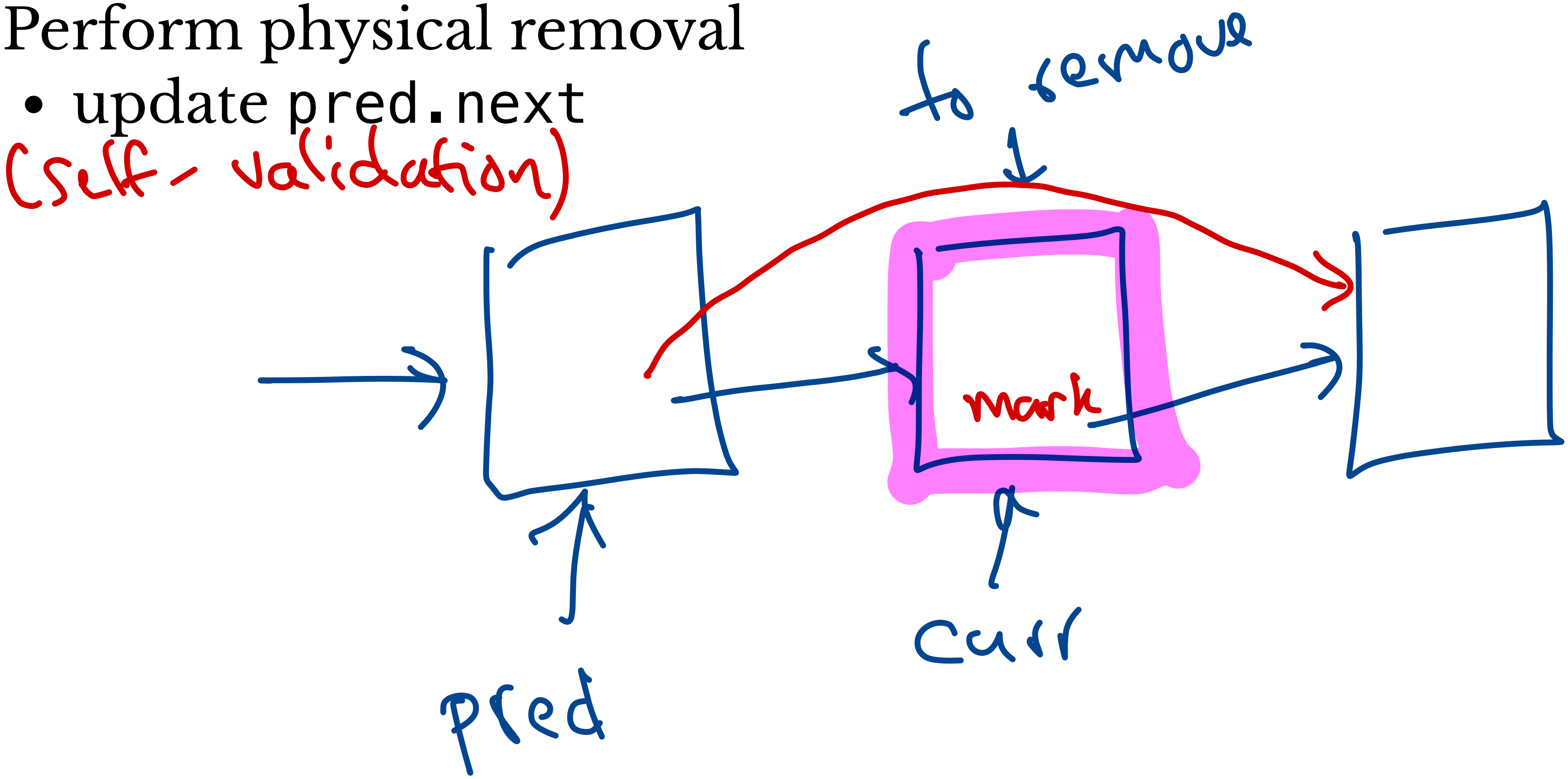
See `NonblockingList.java`

1. For `Node` class, `AtomicMarkableReference<Node> next` is marked if this `Node` is logically removed
   - separate logical/physical removal as in `LazyList`
2. Separate `Window` class stores two `Node`s: `prev`, `curr`
3. `NonblockingList` method `find` returns a `Window`
   - `find` also removes any marked nodes encountered

Question. Why should methods perform physical removal for *other* pending operations?

Removal Sketch

1. Find `Node` `curr` storing value with predecessor `pred`
2. Mark `curr` for (logical) removal
    - set mark of `cur.next` to `true`
    - retry if this fails
3. Perform physical removal
    - update `pred.next`



Removal in Code I

```
public boolean remove(T item) {
int key = item.hashCode();
boolean snip;
while (true) {
    Window window = find(head, key);
    Node pred = window.pred;
    Node curr = window.curr;
    if (curr.key != key) { return false; }
        // curr contains item
        ...
        }
}
```

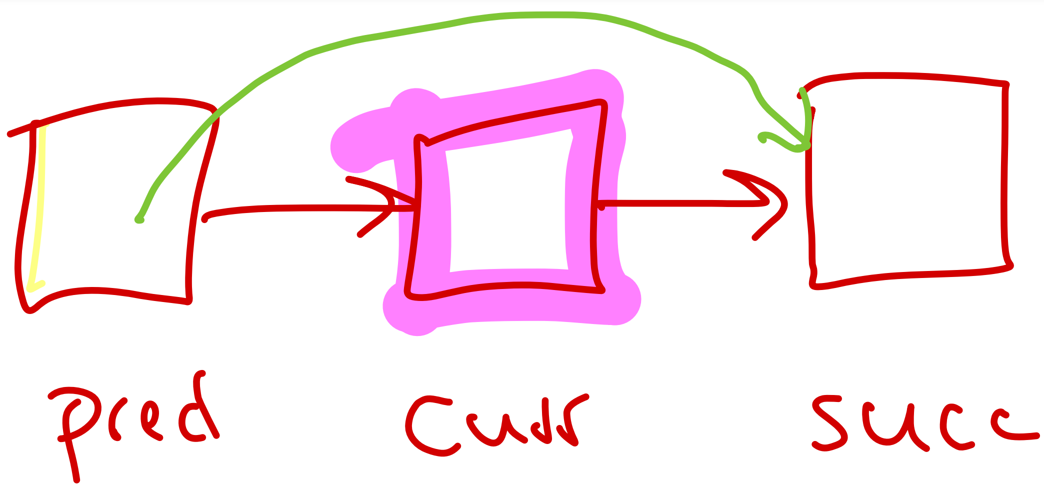
item not in list.

Removal in Code II

```java
public boolean remove(T item) {
    ...
    while (true) {
        ...
        // curr contains item
        Node succ = curr.next.getReference();
        snip = curr.next.compareAndSet(succ, succ, false, true);
        if (!snip) {continue;}
        pred.next.compareAndSet(curr, succ, false, false);
        return true;
    }
}
```

← logical removal fails

pred not removed

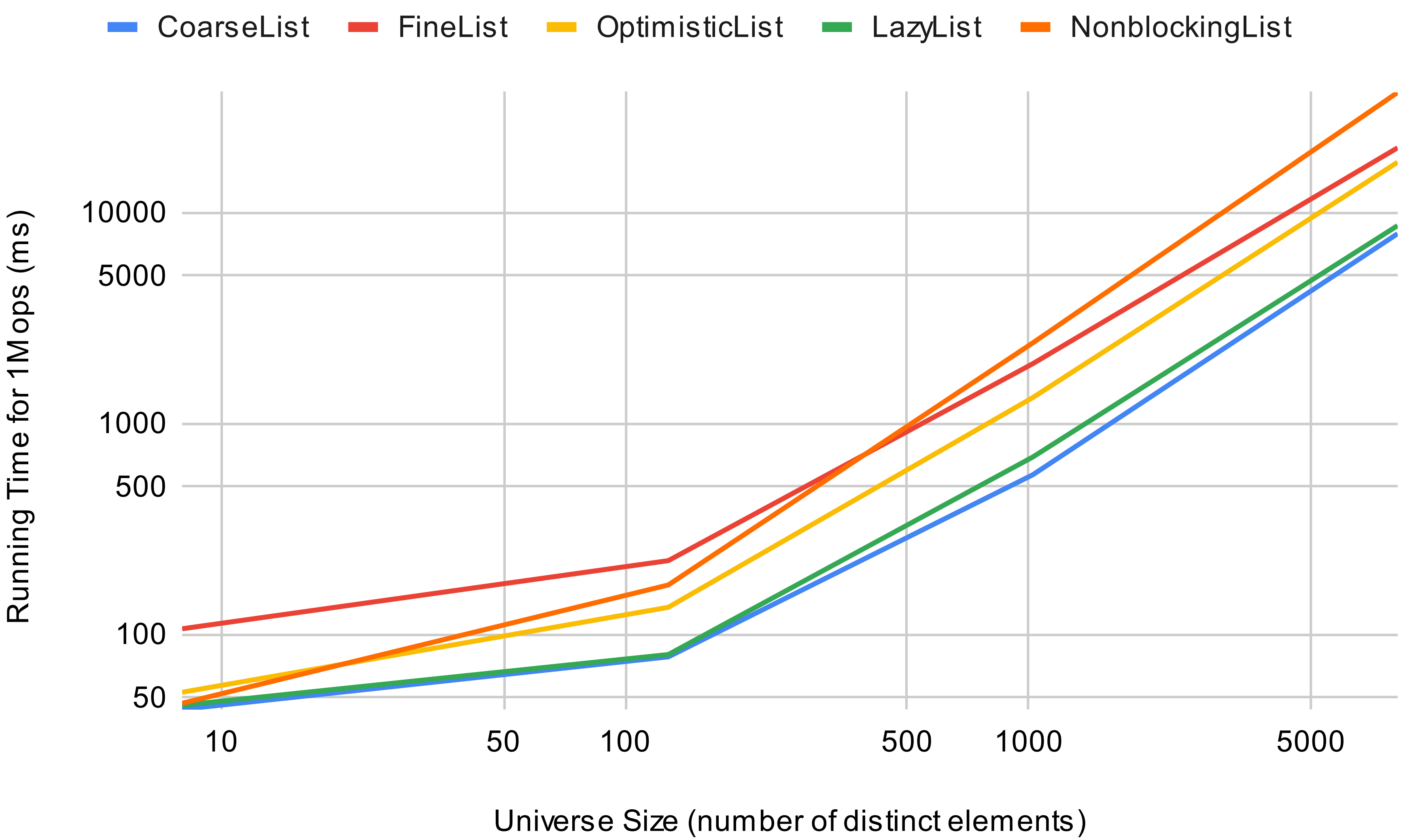


A Puzzle

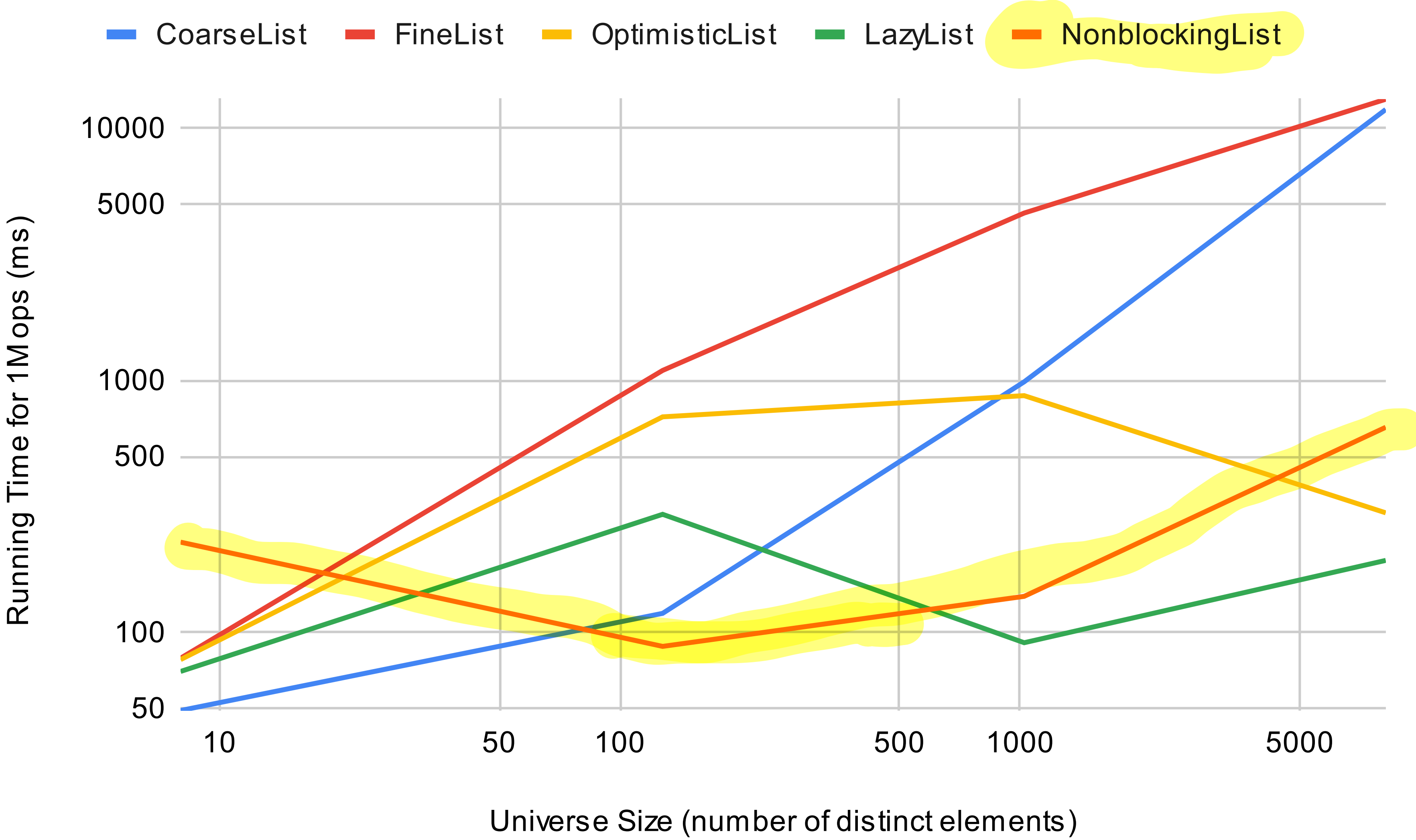
Question. Why don't we care about return value of `pred.next.compareAndSet`?

```
public boolean remove(T item) {
    while (true) {
        ...
        // curr logically removed
        pred.next.compareAndSet(curr, succ, false, false);
        return true;
    }
}
```

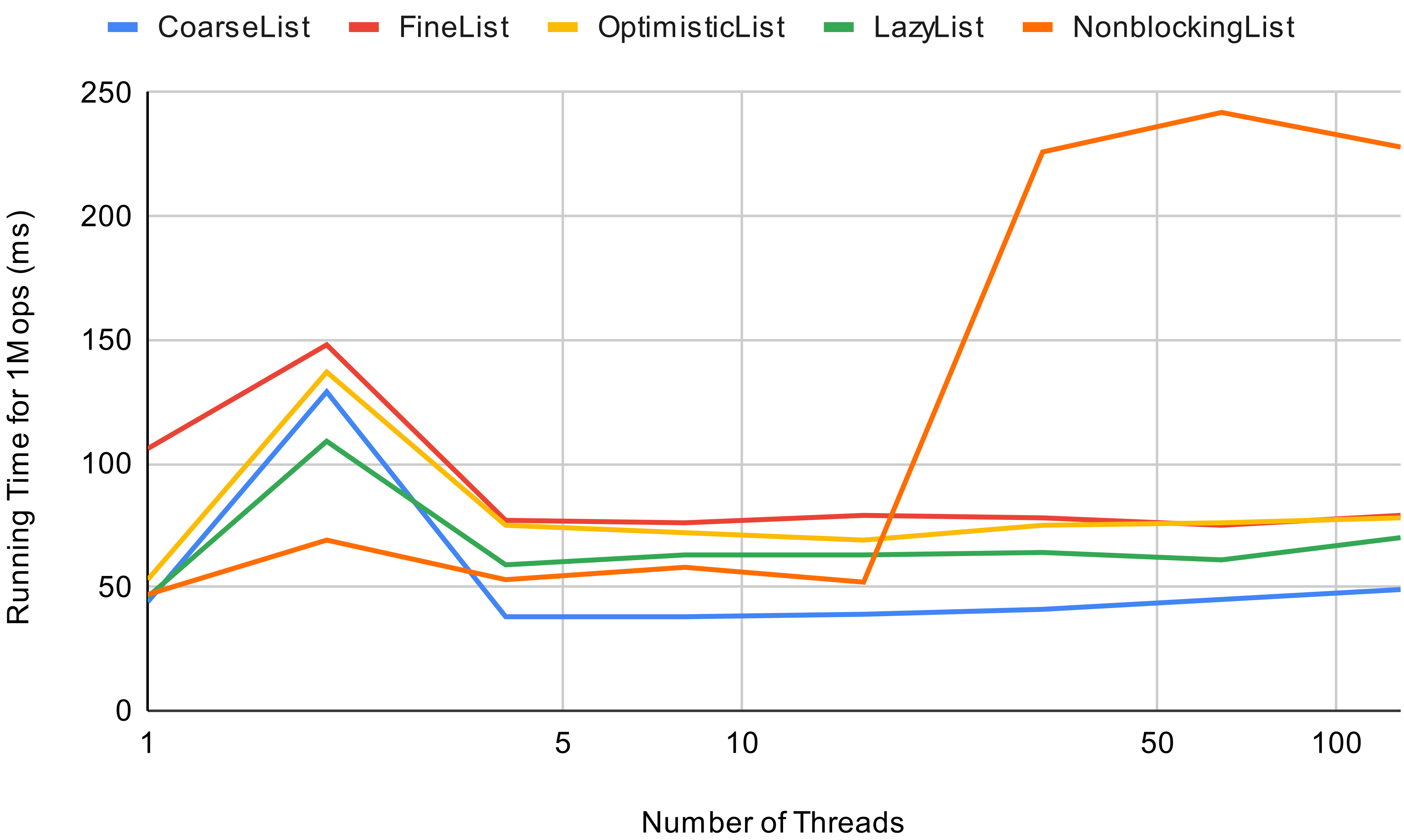
Performance v. Size, 1 Thread



Performance v. Size, 128 Threads



Time v. Threads, 8 Elements



Time v. Threads, 8,192 Elements