[bana]

Lock-free programming, implementations, and comparisons

Contents:

- Introduction
- Lock-based vs. lock-free
- Disadvantage of lock based solutions
- CAS
- Lock-free stack – memory handling
- ABA problem
- Hazard pointers, reference counting
- Concurrent sorted lists, solutions
- Performance and conclusion

Intro:

- Aims
- Multithread programming
- Dynamically shared objects
  - dynamically allocated elements in memory
  - predefined operations (in OO methods)

- Realization and investigation:
  - lock based and lock-free stack, queue, sorted list
  - various spinlocks

Lock-based vs. lock-free

- Lock-based:
  - shared data using mutex (lock/unlock)
  - in the critical section any operations can be executed

- Lock-free:
  - no locking method
  - precise, more focused and less operations on shared objects

Disadvantages of lock based solutions:

- Overhead
- Starvation
- Deadlock
- Livelock
- Priority inversion
- Convoying
- Thread-crash
- Asynchronous signal problem

CAS (compare-and-swap) atomic primitive

Pseudo code:

```
bool compare_and_swap (target, old, newval)
{
    if ( target == old ) {
        target = newval;
        return true;
    } else return false;
}
```

Lock-free stack, example:

```
procedure push(int x){
   node = new Node(x)
   do
      node->next = head
      old = head
    while(!CAS(head,old,node))
}
```

```
procedure pop(){
    do
       if(head==null) break
       old = head
    while(!CAS(head,old,old->next))
}
```

Lock-free stack – memory handling

```
procedure push(int x){
   node = new Node(x)
   do
      node->next = head
      old = head
    while(!CAS(head,old,node))
}
```

```
procedure pop(){
    do
       if(head==null) break
       old = head
    while(!CAS(head,old,old->next))
}
```

Is it good? Depends on language: GC exists => okey

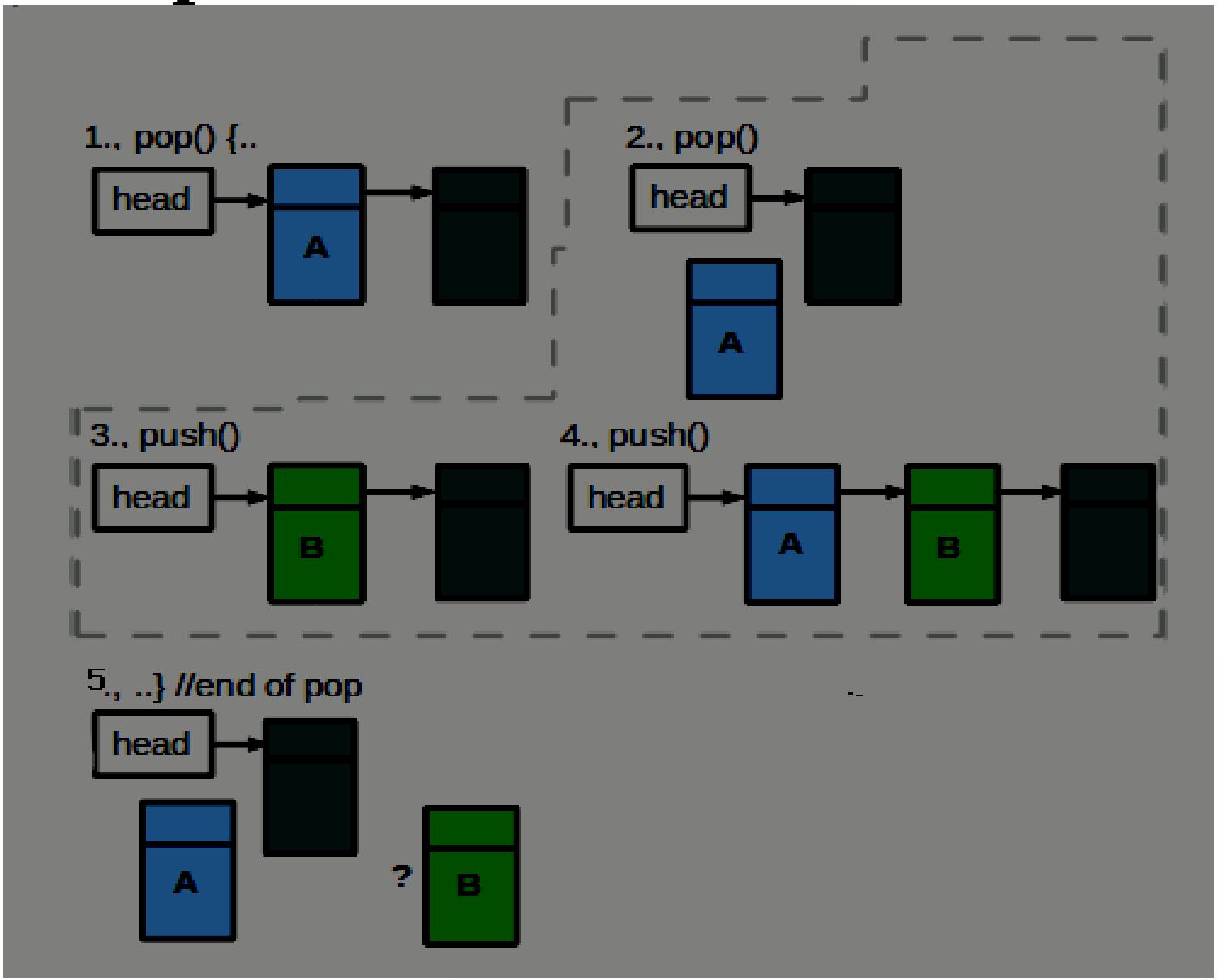
Lock-free stack – memory handling

```
procedure push(int x){
   node = new Node(x)
   do
      node->next = head
      old = head
    while(!CAS(head,old,node))
}
```

```
procedure pop(){
    do
       if(head==null) break
       old = head
    while(!CAS(head,old,old->next))
}
```

Is it good? Depends on language: GC exists => okey
When not, e.g.: in C++? Own implemented memory handler.

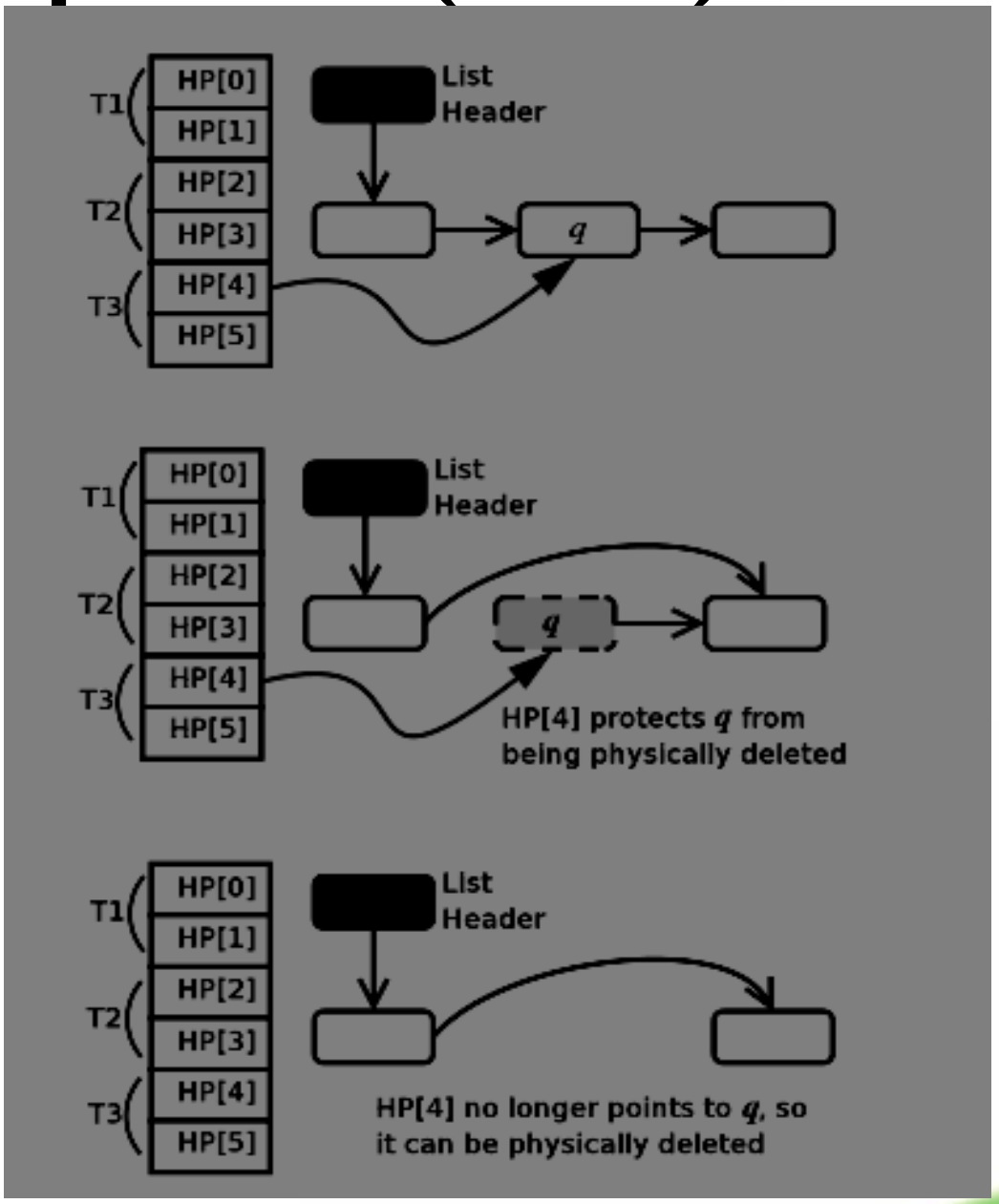
ABA problem


1., pop() {..
head
A
2., pop()
head
A
3., push()
head
B
4., push()
head
A
B
5., ..} //end of pop
head
A
?
B

Hazard pointers (SMR), reference counting

- Memory handler solutions
- Reference counting in C++: shared_ptr<>
- SMR (it’s sure lock-free!)

Hazard pointers (SMR)



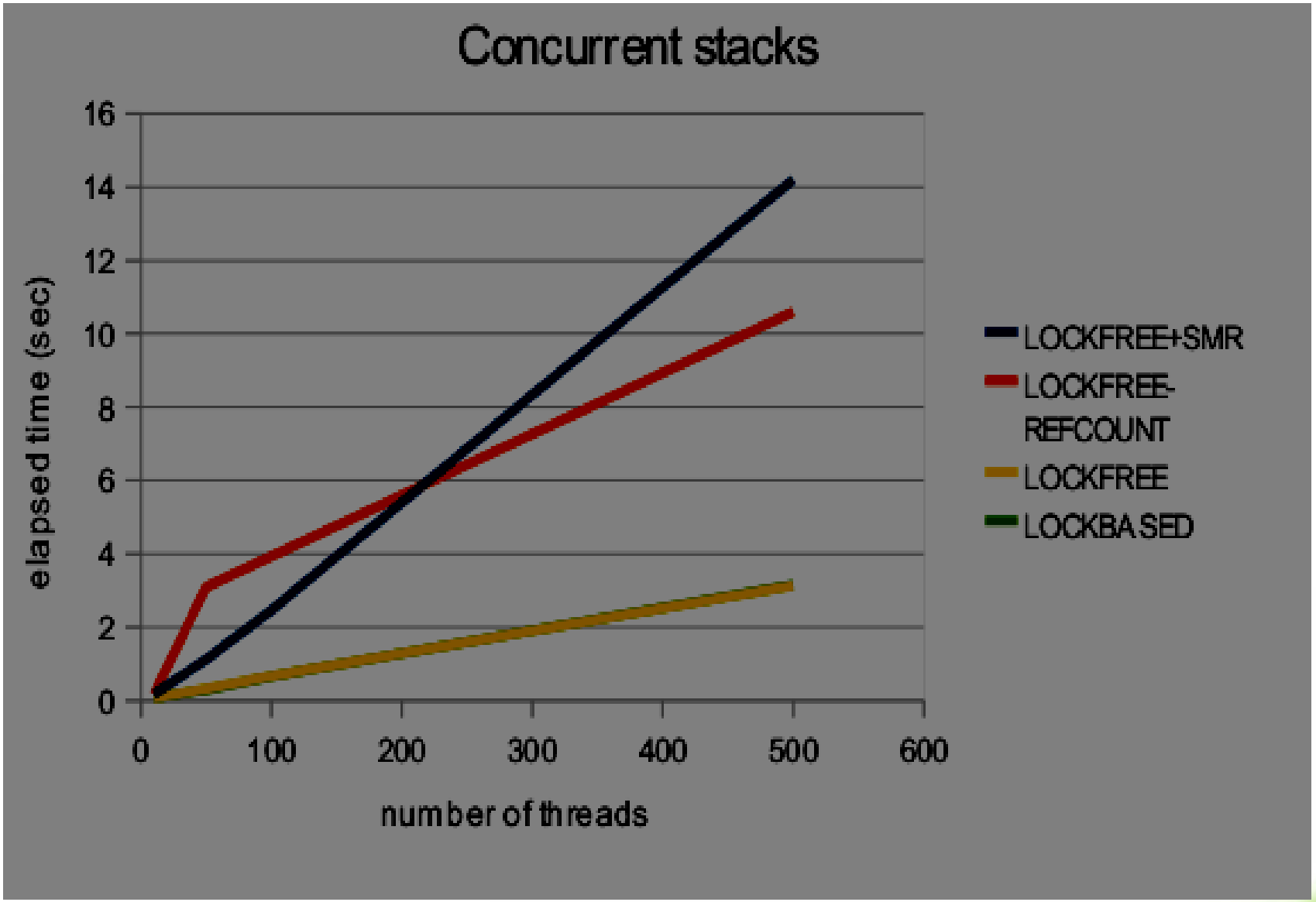
Parameters to testings:

- Keys: [0..2000]
- 2 tests:
  - 10.000 operations
  - (50% add, 25% search 25% remove)

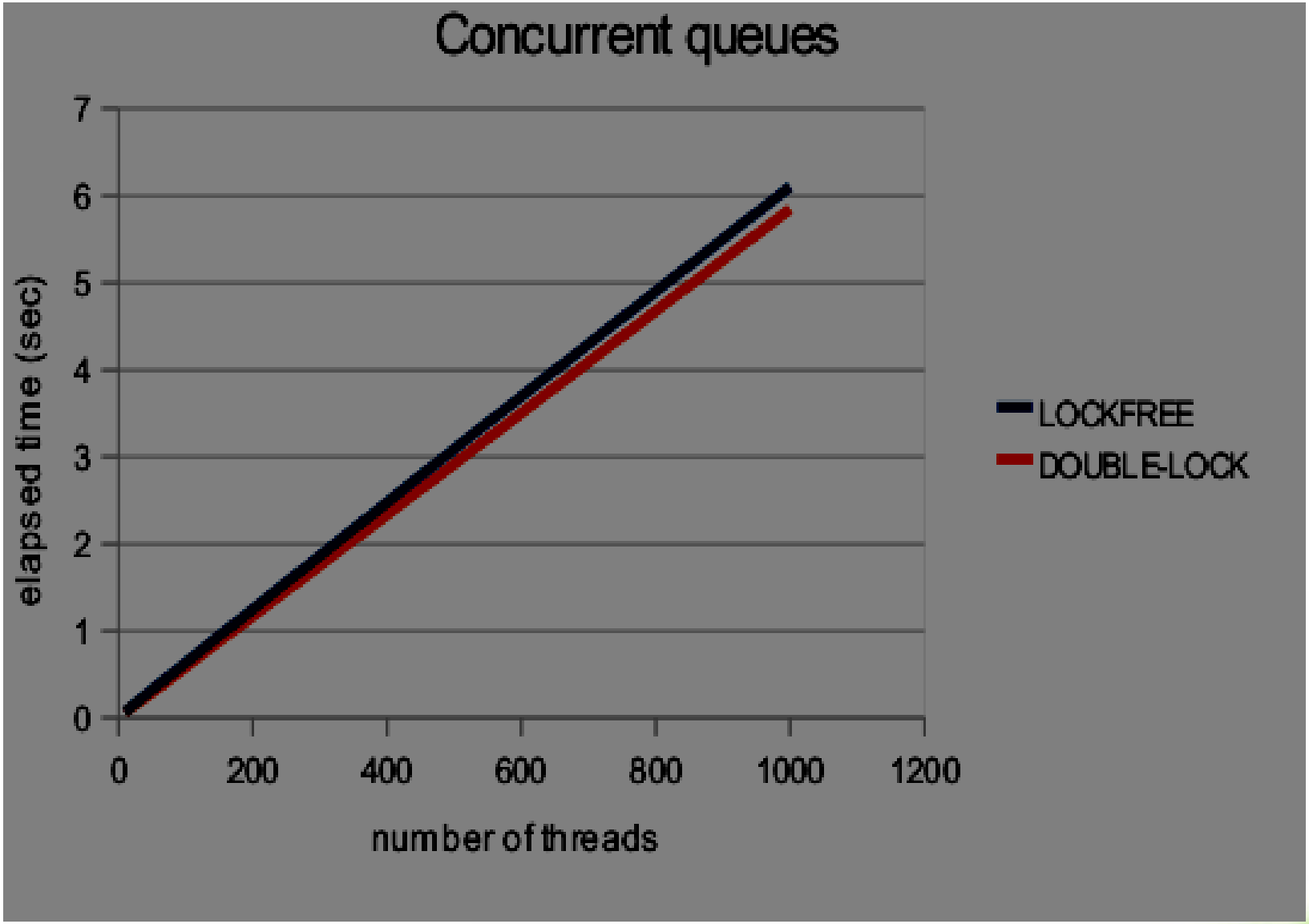
  - 10.0000 operations
  - (50% add, 10% search, 40% remove)

- Machine parameters:
  - Intel Core i7
  - 8 cores (1.73 Ghz, 6144 Kb cache)

Performance - stack

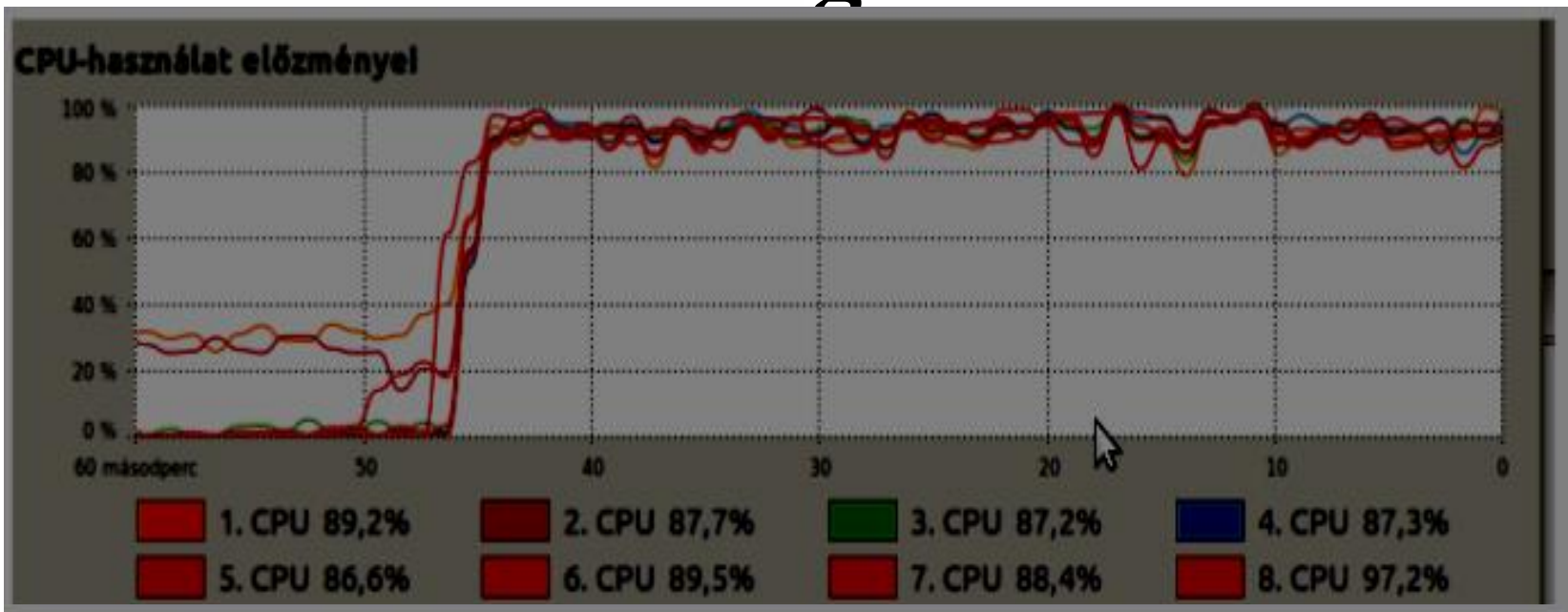


Performance - queue

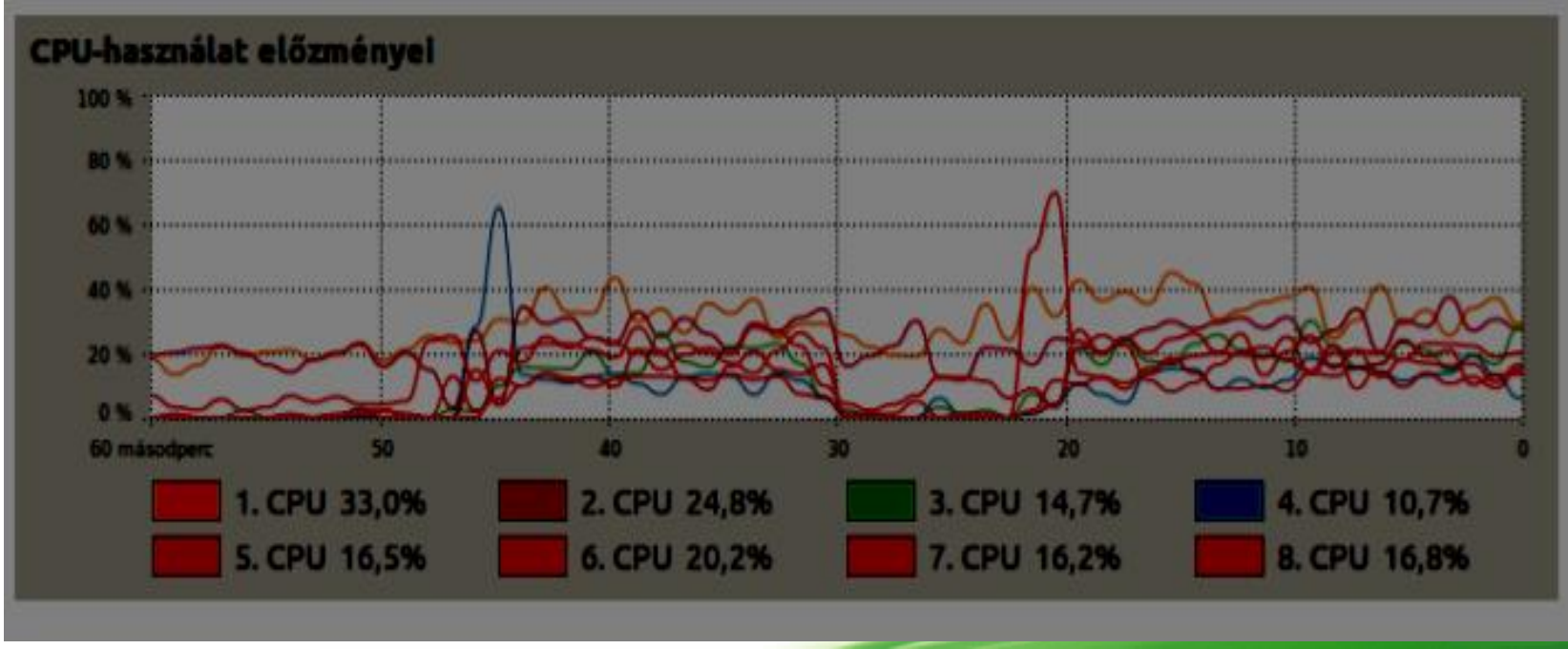


Performance – CPU usage

Lock-free case



Lock-based case



Concurrent sorted list solutions

- Coarse-grained

One lock (for every operations using only one lock).

- Fine-grained

Every element can be locked.

- Optimistic

Search, then locking element(s) (checking).

Optimistic global

Search, then locking all elements (checking).

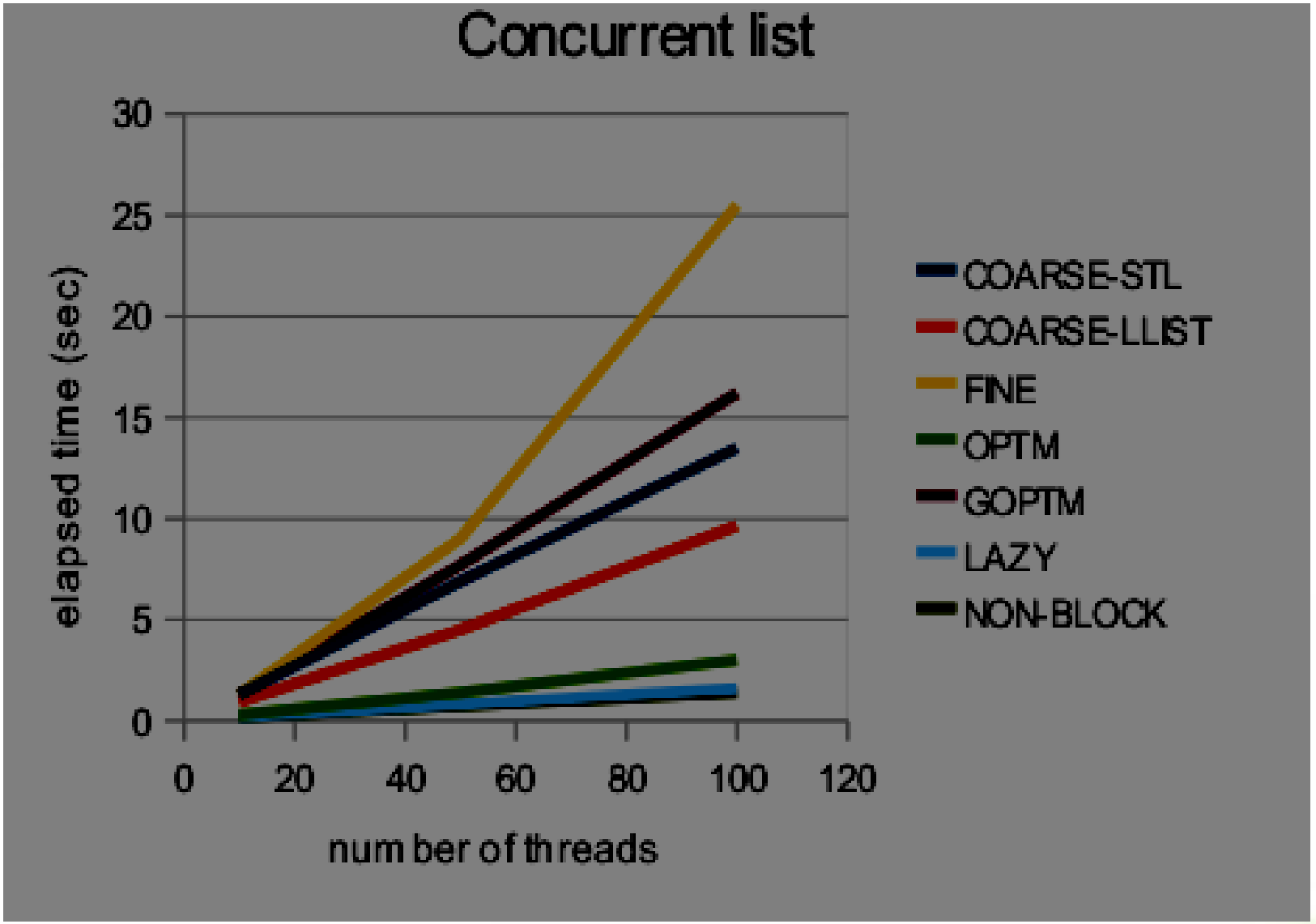
- Lazy

Like the optimistic, but remove in two steps (logicall and physically): mark, then unchaining.
(Here find() method is wait-free.)

- Lock-free

Like lazy, but using CAS.
Special search() method in case of element unchaining.

Performance – sorted list



Conclusion:

- Lock-free:
  - many cases better performance
  - more concurrent access, but under high-peak load it may lead to active waiting
  - implementation is difficult, complex code; in case of no GC needs to have memory handler solution: difficult, even worst performance
  - as component it is difficult to build-in, expand or re-use it
  - in case of simple structure good choice

- Lock-based:
  - simple usage, more adaptable
  - recommended in case of complex structures

Conclusion:

It can **not** be declared that there exists an optimal working data structure that can be used in every situation. The efficiency of all of paralell realization depends on the way of his usage.