bureau 14
SOFTWARE R&D

Scaling with C++ 11

Edouard Alligand, Founder

2013: Expectations

2013: Reality

Congratulations…

…you have
a scalability problem!

Do you?

No

- Performance not important
- CPU not the bottleneck
- Too expensive

Somewhat

- Need « some » performance

Yes

- Need to be as fast as possible
- Strict latency requirements
- It will make my mother proud

Before we continue: a few words about us

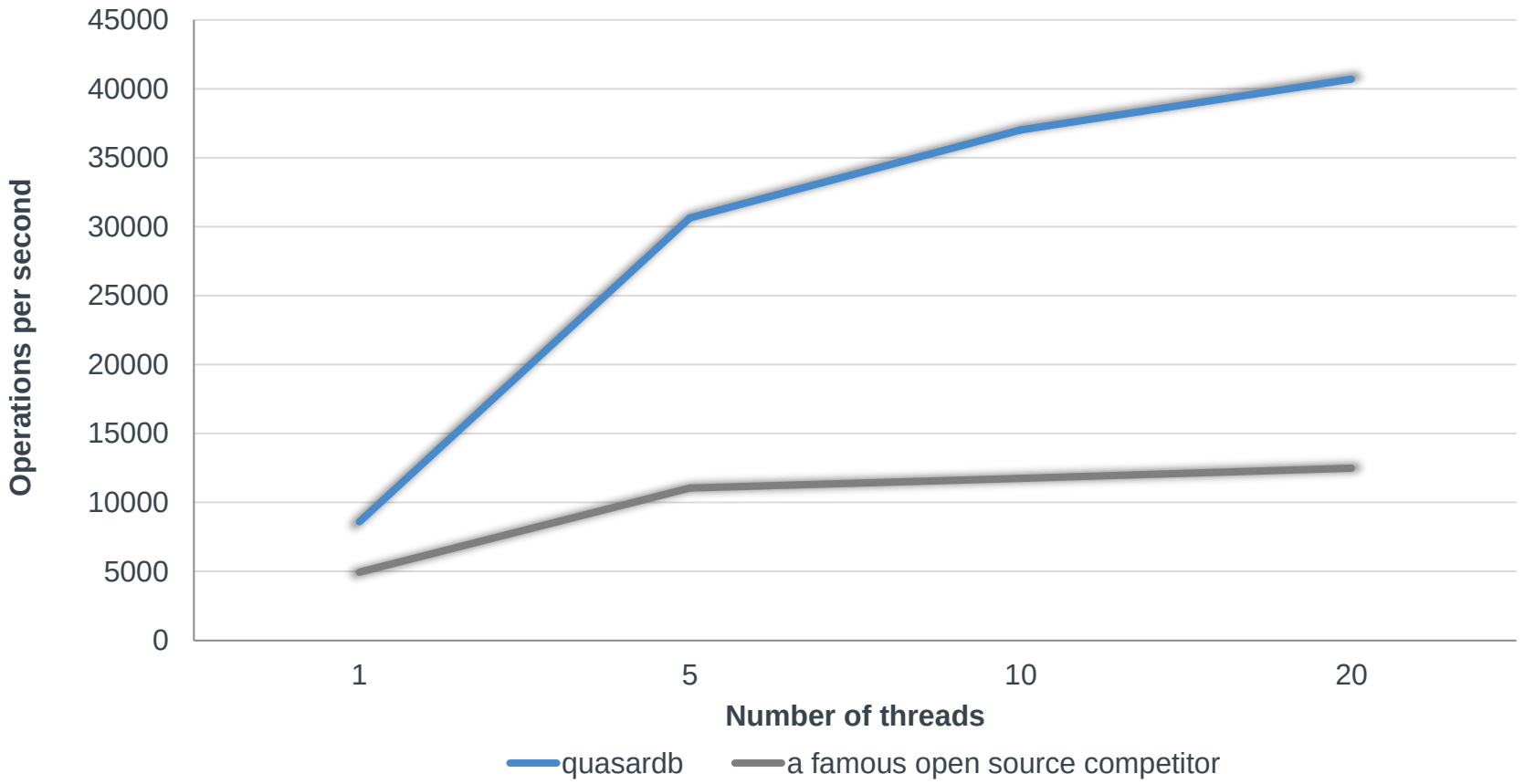
Independent software vendor established in 2008

Bootstrapped through consulting

Designed from the ground up a post-modern database: quasardb

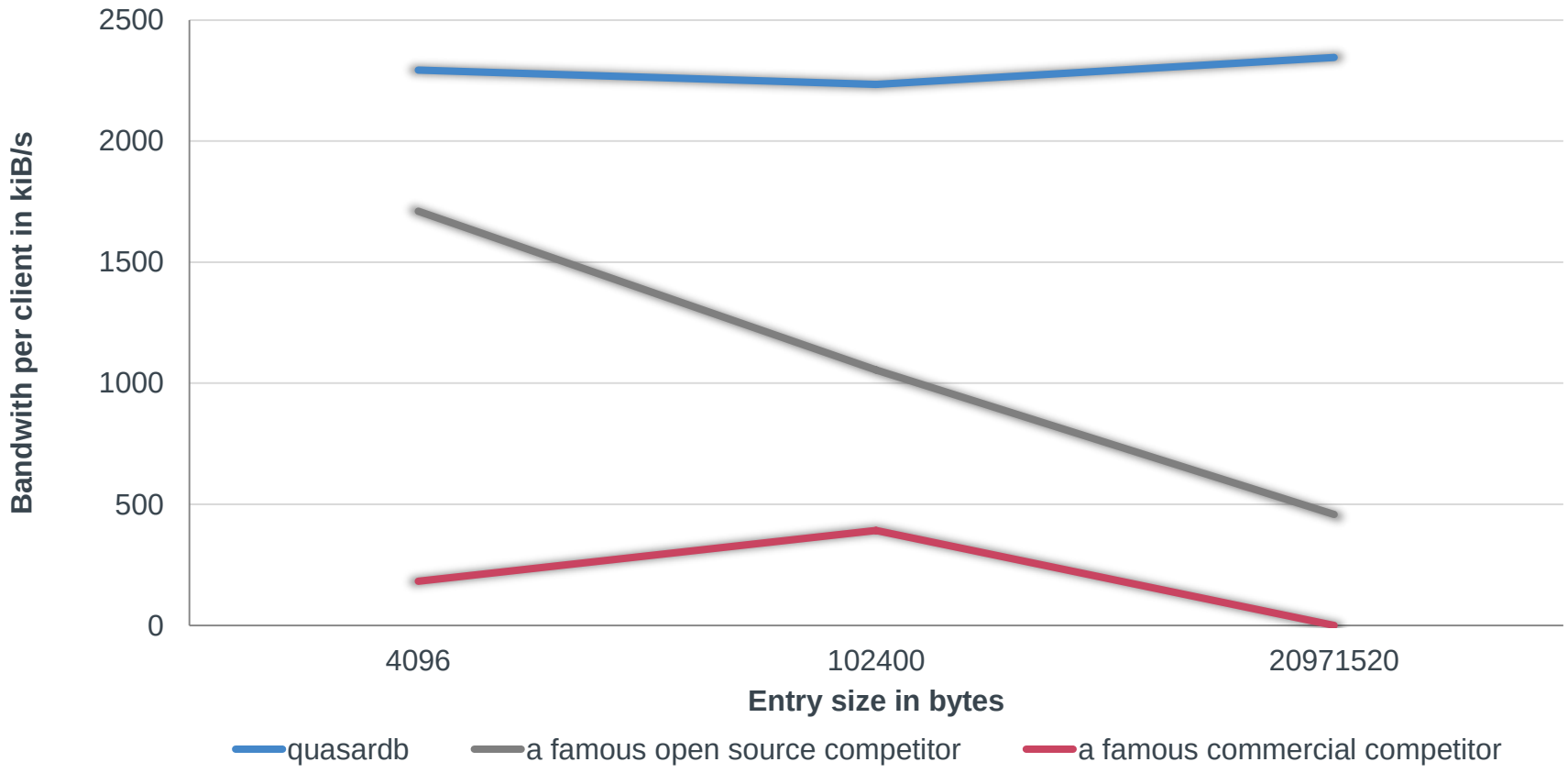
Scalability: our story

Yahoo benchmark - Read mostly 10 kiB entries

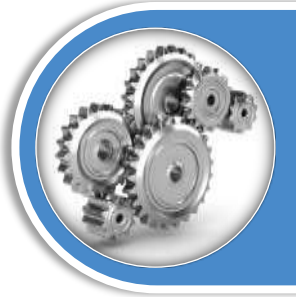


Scalability: more cowbell

500 simultaneous clients reads



Three steps toward extreme scalability



Design

- « Limitless »
- Natural
- Scalable algorithms

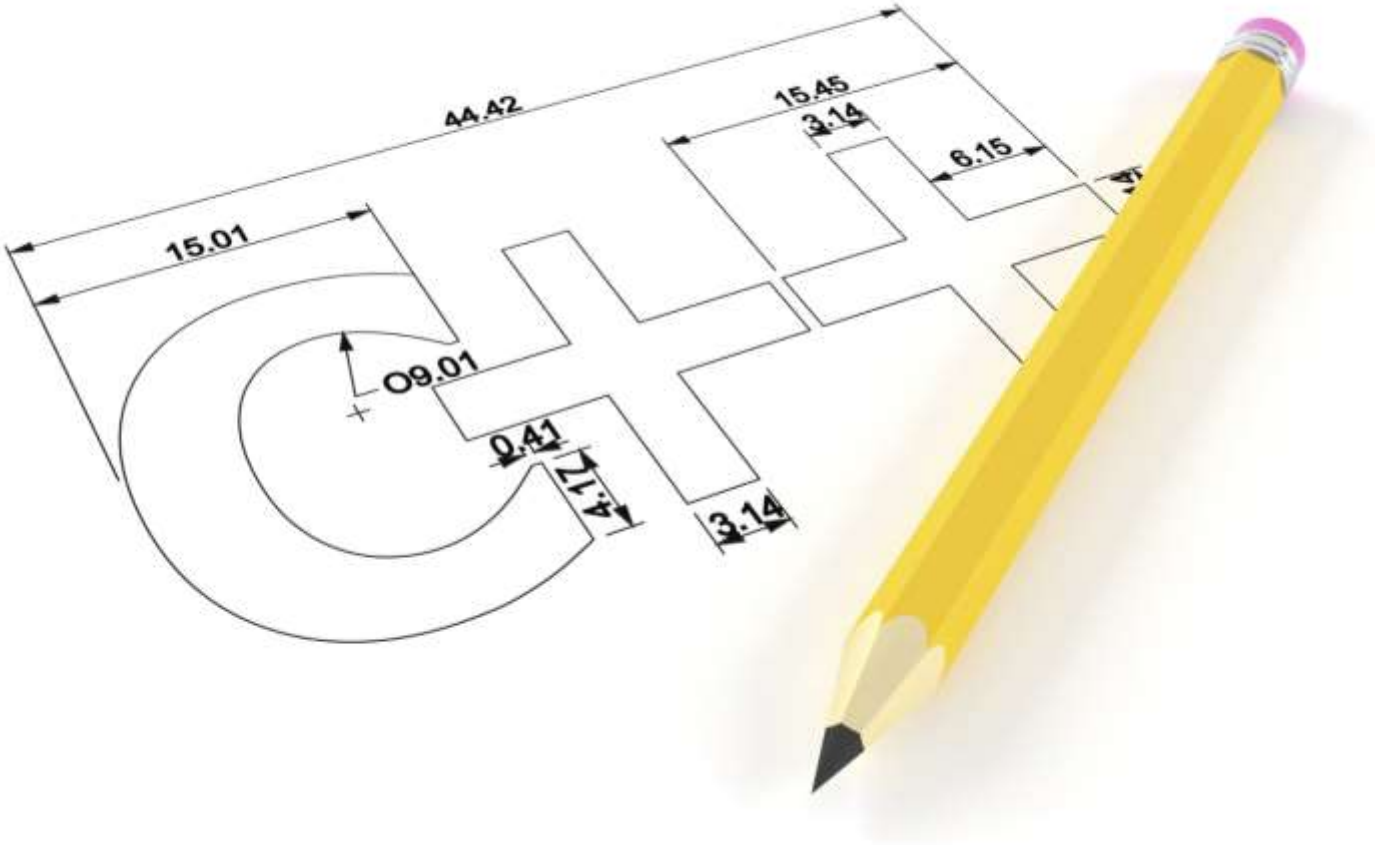
Memory strategies

- Space-time tradeoffs
- Spot hidden costs
- Adaptable

Tactical solutions

- Benchmark everything
- Scalability first, local optimizations second

Is C++ (11) relevant?

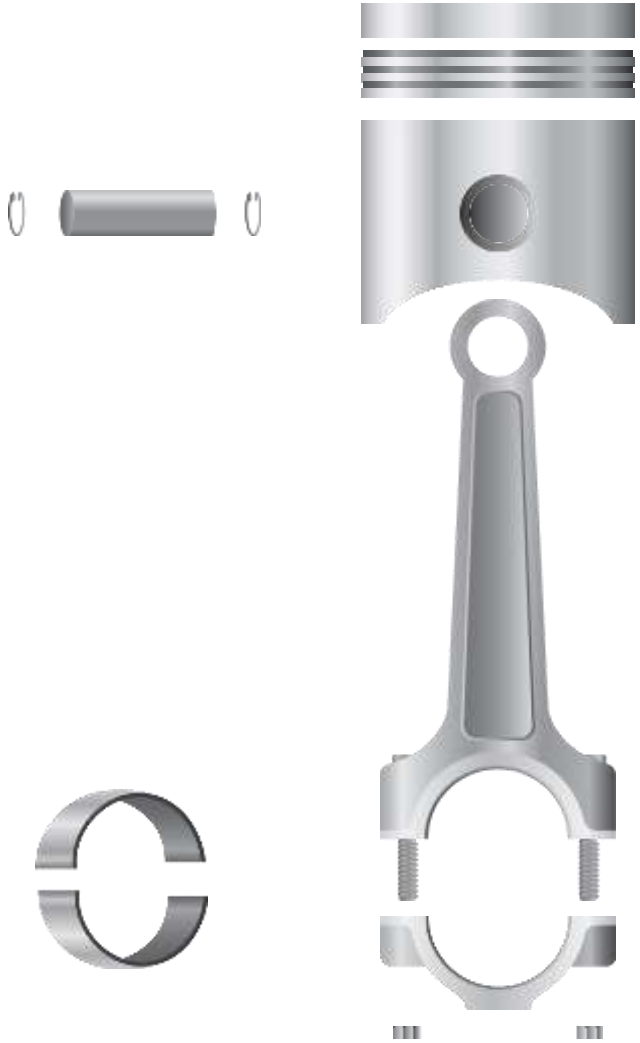


Design for scalability

“Talents imitate, geniuses steal.”

—Oscar Wilde

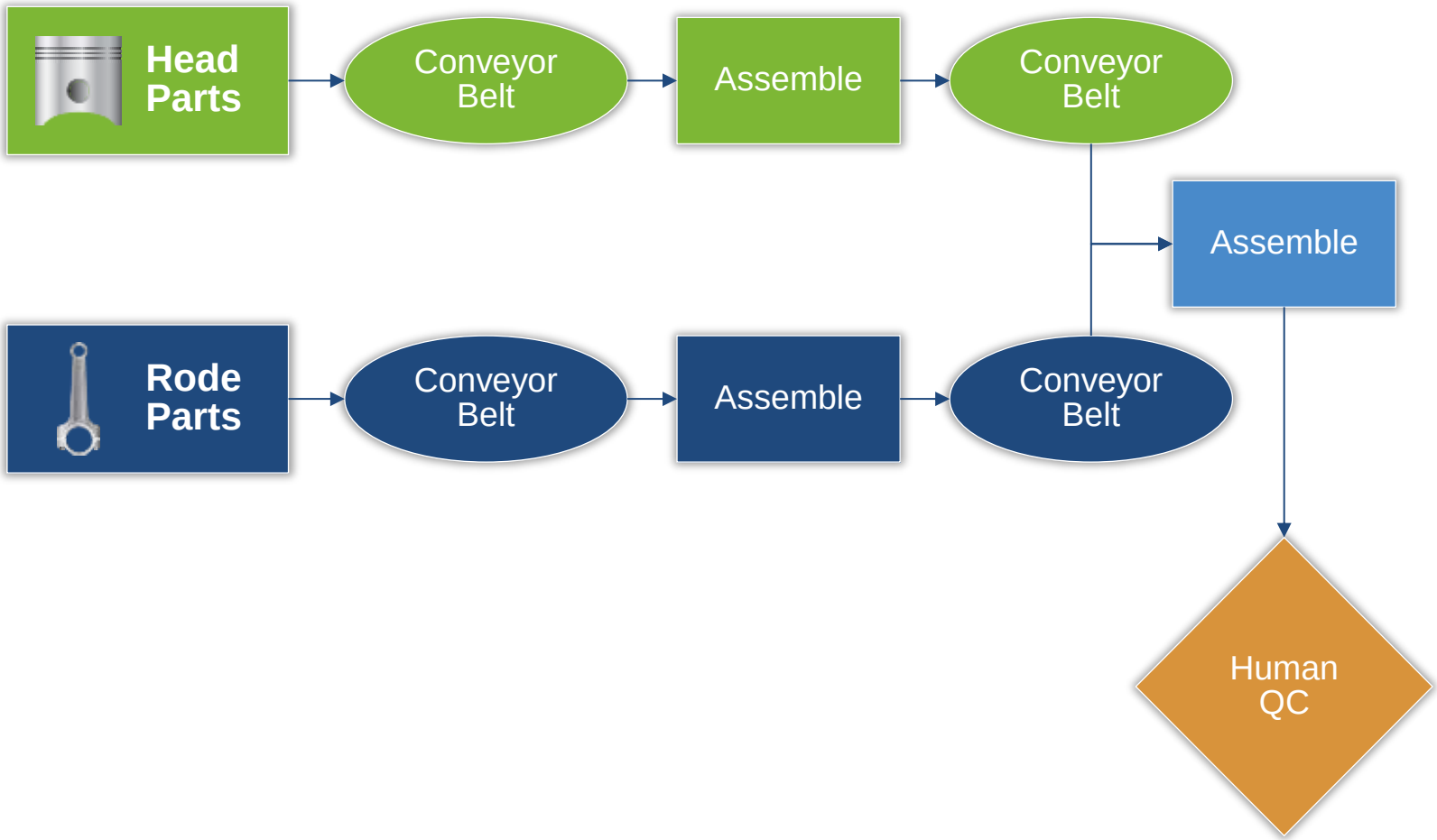
Manufacturing a piston



Manufacturing many pistons

Change the gravitational constant of the universe

Manufacturing billions of pistons



A scalable design

- Design your software as independent modules that communicate through messages
- No shared state
- Every component should be lightweight and fast
- Never block! ***The spice must flow.***

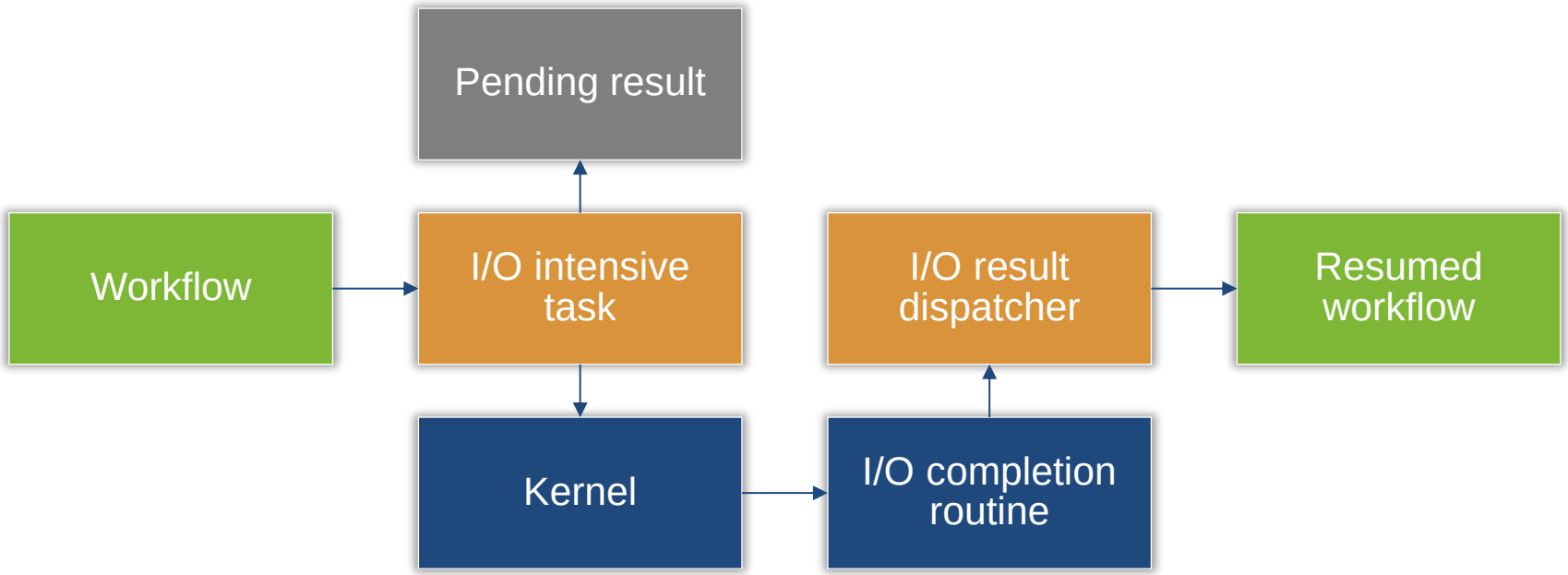
Mathematically speaking

$$P(x) = (h \circ g \circ f)(x)$$

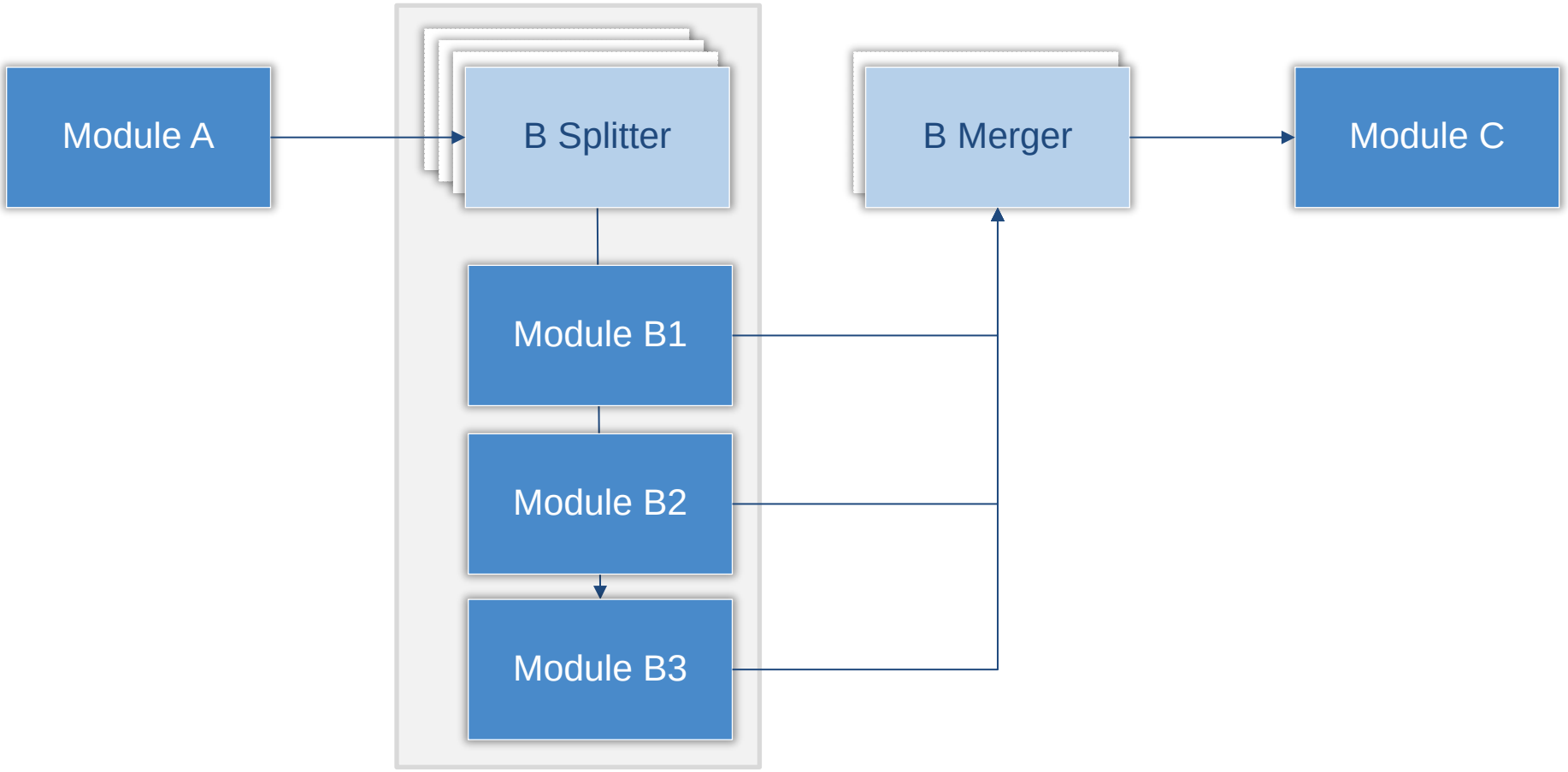
C++ speaking

```
// your program is a series of functors
struct f { int operator()(int x); };
struct g { int operator()(int x); };
struct h { int operator()(int x); };

// once you have “this”
// scaling is easy!
// you can use our composition library
// at https://github.com/bureau14/open_lib
int main(int argc, char ** argv)
{
    return h(g(f(argv[0])));
}
```

Asynchronous I/O

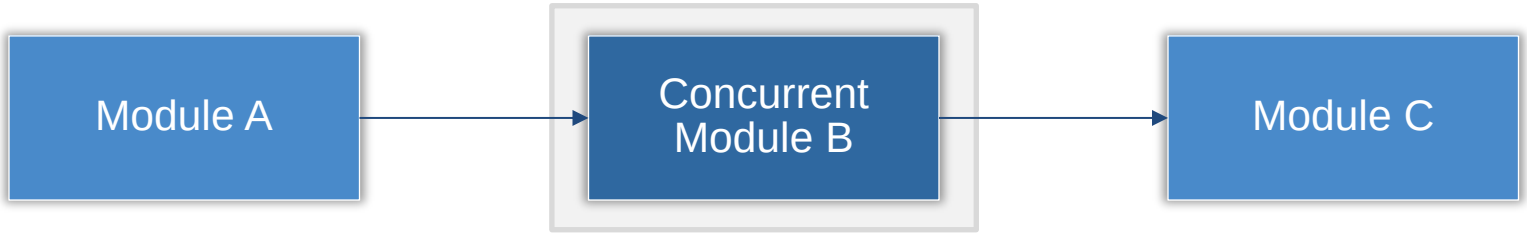


Scalability: you need more



Duplicate CPU intensive components
to increase scalability

Scalability: you need more



Or make them
highly concurrent

Outstanding issues

- Distribution and load-balancing
- Tasks scheduling
- Tasks granularity
- Shared memory design

Designing for scalability: take away

“Scalability isn’t a feature you add to a program, you design a program to be scalable.”

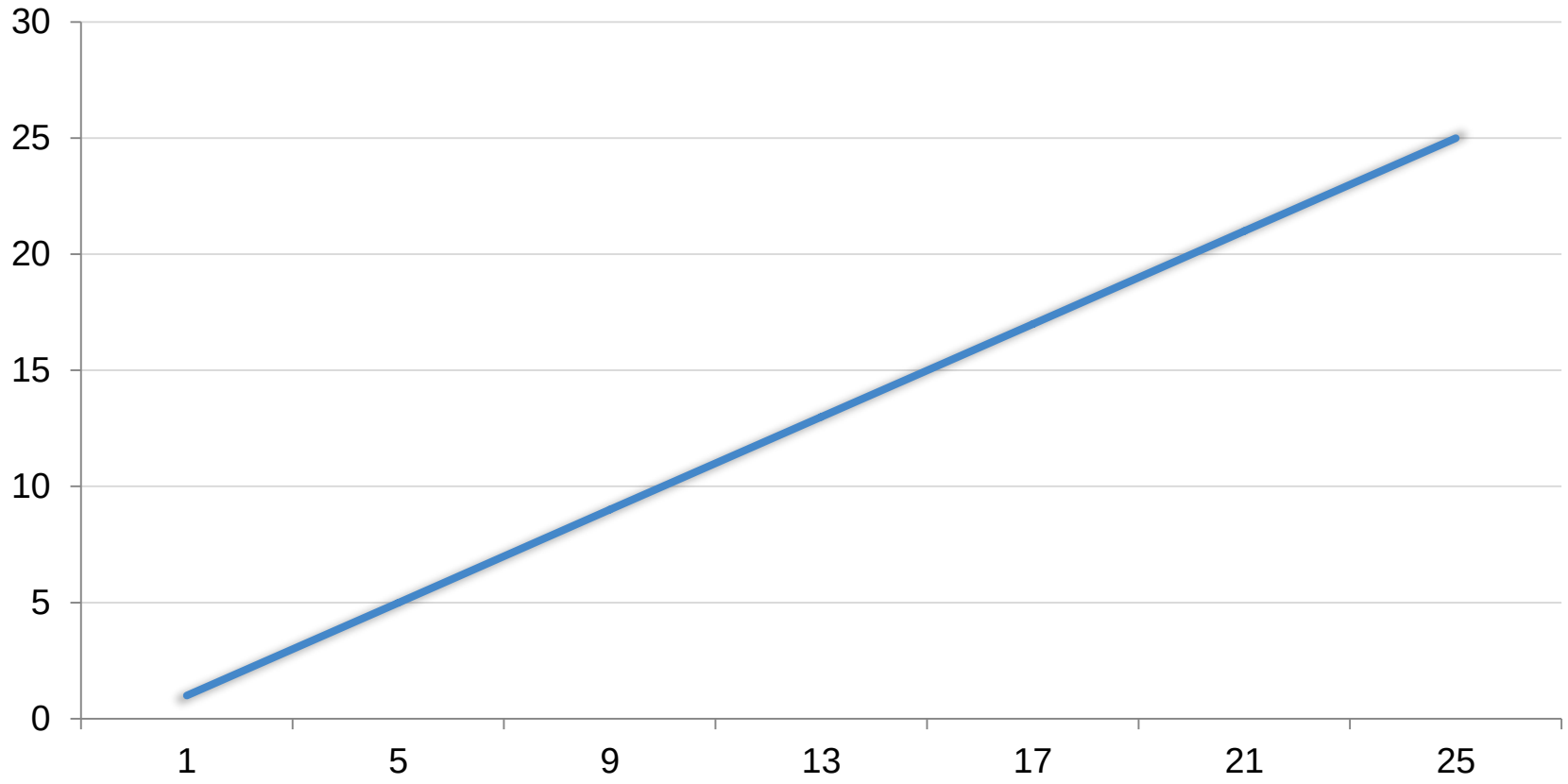
Memory strategies

“Intelligence is the wife, imagination is the mistress, memory is the servant.”

—Victor Hugo

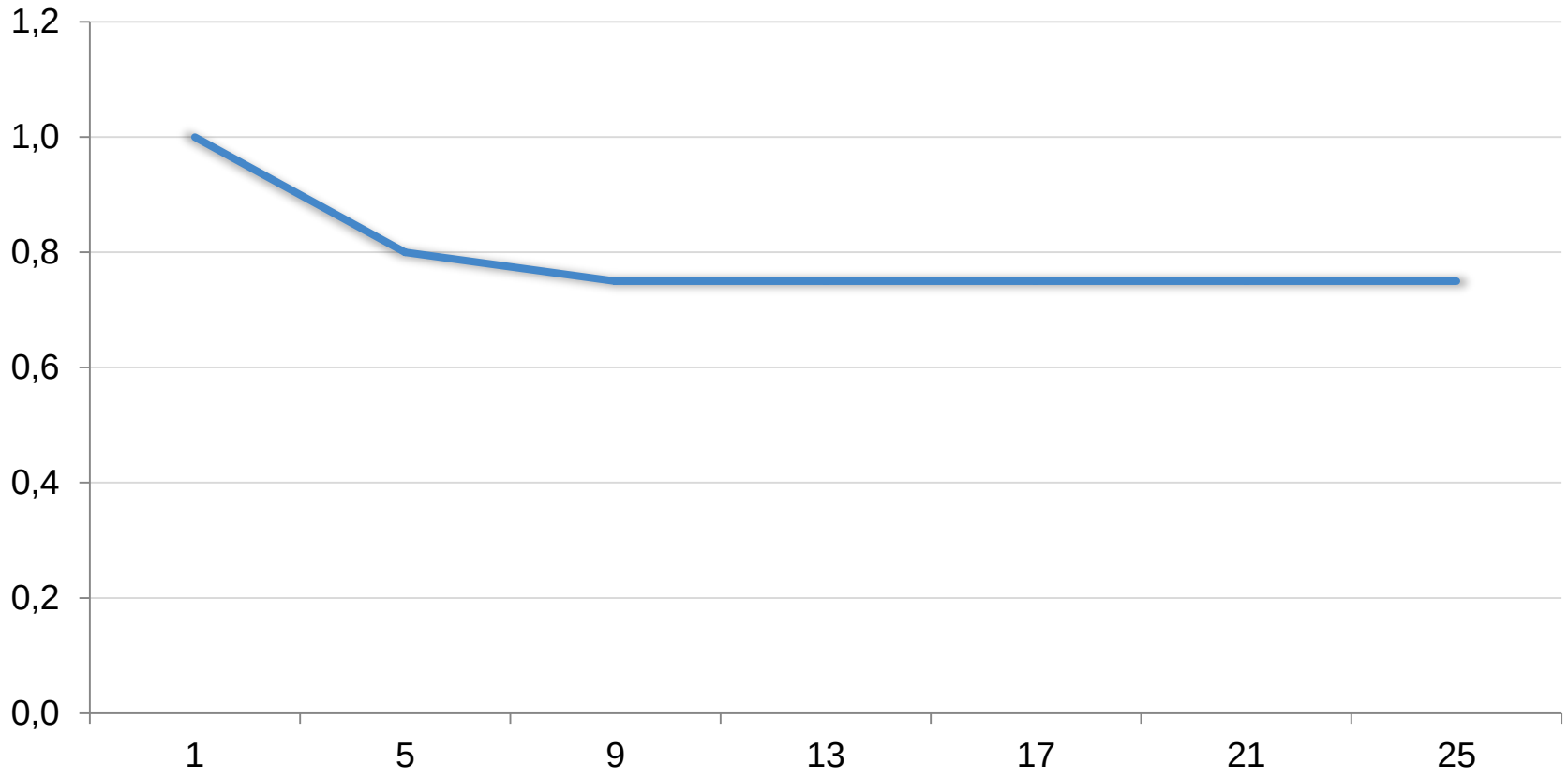
Read scalability

Shared read throughput per threads count



Write scalability

Shared write throughput per threads count



Memory management: it takes out all the fun

- Lifetime management is a fundamental computer science problem and a core C++ issue
- Adding threads: obviously not a simplification

Value based memory management

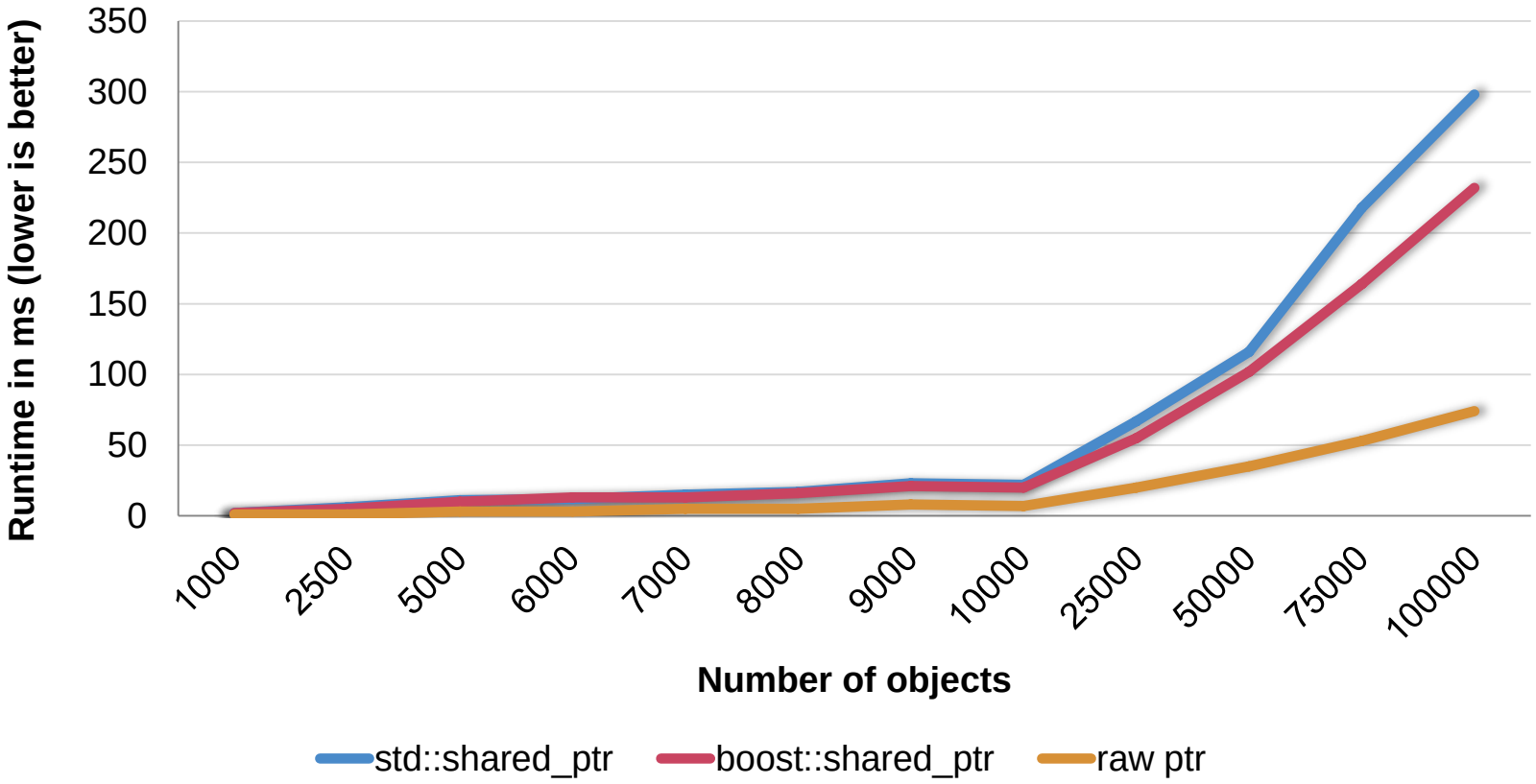
- C++ 11ish
- No leaks
- No locks

- Speed
- Memory usage

The SR-71 solution

Reference counted objects

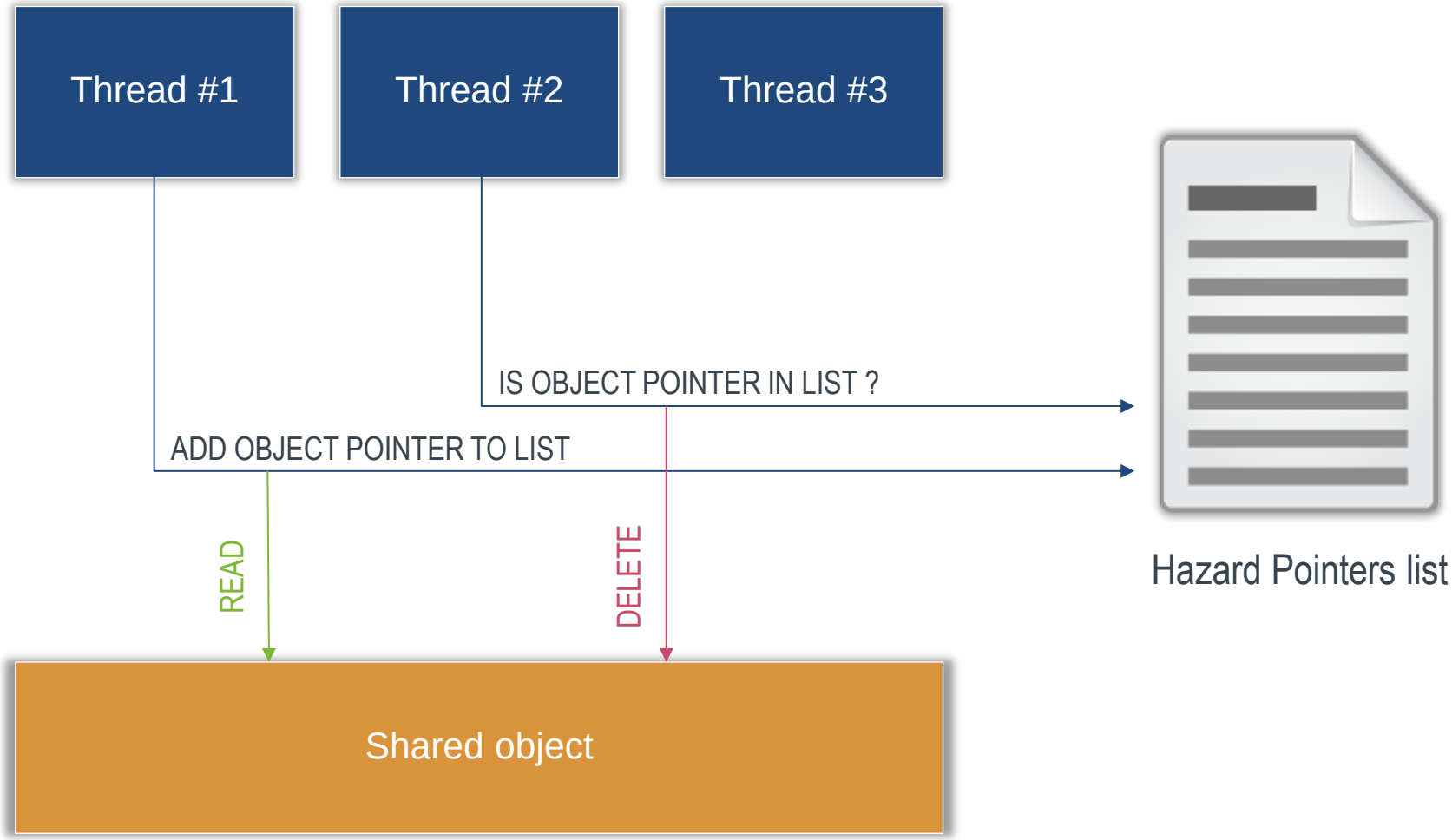
Pointer intensive manipulations—8 threads



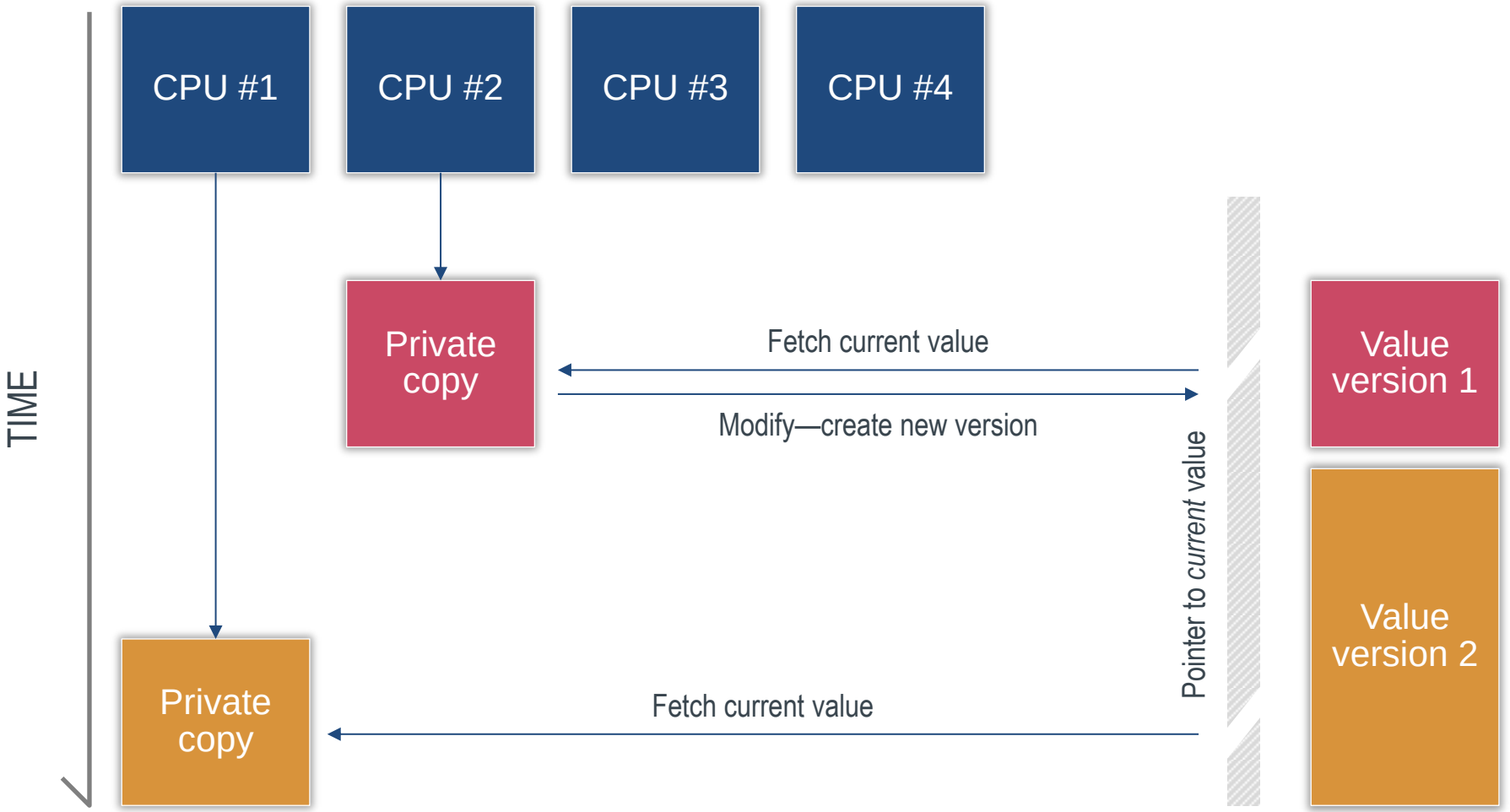
Memory recycling and pools

```
// initialization
void * buf = scalable_alloc(sizeof(T));
T * p = new(buf)T();

// recycling
p->~T();
p = new(p)T();

// final destruction
p->~T();
scalable_free(p);
```

Hazard pointers



Read-copy update



Garbage collectors

Don’t outsource strategic components

- That being said…
  - Many exist for C++
  - Very useful for prototyping
  - “Generally” greater memory usage
  - Corner cases may be catastrophic

Memory management: take away

*“You need to have **several different** memory management strateg**ies**.”*

Tactical solutions

“There are not more than five musical notes, yet the combinations of these five give rise to more melodies than can ever be heard.”

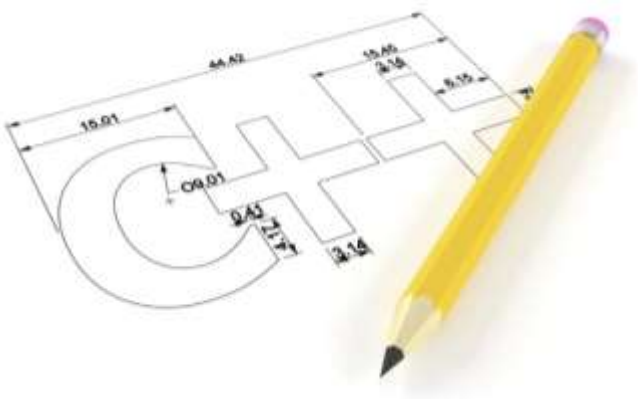
—Sun Tzu

C++ 11 brings us

std::thread()

std::call_once

Locks



thread_local

std::async()

Perfect forwarding

Futures and promises

Lambdas

And much more!

Lockfree structures

```
// most famous lockfree structure: the lockfree queue
boost::lockfree::queue<int> q;

// never blocks, returns false when q is full
q.push(3);

// never blocks, returns false when q is empty
int value = -1;
q.pop(v);

// also available: stacks, skip lists, blocking queues
```

Lockfree queue example

```
static tbb::concurrent_bounded_queue<message> __queue;

void foreground(void) {
   // […]
   __queue.push(m);
   // […]
}

void background(void) {
    message m;
    while(true) {
       __queue.pop(m); if (stop_requested(m)) break;
      // […]
}}
```

Thread local storage — la surprise du chef

```
// now in the C++ 11 standard!
thread_local int __tls_int = -1;

int f(void)
{
    // working directly on the tls variable can be slow
    int local_copy = __tls_int;
    // do stuff on local_copy...
    __tls_int = local_copy;
}

// TBB underestimated jewels:
// tbb::combinable and tbb::enumerable_thread_specific
```

Thread local storage example

```
// waitfree process-wide unique value

// don't
std::atomic<int> __value = 0;
int unique(void) { return __value++; }

// do
thread_local int __value = 0;

int unique(void)
{
    return (__value++ << 16) + (gettid() & 0xffff);
}
```

Atomics!

```
// now in the C++ 11 standard!
std::atomic<int> v = 0;
int snapshot = v++;
if (v == 42) // will be fenced if needed

// relax memory ordering for more performance
int snapshot = v.fetch_add(1, std::memory_order_relaxed);
// but be careful…
if (v.load(std::memory_order_relaxed) == 42)

// may not be lockfree!
v.is_lock_free()
```

Lightweight locks

```
// use them like vanilla locks
{
    tbb::spin_lock::scoped_lock lock(_mutex);
    // lock held
}

// you also have reader-writer lightweight locks
{
    tbb::spin_rw_mutex::scoped_lock lock(_mutex, false);
    // read-only lock held
}
```

And so much more!

- Transactional memory
- Asynchronous operations
- OS specific tools (kpoll, overlapped I/O…)
- Many third party libraries

Tactical solutions: take away

"Be proficient in many arts but thoughtful with their usage."

Tools of the trade

The Boost libraries
http://www.boost.org/

FastFlow
http://calvados.di.unipi.it/dokuwiki/doku.php?id=ffnamespace:about

Valgrind
http://valgrind.org/

Intel Threading Building blocks
http://threadingbuildingblocks.org/

Intel Amplifier XE
http://software.intel.com/en-us/intel-vtune-amplifier-xe

Bureau 14 Open Source Libraries
https://github.com/bureau14/open_lib

Suggested reading

Communicating Sequential Processes
C.A.R. Hoare

C++ Concurrency in Action
Anthony Williams

The Art of Concurrency
Clay Breshears

Synchronization Algorithms and Concurrent Programming
Gadi Taubenfeld

Intel Threading Building Blocks documentation

Questions and answers