Designing multithreaded code for scalability

Anthony Williams

Just Software Solutions Ltd
https://www.justsoftwaresolutions.co.uk

11th April 2018
Designing multithreaded code for scalability

- Scalability
- Limitations
- Designing for Scalability
Scalability
Modern C++ code runs across a wide variety of platforms:

- Embedded single-core microcontrollers
- Embedded multi-core systems
- Multi-core desktop computers
- Multi-core / multi-socket servers
- Many-core / many-socket HPC systems
Modern C++ code runs across a wide variety of platforms:

- 1 CPU / 1 core
- Embedded multi-core systems
- Multi-core desktop computers
- Multi-core / multi-socket servers
- Many-core / many-socket HPC systems
Scalability

Modern C++ code runs across a wide variety of platforms:

- 1 CPU / 1 core
- 1 CPU / 4 cores
- Multi-core desktop computers
- Multi-core / multi-socket servers
- Many-core / many-socket HPC systems
Scalability

Modern C++ code runs across a wide variety of platforms:

- 1 CPU / 1 core
- 1 CPU / 4 cores
- 1 CPU / 8 cores
- Multi-core / multi-socket servers
- Many-core / many-socket HPC systems
Scalability

Modern C++ code runs across a wide variety of platforms:

- 1 CPU / 1 core
- 1 CPU / 4 cores
- 1 CPU / 8 cores
- 4 CPUs / 32 cores per CPU
- Many-core / many-socket HPC systems
Scalability

Modern C++ code runs across a wide variety of platforms:

- 1 CPU / 1 core
- 1 CPU / 4 cores
- 1 CPU / 8 cores
- 4 CPUs / 32 cores per CPU
- 10000 CPUs / 12 cores per CPU
Modern C++ code runs across a wide variety of platforms:

- 1 CPU / 1 core
- 1 CPU / 4 cores
- 1 CPU / 8 cores
- 4 CPUs / 32 cores per CPU
- 10000 CPUs / 12 cores per CPU
- Plus GPUs
Modern C++ code runs across a wide variety of platforms:

- 1 CPU / 1 core
- 1 CPU / 4 cores
- 1 CPU / 8 cores
- 4 CPUs / 32 cores per CPU
- 10000 CPUs / 12 cores per CPU
- Plus GPUs — up to 65536 cores
Scalability

Communicating between threads has different constraints across these systems.

Your code is **scalable** if it can run on any of these systems without penalty.
Why Scalability?

- Desktops are getting more cores
- Phones are getting more cores
- Servers are getting more CPUs and cores
- Our customer's machines are getting more CPUs and more cores.

Our software needs to be scalable.
Why Scalability?

- Desktops are getting more cores
- Phones are getting more cores

Designing multithreaded code for scalability

Anthony Williams  Just Software Solutions Ltd  https://www.justsoftwaresolutions.co.uk
Why Scalability?

- Desktops are getting more cores
- Phones are getting more cores
- Servers are getting more CPUs and cores

Designing multithreaded code for scalability
Why Scalability?

- Desktops are getting more cores
- Phones are getting more cores
- Servers are getting more CPUs and cores
- Our customer’s machines are getting more CPUs and more cores.
Why Scalability?

- Desktops are getting more cores
- Phones are getting more cores
- Servers are getting more CPUs and cores
- Our customer’s machines are getting more CPUs and more cores.

Our software needs to be scalable
Limitations
Limitations: Mutex contention

Mutex: Mutual Exclusion

A mutex is a means of preventing concurrent execution.

instead of picking up Djikstra’s cute acronym we should have called the basic synchronization object "the bottleneck" (David Butenhof)
Limitations: Mutex contention

Mutex: Mutual Exclusion

A mutex is a means of preventing concurrent execution.

*instead of picking up Djikstra’s cute acronym we should have called the basic synchronization object "the bottleneck" (David Butenhof)*

⇒ For scalable solutions, we need to avoid mutex contention.
Atomic operations can suffer from contention too:

**Read-Modify-Write** operations always affect the *latest* values

⇒ RMW operations on a single location need to be serialized by the CPUs
Limitations: Atomic contention

Atomic operations can suffer from contention too:

**Read-Modify-Write** operations always affect the *latest* values

⇒ RMW operations on a single location need to be serialized by the CPUs

⇒ For scalable solutions, we need to be sparing with RMW operations
Limitations: False Sharing

CPUs synchronize memory at the granularity of a cache line.

Cache lines are typically 16-128 bytes

⇒ objects that are on the same cache line are essentially the same object for contention purposes
Limitations: Cache Ping-Pong
Cache Ping-Pong is where a cacheline is continuously shuttled back and forth between two processors. This occurs when two threads are accessing either:

- **the same** atomic variable
- **different** variables on **the same cache line**

This can have a **big** performance impact, because transferring cache lines is **slow**.
Limitations: Speed of Light

The speed of light is $3 \times 10^8$ m/s

CPU clocks are around 3GHz

⇒ The speed of light is around 10cm/tick
Limitations: Speed of Light

The speed of light is $3 \times 10^8$ m/s

CPU clocks are around 3GHz

$\Rightarrow$ The speed of light is around 10cm/tick

$\Rightarrow$ There is a hard upper limit on communication speed for multi-socket systems
Limitations: Memory bandwidth

Intel Xeon Phi 7295:
- 115.2Gb/s Memory bandwidth
- 1.5Ghz Clock speed
- 72 Cores

⇒ 76.8 bytes per clock
⇒ 1.1 bytes per clock per core
Designing for Scalability
Strategies: Batch Communications

Can you avoid intermediate synchronization?

Each thread works on its own data, and only modifies shared data at the end.
std::vector<unsigned> const values=get_values();
std::atomic<unsigned long long> total{0};
unsigned const num_threads=...;
std::vector<joining_thread> threads(num_threads);
for(unsigned t=0;t<num_threads;++t){
  threads[t]=joining_thread([&],t){
    auto start=...;
    auto end=...;
    std::for_each(start,end,[&](auto x){
      total+=x;
    });
  }
}
std::vector<unsigned> const values=get_values();
std::atomic<unsigned long long> total{0};
unsigned const num_threads=...;
std::vector<joining_thread> threads(num_threads);
for(unsigned t=0;t<num_threads;++t){
    threads[t]=joining_thread([&,t]{
        auto start=...;
        auto end=...;
        auto local_total=std::accumulate(start,end,0ull);
        total+=local_total;
    });
}
Batch Communication Costs

Designing multithreaded code for scalability
std::vector<unsigned> const values = get_values();
unsigned const num_threads = ...;
std::vector<std::future<unsigned long long>> futures(num_threads);
for (unsigned t = 0; t < num_threads; ++t) {
    futures[t] = std::async(std::launch::async, [&,&, t] {
        auto start = ...;
        auto end = ...;
        return std::accumulate(start, end, 0ull);
    });
}
unsigned long long total = 0;
for (auto& f : futures) total += f.get();
Batch Communication Costs

Designing multithreaded code for scalability
Batch Communication Costs

Sum of 100000000 elements on 4 threads:

<table>
<thead>
<tr>
<th>Run</th>
<th>Time</th>
<th>Ratio to serial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>0.081s</td>
<td>1</td>
</tr>
<tr>
<td>All atomic</td>
<td>9.74s</td>
<td>120x slower!</td>
</tr>
<tr>
<td>End atomic</td>
<td>0.052s</td>
<td>1.6x faster</td>
</tr>
<tr>
<td>Futures</td>
<td>0.052s</td>
<td>1.6x faster</td>
</tr>
</tbody>
</table>
Contended Lists

Suppose we have a linked list, accessible by multiple threads, and we might need to add or remove elements. What can we do?

- Use a mutex for the whole list
- Use a mutex for each link in the list
- Use `std::atomic<std::shared_ptr<Node*> >` for the node links
- Use `std::atomic<Node*>` for the node links, and a Safe Reclamation scheme to ensure Nodes can be removed safely
Contended Lists: Costs

- Whole list mutex $\implies$ big bottleneck
- Node mutex $\implies$ lots of small bottlenecks
- \texttt{std::atomic<\texttt{std::shared_ptr<Node*> }}> $\implies$ spin-locks, or RMW operations
- \texttt{std::atomic<Node*> } $\implies$ low-cost for readers, \textbf{big cost} for writers
Safe Reclamation Options

- Garbage Collection
- RCU
- Hazard Pointers
Safe Reclamation: RCU

Readers just record entry/exit to the read function.

Writers make atomic changes, then wait for a grace period before deleting removed objects.
RCU costs

In user space:

- Read side:
  - Atomic read of global marker
  - Two atomic writes to a per-thread location

- Write side:
  - Atomic write of global marker
  - Multiple atomic reads of all per-thread locations for readers
  - Mutex locks, delays and spin-loops until all readers ready

In kernel space:

- Read side: no overhead!
- Write side:
  - Blocking wait until all processors have cycled a time slice
Safe Reclamation: Hazard Pointers

Readers store **hazard pointers** referring to objects being accessed.

Writers make atomic changes, then check the **hazard pointers** to see if it is safe to delete an object.
Hazard Pointers Costs

- **Read side:**
  - Two (or more) atomic writes to a per-thread hazard pointer
  - Spin-loop ensuring value hasn’t changed while updating hazard pointer

- **Write side:**
  - Atomic RMW operation adding to reclamation list
  - Objects not immediately destroyed
  - Period reclamation checks: when N objects are queued for reclamation
    - N depends on configuration parameters and number of threads
    - Each reclamation does atomic reads of all per-thread hazard pointers for readers
    - Cost of retiring objects varies by orders of magnitude
There is a proposal under discussion for both RCU and Hazard Pointers, with a sample implementation:

P0566R4: Proposed Wording for Concurrent Data Structures: Hazard Pointer and Read-Copy-Update (RCU)
http://wg21.link/p0566

RCU implementation:
https://github.com/paulmckrcu/RCUCPPbindings

Hazard Pointer implementation:
https://github.com/facebook/folly/tree/master/folly/experimental/hazptr
Sequential Consistency vs Eventual Consistency

Sequential Consistency:
- All threads see the same view of shared state
- Single Total Order of operations
- This requires serialization, or extensive communication

Eventual Consistency:
- Threads may see different views of shared state
  Provided each thread has a self-consistent view all is well
- All changes propagate to all threads eventually
- Cannot write a Single Total Order of operations
- Much less communication required
Sequential Consistency vs Eventual Consistency

Sequential Consistency is easier to reason about. Eventual Consistency is more scalable.
Summary

- Multithreaded code needs to be scalable
- Avoid contention
- Avoid cache ping-pong
- Use Safe Reclamation schemes
- Use Eventual Consistency
C++ Concurrency in Action: Practical Multithreading, Second Edition

Covers C++17 and the Concurrency TS

Early Access Edition now available

http://stdthread.com/book

Designing multithreaded code for scalability
Just::Thread Pro provides an actor framework, a concurrent hash map, a concurrent queue, synchronized values and a complete implementation of the C++ Concurrency TS, including a lock-free implementation of `atomic_shared_ptr`.

http://stdthread.co.uk
Questions?