ELT WFRTC
Software patterns and library solutions for low latency multithreading

Calle Rosenquist, ESO

Real time core

- CPU-based
- Single server (currently 128 core AMD EPYC Zen 3)
- Minimal Linux
- Control software
  - Single process
  - Statically linked
  - C++
  - Real-time threads running on isolated cores without scheduler ("isolcpus")

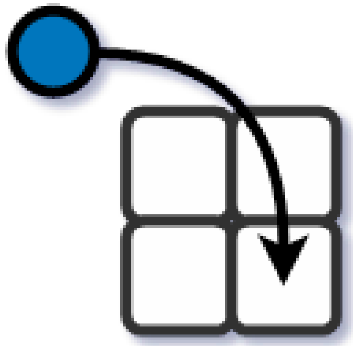
NUMA

non-uniform memory access

Data Classification: PUBLIC

"Pinning" real time threads

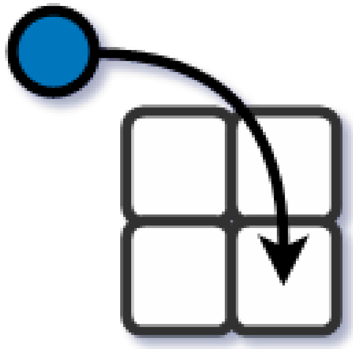
- In Linux each thread have two masks
  - "CPUs allowed" - where to run
  - "Mems allowed" - where to allocate memory
- By *pinning* we allow only one core and its local memory



Data Classification: PUBLIC

Memory allocations

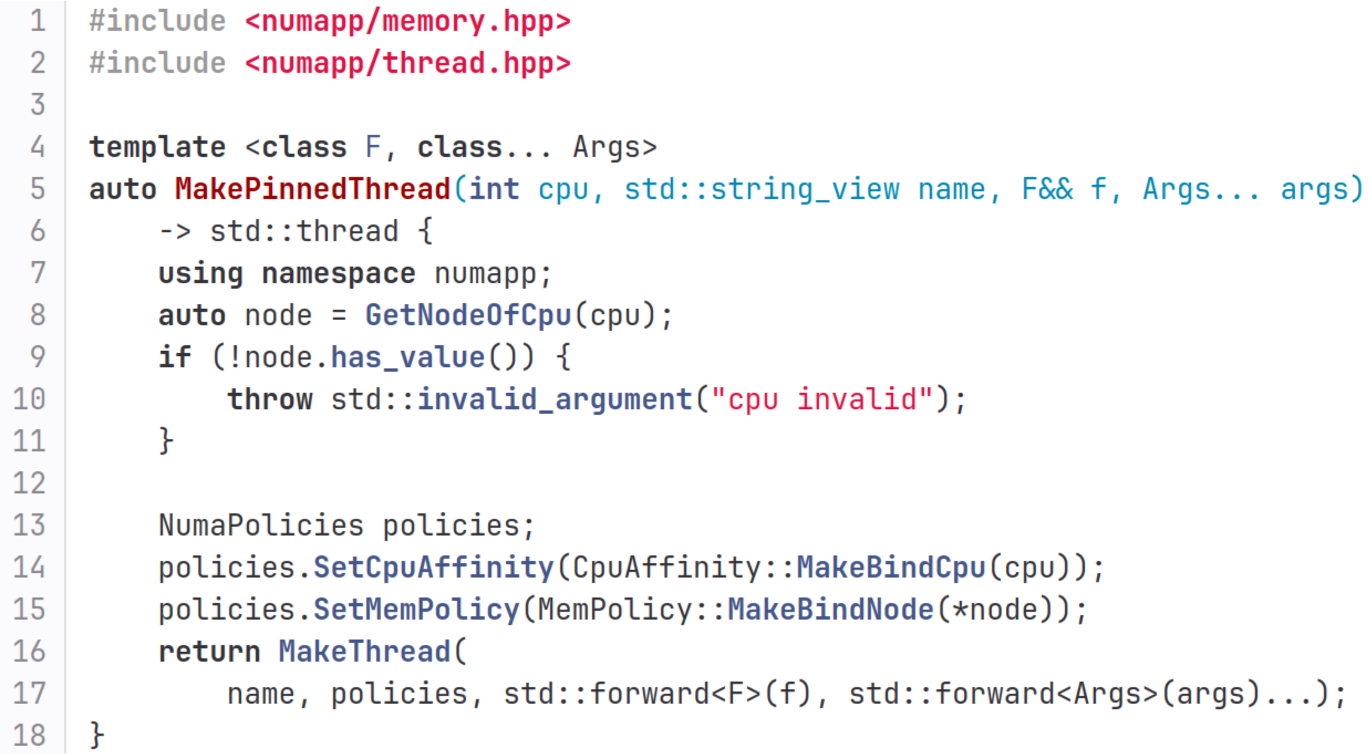
- Critical heap memory allocated from specific NUMA nodes
- Avoid allocators that rely on thread policy
  - E.g. default `operator new`, `malloc()`
  - Not easily predictable
  - Memory may come from allocator "free list"



Data Classification: PUBLIC

Support library: NUMA++

Creating pinned threads



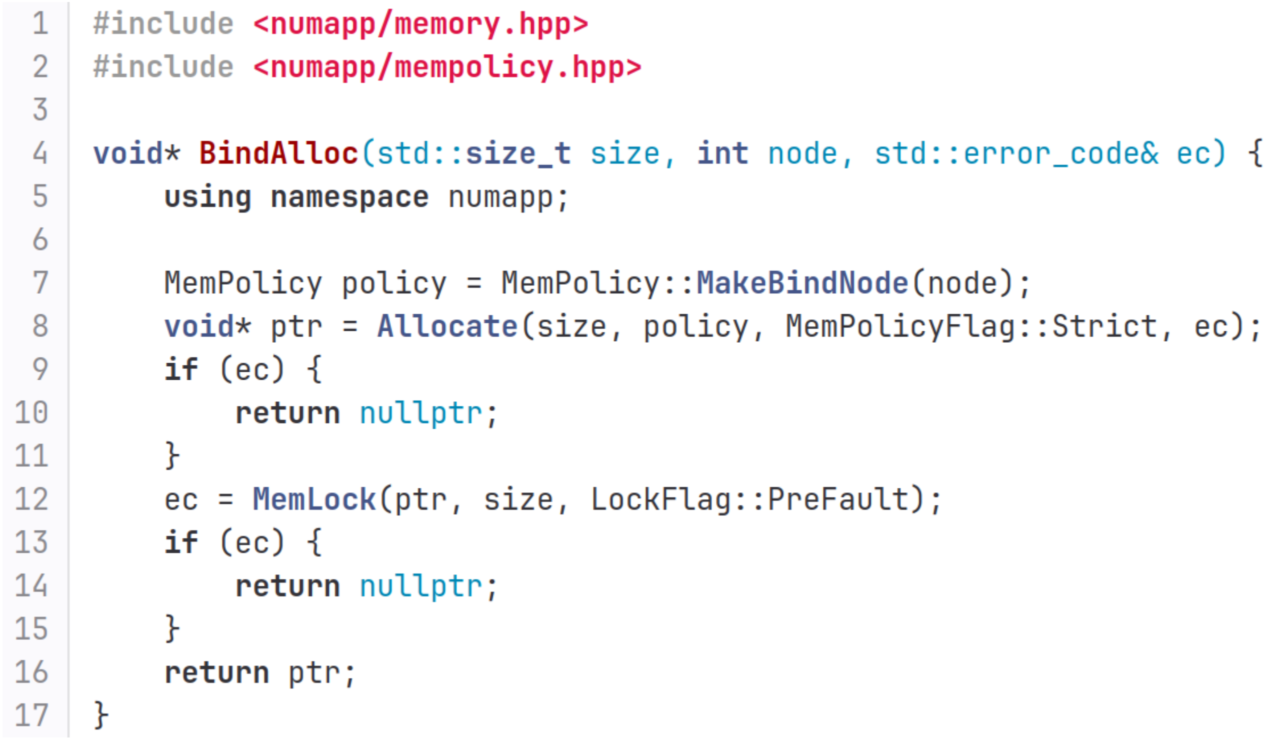
```cpp
#include <numapp/memory.hpp>
#include <numapp/thread.hpp>

template <class F, class... Args>
auto MakePinnedThread(int cpu, std::string_view name, F&& f, Args... args)
    -> std::thread {
    using namespace numapp;
    auto node = GetNodeOfCpu(cpu);
    if (!node.has_value()) {
        throw std::invalid_argument("cpu invalid");
    }

    NumaPolicies policies;
    policies.SetCpuAffinity(CpuAffinity::MakeBindCpu(cpu));
    policies.SetMemPolicy(MemPolicy::MakeBindNode(*node));
    return MakeThread(
        name, policies, std::forward<F>(f), std::forward<Args>(args)...);
}
```

In short

- Lookup node from CPU core
- Create policies
  - Pin thread to core
  - Memory from local node
- Create thread with provided policies

Support library: NUMA++

Allocating memory from a specific node



```cpp
#include <numapp/memory.hpp>
#include <numapp/mempolicy.hpp>

void* BindAlloc(std::size_t size, int node, std::error_code& ec) {
    using namespace numapp;

    MemPolicy policy = MemPolicy::MakeBindNode(node);
    void* ptr = Allocate(size, policy, MemPolicyFlag::Strict, ec);
    if (ec) {
        return nullptr;
    }
    ec = MemLock(ptr, size, LockFlag::PreFault);
    if (ec) {
        return nullptr;
    }
    return ptr;
}
```

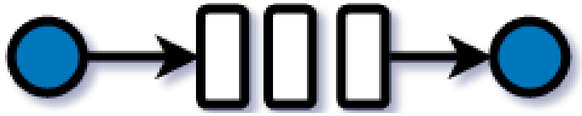
- Create policy to use single node
- Allocate
  - Uses `mmap()`
- Lock and pre-fault to trigger physical memory allocation

Inter-thread queues

Data Classification: PUBLIC

Inter-thread queues

- Deliver data from single producer to single consumer without locking or blocking
- Fix capacity at construction to avoid reallocation
- Use *views* (e.g., pointer + size) to avoid unnecessary memory copies

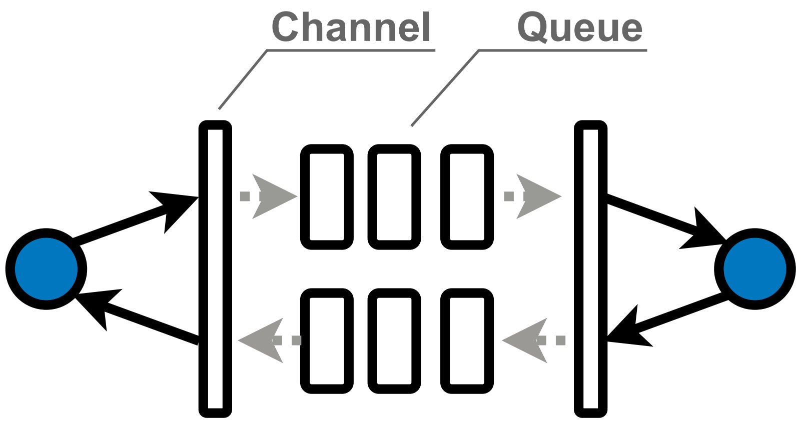


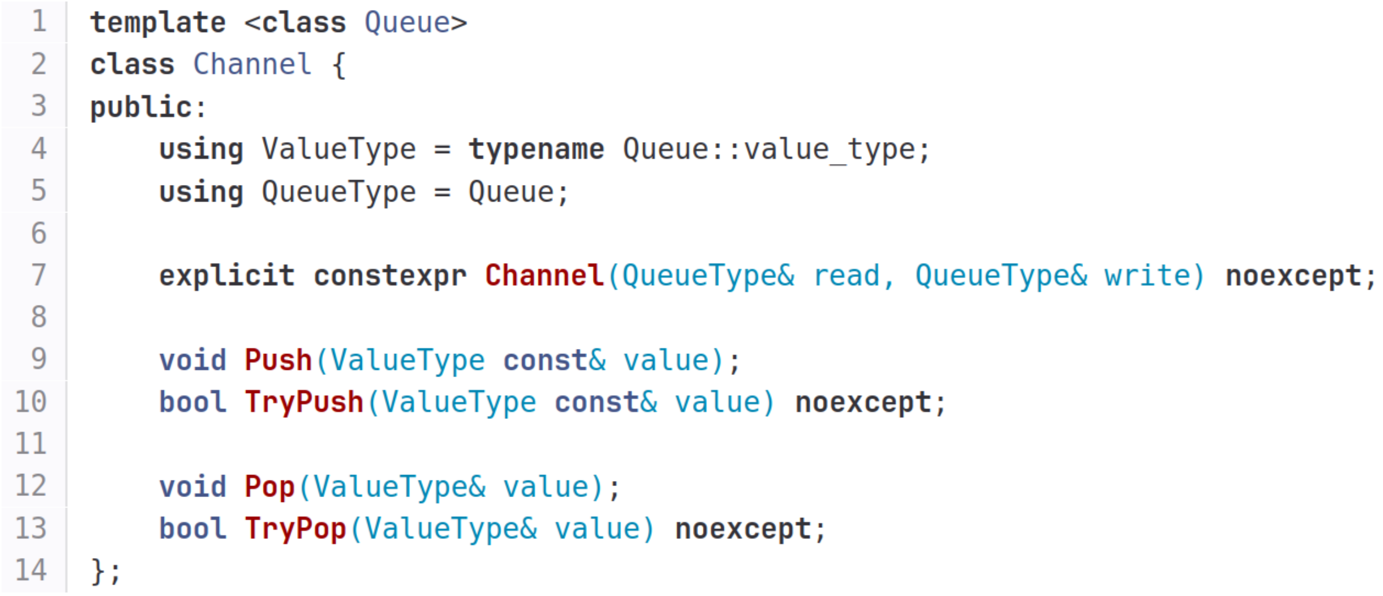
Data Classification: PUBLIC

Channels

WFRTC composition of two directional queues

- Second "free-list" queue
- Two queues forms a *Channel*
- `boost::lockfree::spsc_queue`



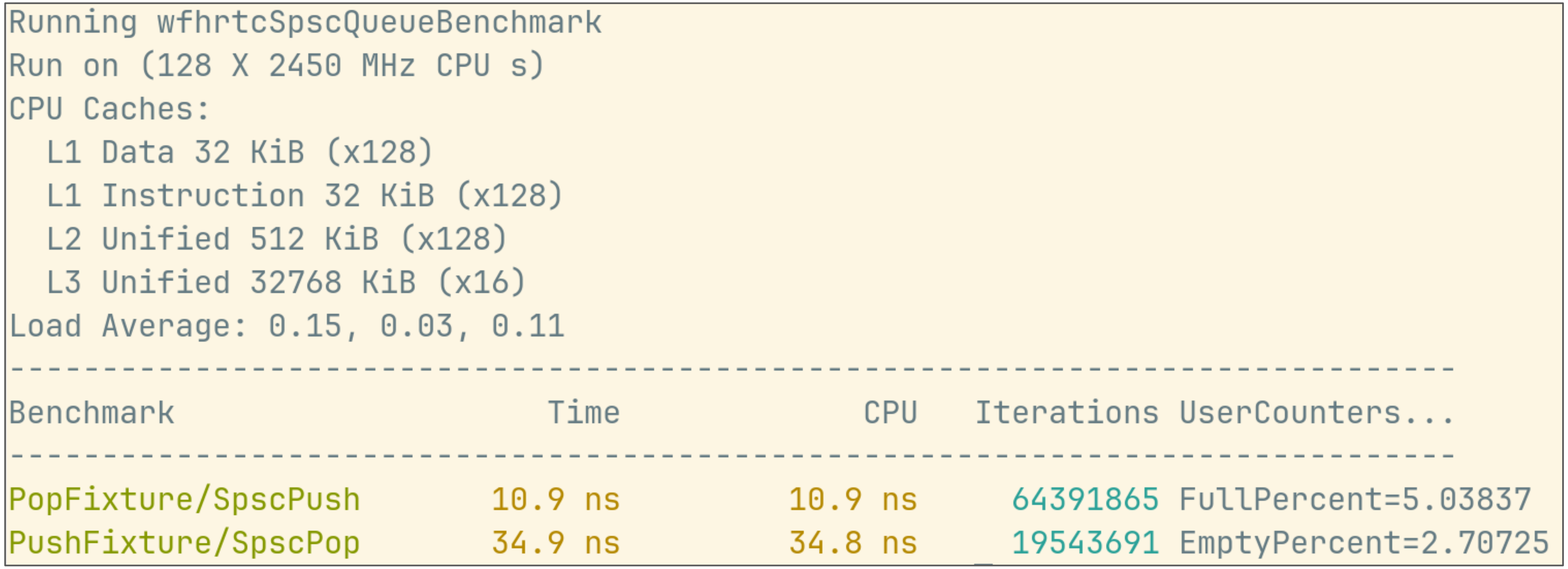


```cpp
template <class Queue>
class Channel {
public:
    using ValueType = typename Queue::value_type;
    using QueueType = Queue;

    explicit constexpr Channel(QueueType& read, QueueType& write) noexcept;

    void Push(ValueType const& value);
    bool TryPush(ValueType const& value) noexcept;

    void Pop(ValueType& value);
    bool TryPop(ValueType& value) noexcept;
};
```

Benchmark

- Measures push/pop operations under maximum contention

- Unpinned threads on unisolated cores



```
Running wfhrtcSpscQueueBenchmark
Run on (128 X 2450 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x128)
  L1 Instruction 32 KiB (x128)
  L2 Unified 512 KiB (x128)
  L3 Unified 32768 KiB (x16)
Load Average: 0.15, 0.03, 0.11
------------------------------------------------------------------------------
Benchmark                      Time             CPU   Iterations UserCounters...
------------------------------------------------------------------------------
PopFixture/SpscPush         10.9 ns         10.9 ns     64391865 FullPercent=5.03837
PushFixture/SpscPop         34.9 ns         34.8 ns     19543691 EmptyPercent=2.70725
```

AMD EPYC 7763 (128 Core Zen 3 Milan)

Scatter-gather

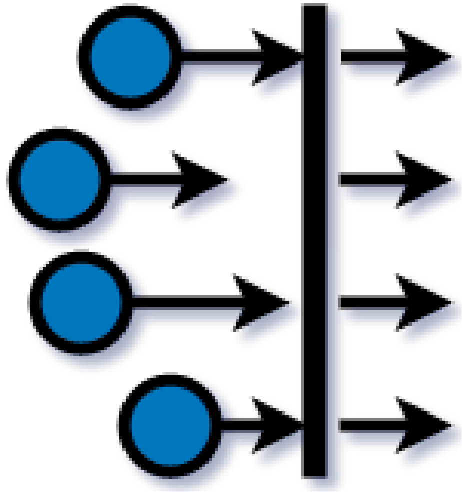
Data Classification: PUBLIC

Scatter-gather

WFRTC use this to

- Offload work to multiple threads
- Gather results when all threads have completed

Achieved using the *barrier* primitive

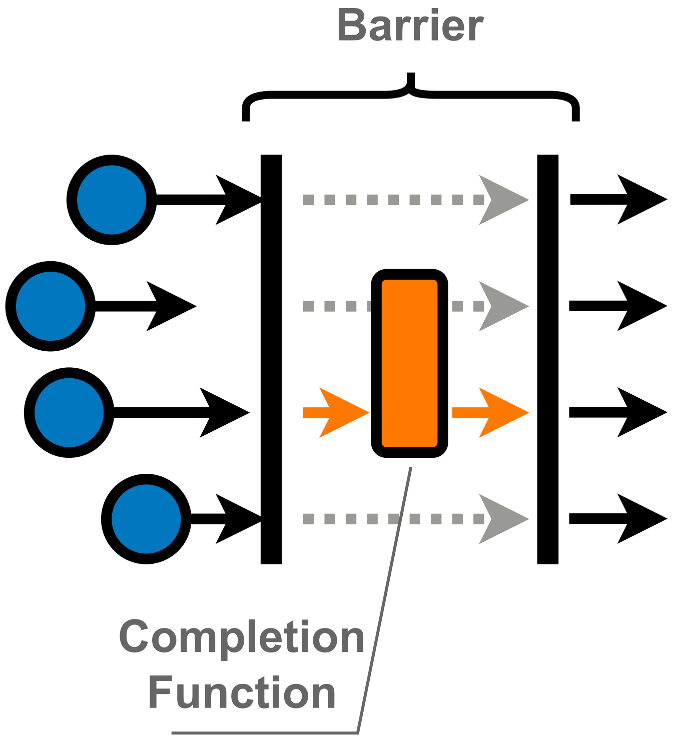


Data Classification: PUBLIC

Barrier

Introducing the barrier

- Threads *arrive and wait*
- When last thread arrives a *Completion Function* is invoked by one of the threads
  - *Completion Function* can **safely modify shared state**
- When *Completion Function* returns barrier is lifted and threads continue
- C++20 `std::barrier`



Data Classification: PUBLIC

Barriers in WFRTC

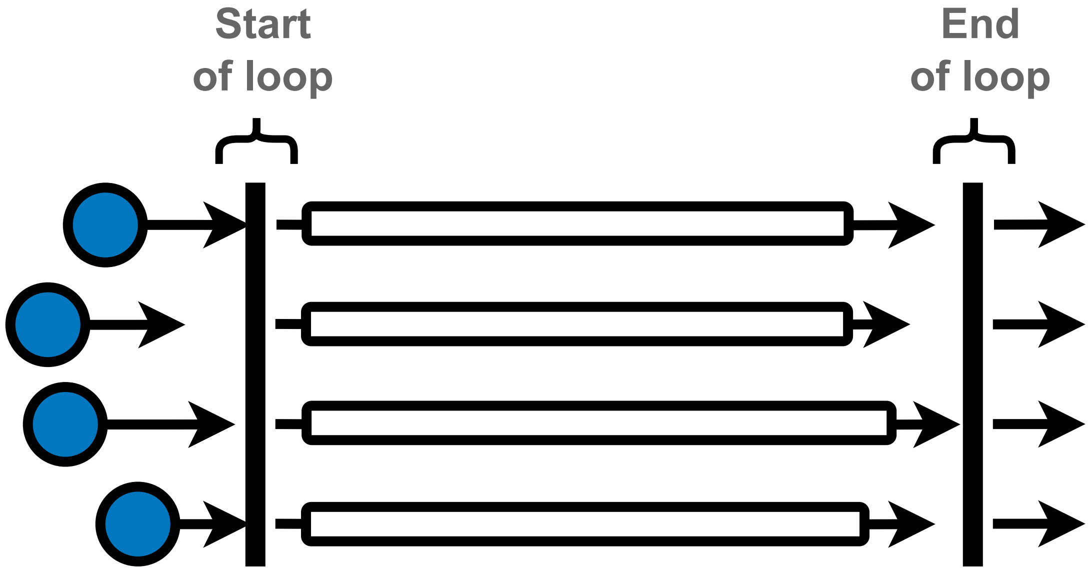
Computation loop

Start of loop barrier

- Wait for start condition, e.g.
  - Inputs from *channel*
  - State change *signal*
- Distribute data or signal to threads

End of loop barrier

- Gather results
- Push result to output *channel*

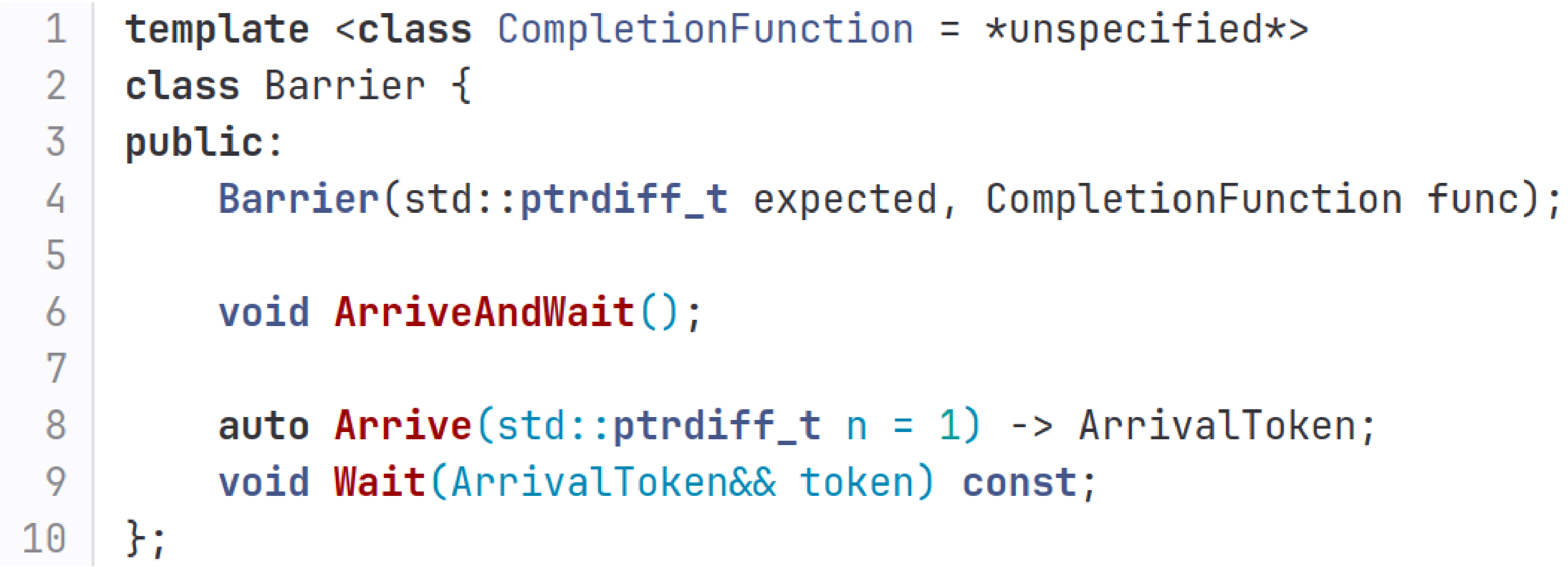


Data Classification: PUBLIC

Support library: ion

ion::Barrier

- Semantics like `std::barrier`
- Busy-waits



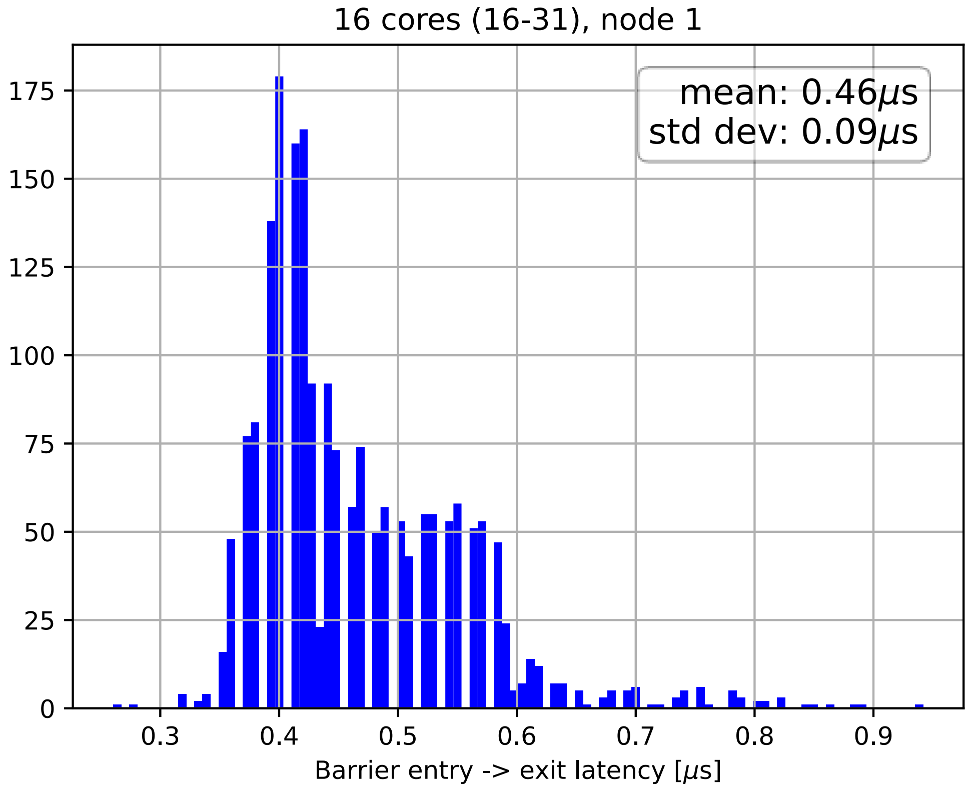
```
template <class CompletionFunction = *unspecified*>
class Barrier {
public:
    Barrier(std::ptrdiff_t expected, CompletionFunction func);

    void ArriveAndWait();

    auto Arrive(std::ptrdiff_t n = 1) -> ArrivalToken;
    void Wait(ArrivalToken&& token) const;
};
```

Support library: ion

Benchmark

One measure of synchronization latency

- Time between last thread arriving and first thread leaving barrier
- Empty completion function
- Last 2'000 samples from 10'000 iterations

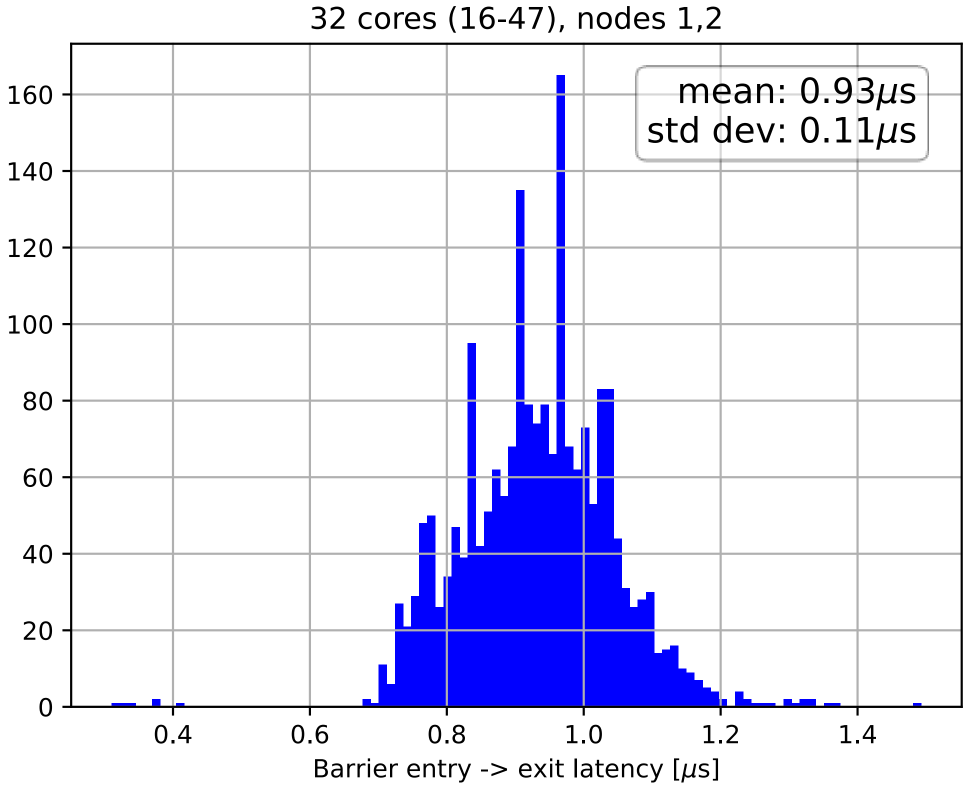


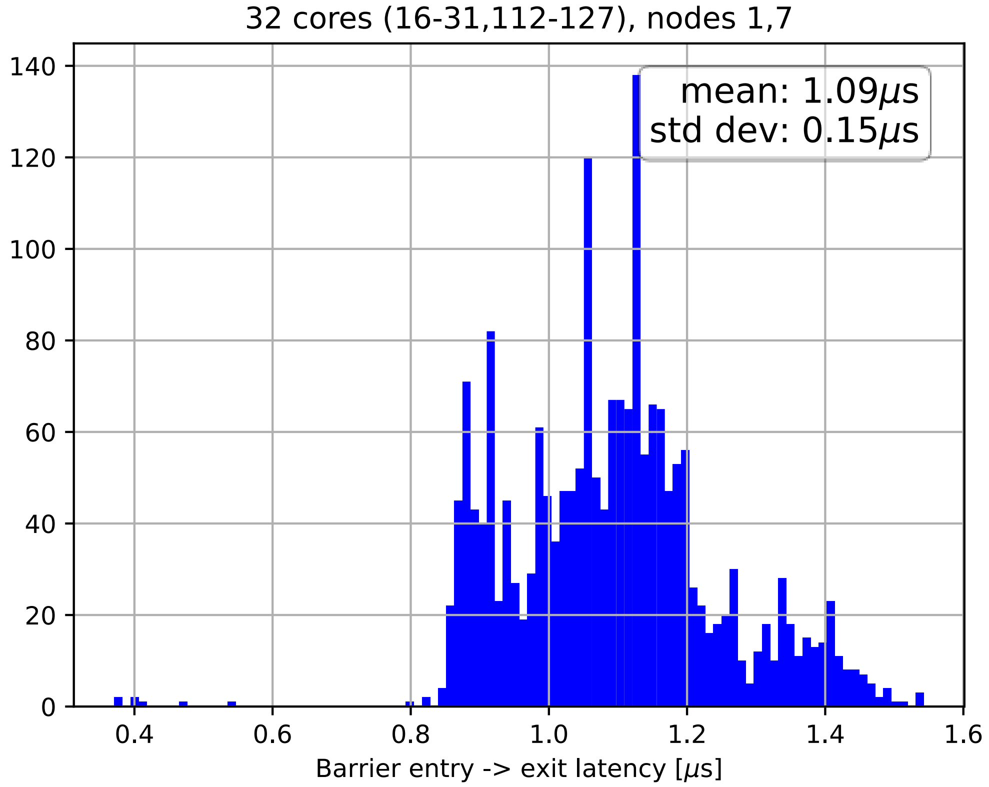
AMD EPYC 7763 (128 Core Zen 3 Milan)

Data Classification: PUBLIC

Support library: ion

As threads and NUMA distance increase so does latency





AMD EPYC 7763 (128 Core Zen 3 Milan)

Data Classification: PUBLIC

Signals

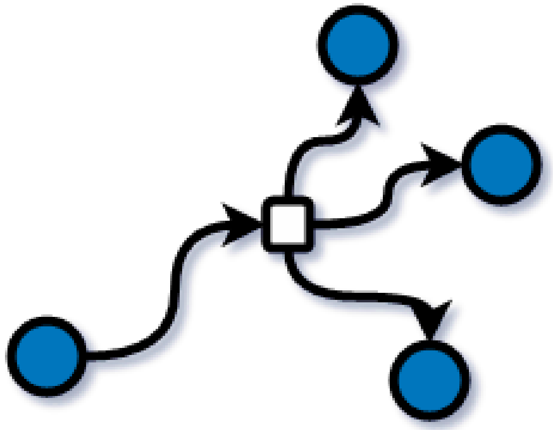
Data Classification: PUBLIC

Signals

Notify, observe, wait

WFRTC use signals to

- Request state change (state enumeration)
- Indicate current state (state enumeration)
- Trigger telemetry sending (sample id)



Data Classification: PUBLIC

Support library: ion

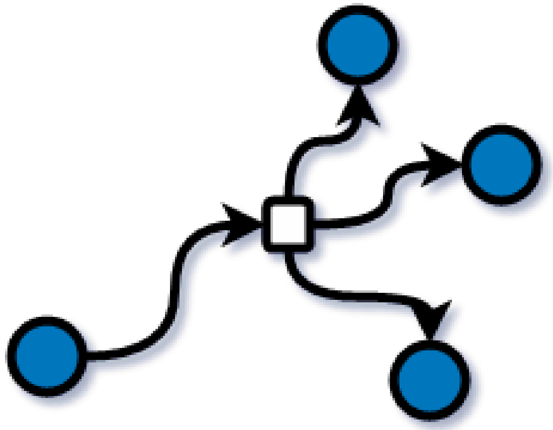
Thin abstraction on top of `std::atomic<T>`

`ion::SignalSource<T>`

- Lightweight thread safe, lock-free atomic value of type T
- No synchronization or order constraints

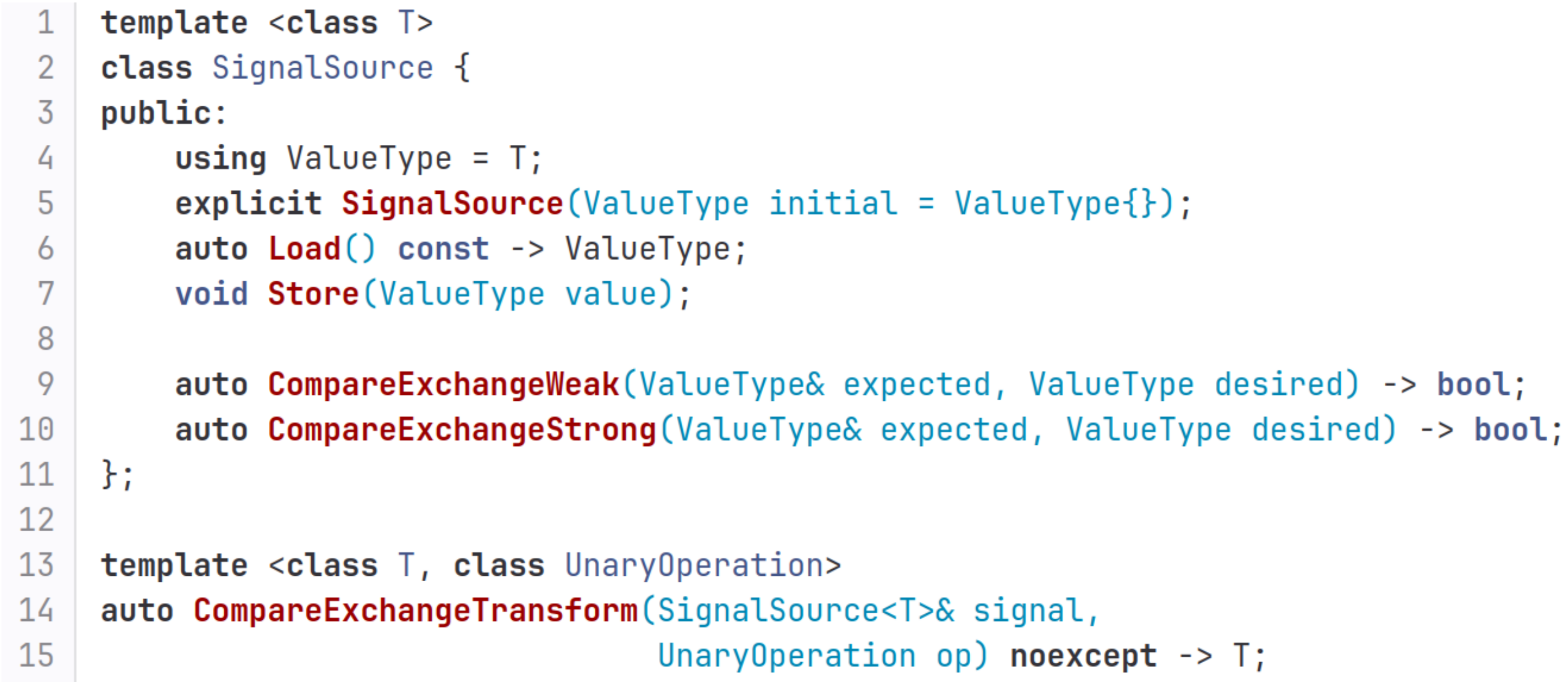
`ion::SignalToken<T>`

- Keeps association with source and remember last value



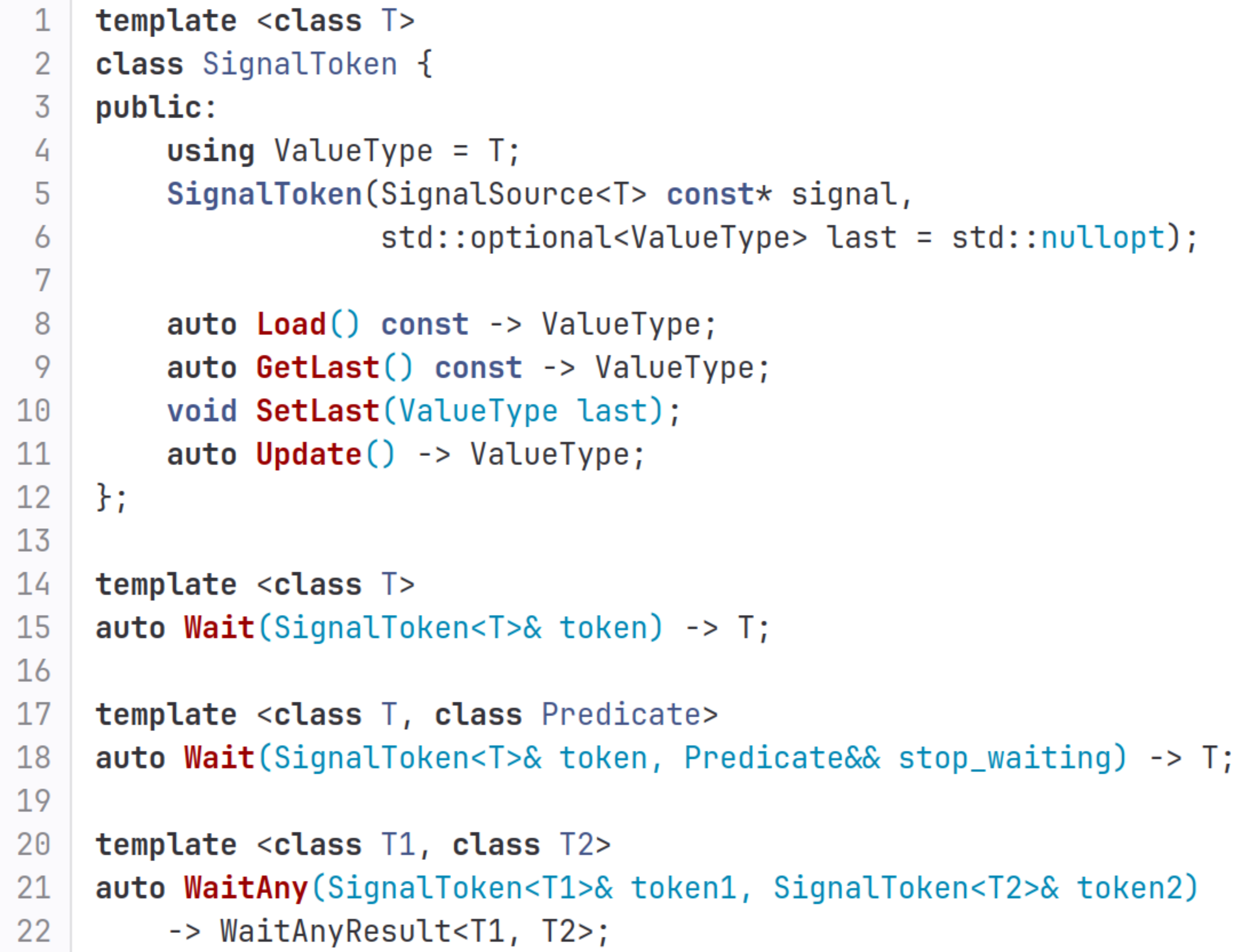
Data Classification: PUBLIC

Support library: ion



```cpp
template <class T>
class SignalSource {
public:
    using ValueType = T;
    explicit SignalSource(ValueType initial = ValueType{});
    auto Load() const -> ValueType;
    void Store(ValueType value);

    auto CompareExchangeWeak(ValueType& expected, ValueType desired) -> bool;
    auto CompareExchangeStrong(ValueType& expected, ValueType desired) -> bool;
};

template <class T, class UnaryOperation>
auto CompareExchangeTransform(SignalSource<T>& signal,
                              UnaryOperation op) noexcept -> T;
```

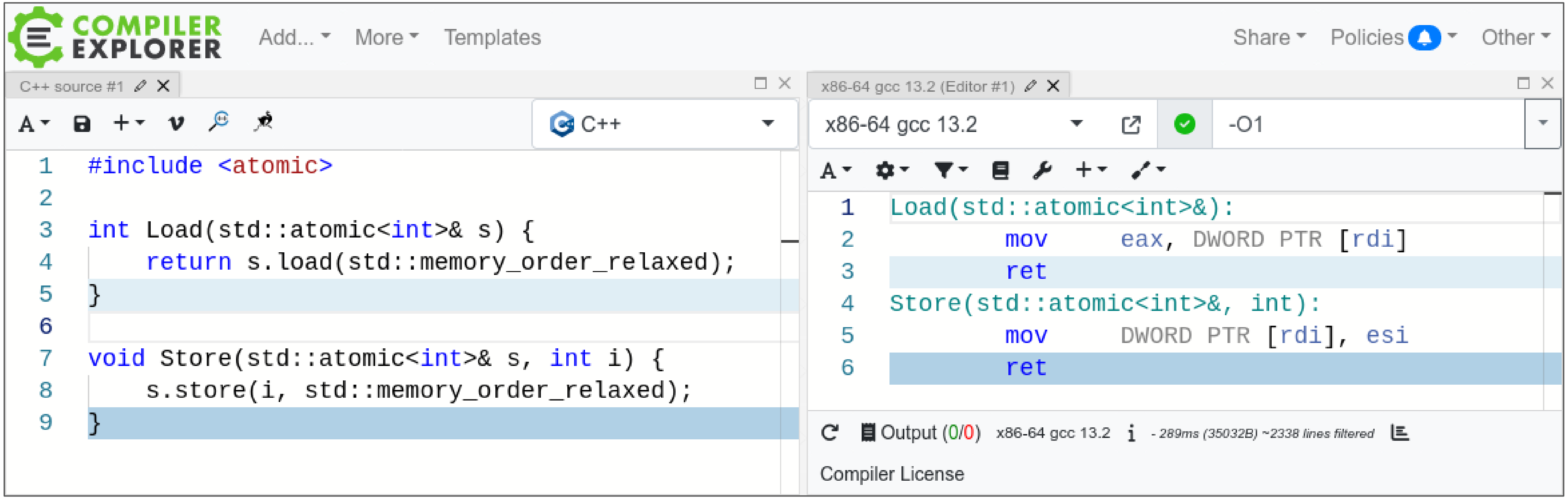
Support library: ion



```cpp
template <class T>
class SignalToken {
public:
    using ValueType = T;
    SignalToken(SignalSource<T> const* signal,
                std::optional<ValueType> last = std::nullopt);

    auto Load() const -> ValueType;
    auto GetLast() const -> ValueType;
    void SetLast(ValueType last);
    auto Update() -> ValueType;
};

template <class T>
auto Wait(SignalToken<T>& token) -> T;

template <class T, class Predicate>
auto Wait(SignalToken<T>& token, Predicate&& stop_waiting) -> T;

template <class T1, class T2>
auto WaitAny(SignalToken<T1>& token1, SignalToken<T2>& token2)
    -> WaitAnyResult<T1, T2>;
```

Support library: ion

Signals under the hood



```cpp
#include <atomic>

int Load(std::atomic<int>& s) {
    return s.load(std::memory_order_relaxed);
}

void Store(std::atomic<int>& s, int i) {
    s.store(i, std::memory_order_relaxed);
}
```

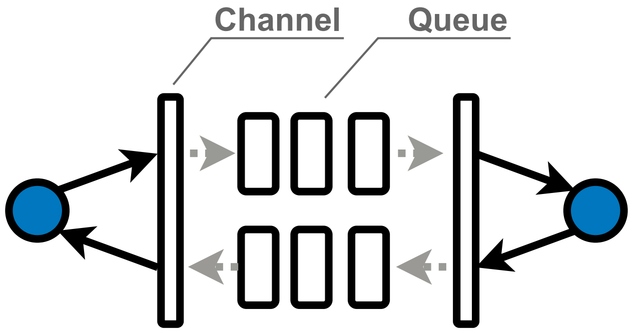
```
Load(std::atomic<int>&):
        mov     eax, DWORD PTR [rdi]
        ret
Store(std::atomic<int>&, int):
        mov     DWORD PTR [rdi], esi
        ret
```

Output (0/0) x86-64 gcc 13.2 - 289ms (35032B) ~2338 lines filtered

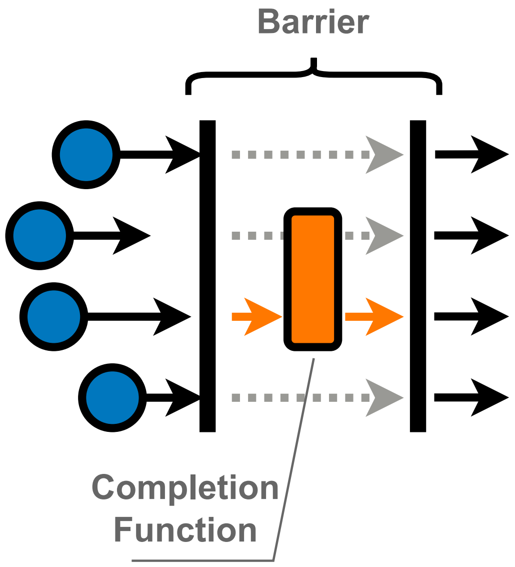
Compiler License

https://godbolt.org/z/6d7jMfefW

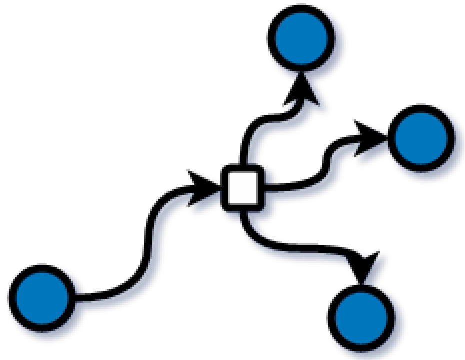
Summary



1:1



1:N:1



N:M

Thank you!

Calle Rosenquist

crosenqu@eso.org

Boost.Lockfree
www.boost.org/doc/libs/1_83_0/doc/html/lockfree.html

NUMA++ & ion
gitlab.eso.org/rtctk/roadrunner

@ESOAstronomy

@esoastronomy

@ESO

european-southern-observatory

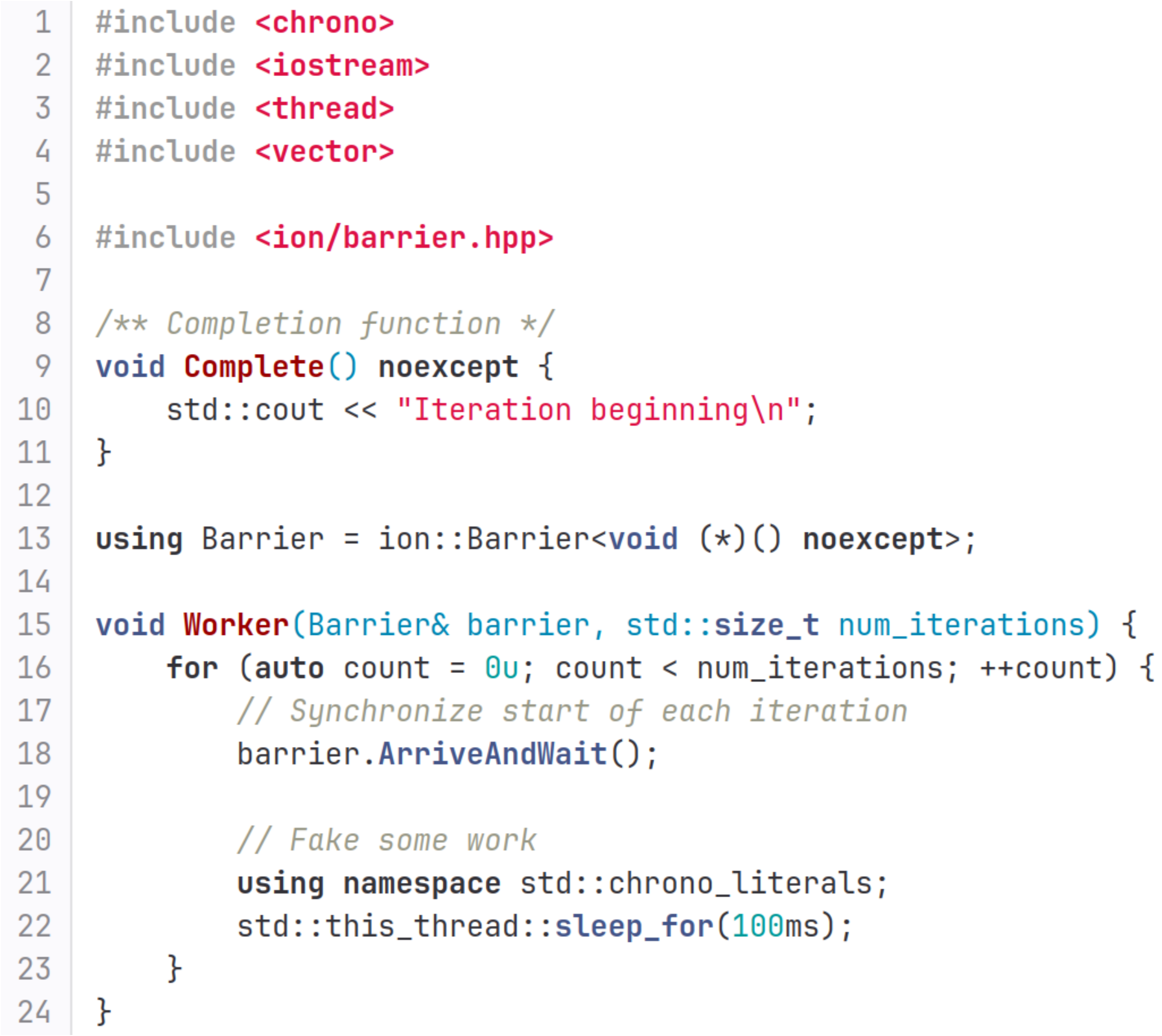
@ESOobservatory

Data Classification: PUBLIC

Bonus Slides

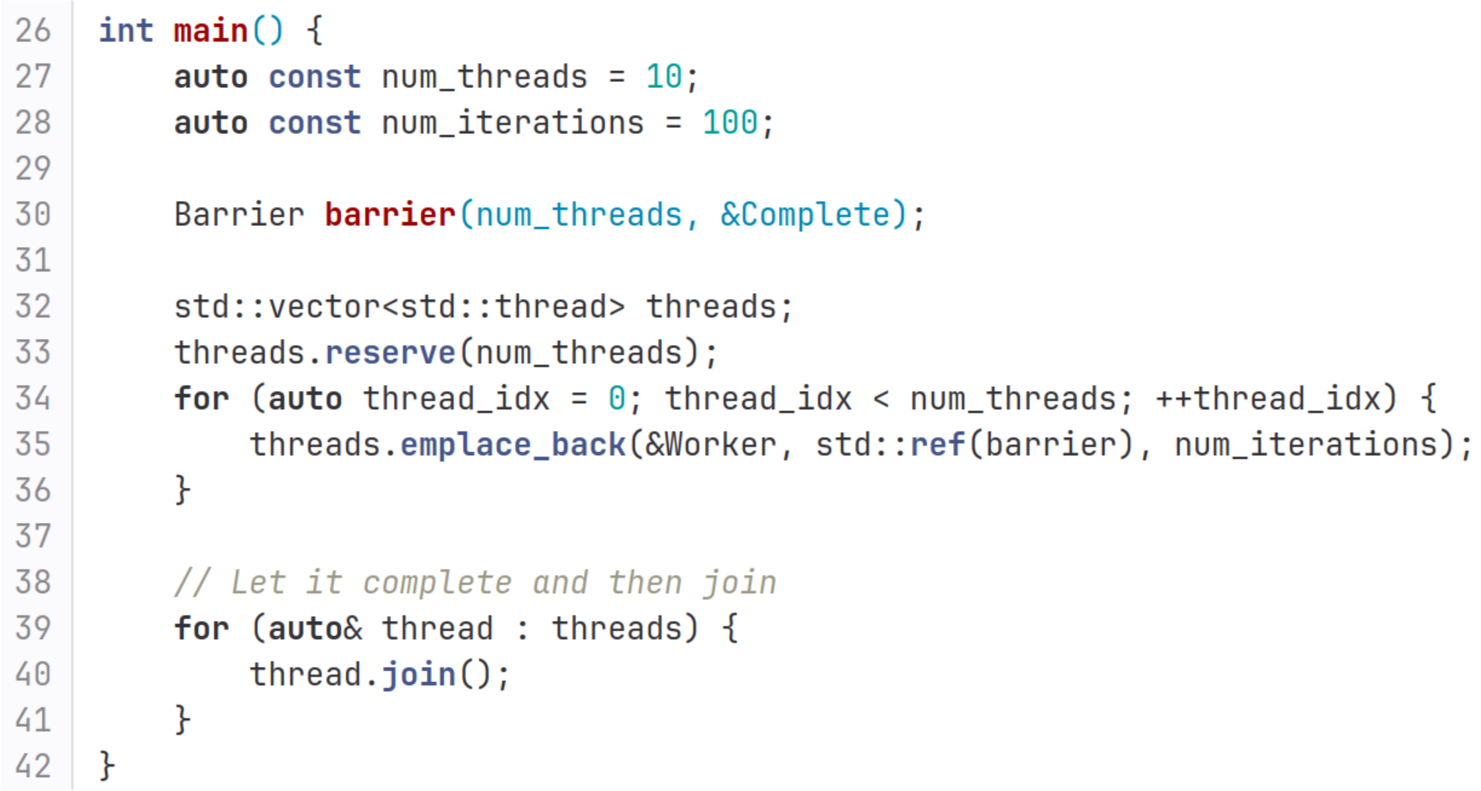
Data Classification: PUBLIC

Barrier Example 1/2



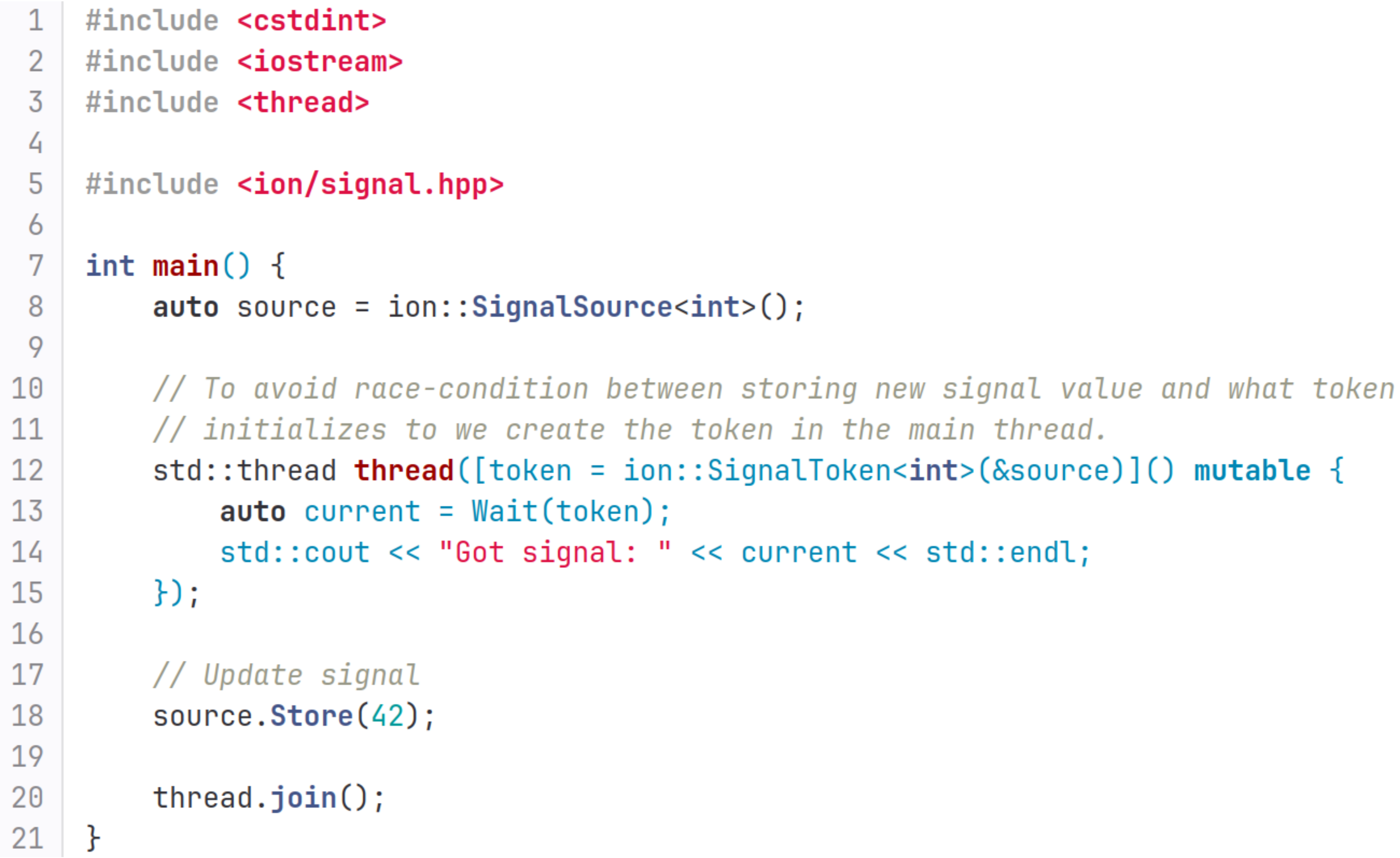
```cpp
#include <chrono>
#include <iostream>
#include <thread>
#include <vector>

#include <ion/barrier.hpp>

/** Completion function */
void Complete() noexcept {
    std::cout << "Iteration beginning\n";
}

using Barrier = ion::Barrier<void (*)() noexcept>;

void Worker(Barrier& barrier, std::size_t num_iterations) {
    for (auto count = 0u; count < num_iterations; ++count) {
        // Synchronize start of each iteration
        barrier.ArriveAndWait();

        // Fake some work
        using namespace std::chrono_literals;
        std::this_thread::sleep_for(100ms);
    }
}
```

Barrier Example 2/2



```cpp
int main() {
    auto const num_threads = 10;
    auto const num_iterations = 100;

    Barrier barrier(num_threads, &Complete);

    std::vector<std::thread> threads;
    threads.reserve(num_threads);
    for (auto thread_idx = 0; thread_idx < num_threads; ++thread_idx) {
        threads.emplace_back(&Worker, std::ref(barrier), num_iterations);
    }

    // Let it complete and then join
    for (auto& thread : threads) {
        thread.join();
    }
}
```

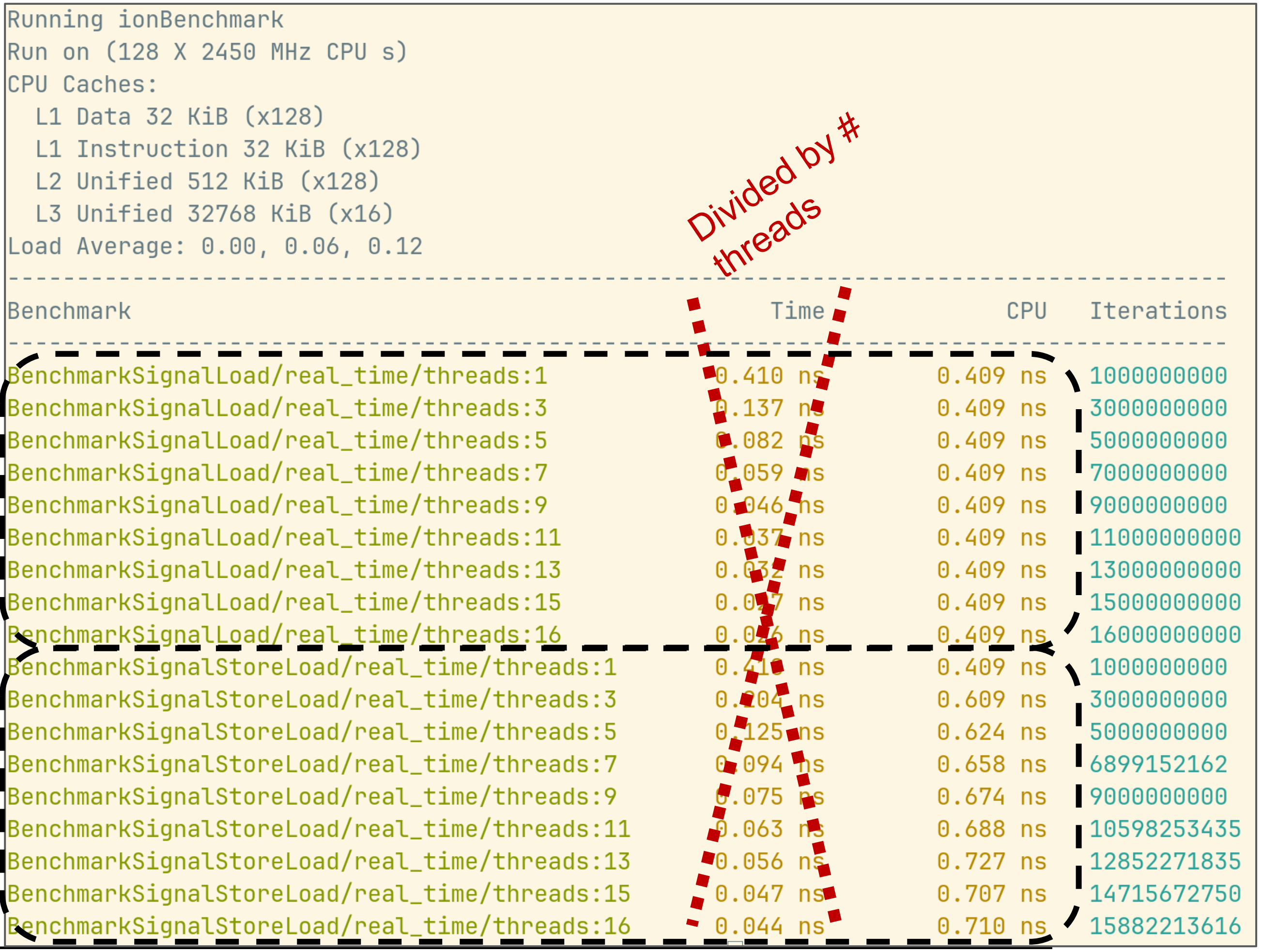
Signal Example



```cpp
#include <cstdint>
#include <iostream>
#include <thread>

#include <ion/signal.hpp>

int main() {
    auto source = ion::SignalSource<int>();

    // To avoid race-condition between storing new signal value and what token
    // initializes to we create the token in the main thread.
    std::thread thread([token = ion::SignalToken<int>(&source)]() mutable {
        auto current = Wait(token);
        std::cout << "Got signal: " << current << std::endl;
    });

    // Update signal
    source.Store(42);

    thread.join();
}
```

Support library: ion

Benchmark



```
Running ionBenchmark
Run on (128 X 2450 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x128)
  L1 Instruction 32 KiB (x128)
  L2 Unified 512 KiB (x128)
  L3 Unified 32768 KiB (x16)
Load Average: 0.00, 0.06, 0.12
-----------------------------------------------------------------------------------------
Benchmark                                            Time             CPU   Iterations
-----------------------------------------------------------------------------------------
BenchmarkSignalLoad/real_time/threads:1          0.410 ns        0.409 ns   1000000000
BenchmarkSignalLoad/real_time/threads:3          0.137 ns        0.409 ns   3000000000
BenchmarkSignalLoad/real_time/threads:5          0.082 ns        0.409 ns   5000000000
BenchmarkSignalLoad/real_time/threads:7          0.059 ns        0.409 ns   7000000000
BenchmarkSignalLoad/real_time/threads:9          0.046 ns        0.409 ns   9000000000
BenchmarkSignalLoad/real_time/threads:11         0.037 ns        0.409 ns   11000000000
BenchmarkSignalLoad/real_time/threads:13         0.032 ns        0.409 ns   13000000000
BenchmarkSignalLoad/real_time/threads:15         0.027 ns        0.409 ns   15000000000
BenchmarkSignalLoad/real_time/threads:16         0.026 ns        0.409 ns   16000000000
BenchmarkSignalStoreLoad/real_time/threads:1     0.410 ns        0.409 ns   1000000000
BenchmarkSignalStoreLoad/real_time/threads:3     0.204 ns        0.609 ns   3000000000
BenchmarkSignalStoreLoad/real_time/threads:5     0.125 ns        0.624 ns   5000000000
BenchmarkSignalStoreLoad/real_time/threads:7     0.094 ns        0.658 ns   6899152162
BenchmarkSignalStoreLoad/real_time/threads:9     0.075 ns        0.674 ns   9000000000
BenchmarkSignalStoreLoad/real_time/threads:11    0.063 ns        0.688 ns   10598253435
BenchmarkSignalStoreLoad/real_time/threads:13    0.056 ns        0.727 ns   12852271835
BenchmarkSignalStoreLoad/real_time/threads:15    0.047 ns        0.707 ns   14715672750
BenchmarkSignalStoreLoad/real_time/threads:16    0.044 ns        0.710 ns   15882213616
```

Under varying number of unpinned threads:

- `SignalSource::Load()`

- `SignalSource::Store()` [thread 1]
  `SignalSource::Load()` [threads 2..N-1]

AMD EPYC 7763 (128 Core Zen 3 Milan)

Data Classification: PUBLIC