# Interesting Upcoming Features from Low Latency, Parallelism and Concurrency

## From Kona 2023, Tokyo 2024, and St. Louis 2024

PAUL E. MCKENNEY,
MAGED MICHAEL
& MICHAEL WONG

Cppcon
The C++ Conference

2024
September 15 - 20

## Agenda

1. Improving C++20 Atomic Min/Max(P0493; Michael)
2. Hazard pointer extensions (P3135; Maged)
3. Pointer tagging (P3125; Maged)
4. Parallel Range algorithms (P3179; Michael), may be Parallel Algorithms (P2500)

# C++26: Atomic Min/Max

# atomic min/max (P0493)

# Atomic min/max motivation (P0493)

Long history - almost as old as atomic addition

Multithreaded applications often involve scenarios where multiple threads need to concurrently update a shared variable to track the minimum or maximum value.

Without atomic operations, race conditions can occur, leading to data corruption and unpredictable behavior.

Atomic min/max operations provide a solution by ensuring that updates to the shared variable are performed atomically, guaranteeing data integrity.

Efficient and safe concurrent updates to shared variables.

Enable the implementation of lock-free data structures, leading to improved performance and scalability.

Useful in various applications, including reductions in data-parallel algorithms, statistics collection, and optimization processes.

Useful for:

- Lock-free data structures
- Parallel reductions (OpenMP)
- Optimization algorithms
- Statistics collection

# Proposed interface

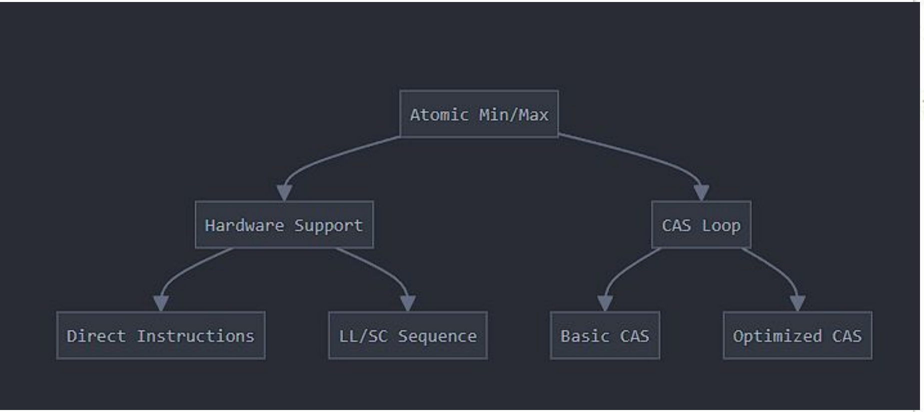
```
namespace std {

template<class T>
T atomic_fetch_max(atomic<T>*,
                   typename atomic<T>::value_type) noexcept;

template<class T>
T atomic_fetch_max_explicit(atomic<T>*,
                            typename atomic<T>::value_type,
                            memory_order) noexcept;

// Similar for atomic_fetch_min
// Member functions on atomic<T>

 }
```

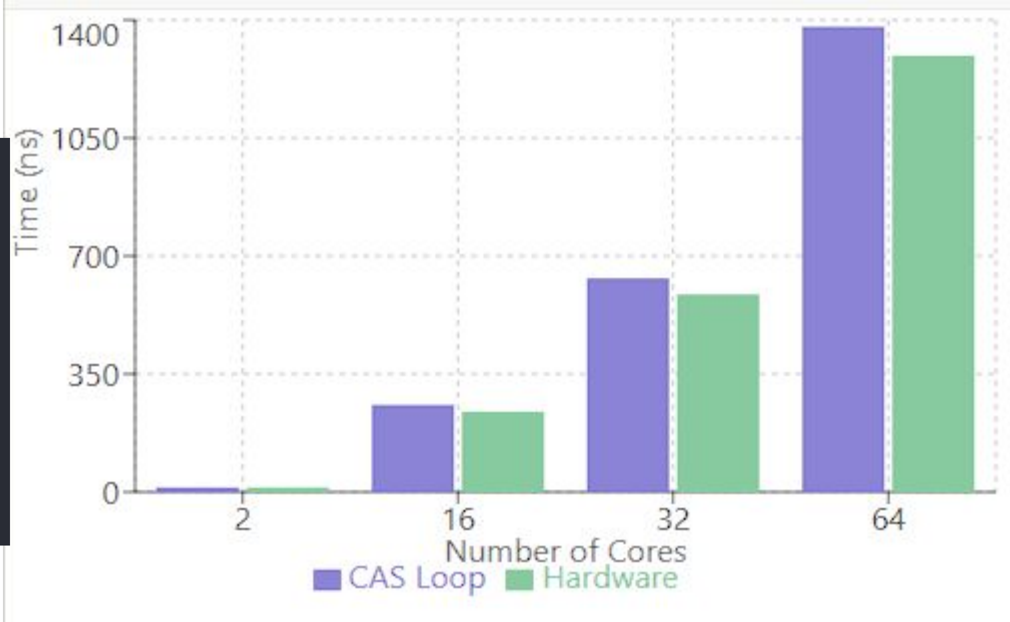
# Performance Considerations & Benchmark Results

Hardware support can be significantly faster

CAS loops have higher contention

But optimized CAS can be competitive



# Example Usage

```cpp
#include <atomic>
#include <thread>
#include <vector>

std::atomic<int> max_value{0};

void find_max_in_range(int
start, int end) {
    for (int i = start; i <
end; ++i) {

    max_value.fetch_max(i,
    std::memory_order_relaxed)
    ;
    }
}
```

```cpp
int main() {
    std::vector<std::thread> threads;
    for (int i = 0; i < 10; ++i) {

threads.emplace_back(find_max_in_range
, i * 100, (i + 1) * 100);
    }
    for (auto& thread : threads) {
        thread.join();
    }
    std::cout << "Maximum value: " <<
max_value << std::endl;
    return 0;
 }
```

# Memory Ordering


`std::memory_order_relaxed`: No synchronization or ordering constraints.

`std::memory_order_acquire`: Ensures that subsequent loads see the effects of this operation and any prior release operations.

`std::memory_order_release`: Makes the effects of this operation and any prior operations visible to subsequent acquire operations.

`std::memory_order_acq_rel`: Combines acquire and release semantics.

`std::memory_order_seq_cst`: Provides the strongest ordering guarantees, ensuring a globally consistent view of memory across all threads.

# Current Status, Next Steps, & Conclusion

Proposal in flight since 2016, aiming for C++26

Current revision: P0493R5 (February 2024)

Implementation experience in Clang

Open questions:

- Floating point support?
- New-value-returning variants?

- Atomic min/max operations are valuable additions to the C++ standard library.
- They enable efficient and safe concurrent updates to shared variables, facilitating the development of high-performance and scalable multithreaded applications.
- By understanding the semantics and proper usage of atomic min/max, you can leverage their power to build robust and reliable concurrent code.

# Atomic Floating-Point Min/Max in C++ (P3008)

**Atomic min/max operations are crucial for concurrent algorithms**

**Integer versions added in C++20**

**Floating-point versions were removed due to concerns**

**The Challenge of Floating-Point Corner Cases**

- NaN (Not a Number): Represents undefined or unrepresentable values.
- Signed Zero: Both +0 and -0 exist, but their ordering can be ambiguous

**Evolution of Floating-Point Min/Max in C++**

- `std::min`, `std::max`: Undefined behavior with NaNs.
    a. std::min(-0.0f, +0.0f); // Returns -0.0f
    b. std::min(NaN, 2.0f);    // Undefined behavior
- `fmin`, `fmax`: Treat NaNs as missing data, signed zero behavior can vary.
- `fminimum`, `fmaximum`: Treat NaNs as errors, -0 < +0.
- `fminimum_num`, `fmaximum_num`: Treat NaNs as missing data, -0 < +0.

## Hardware Support

- Many GPUs have native atomic float min/max
- CPU support growing (e.g. ARM v8.1)
- Hardware treats -0 < +0 and NaN as "missing data"

# Atomic Floating-Point Min/Max

`atomic<T>::fetch_min`, `atomic<T>::fetch_max`: Proposed atomic operations.

float fetch_min(float, memory_order = memory_order::seq_cst) noexcept;

float fetch_max(float, memory_order = memory_order::seq_cst) noexcept;

Semantics:

- Unspecified behavior with NaNs.
- Treats -0 < +0 (recommended, not required)
- Based on C23's fminimum_num/fmaximum_num

float fetch_fminimum(float, memory_order = memory_order::seq_cst) noexcept;

float fetch_fmaximum(float, memory_order = memory_order::seq_cst) noexcept;

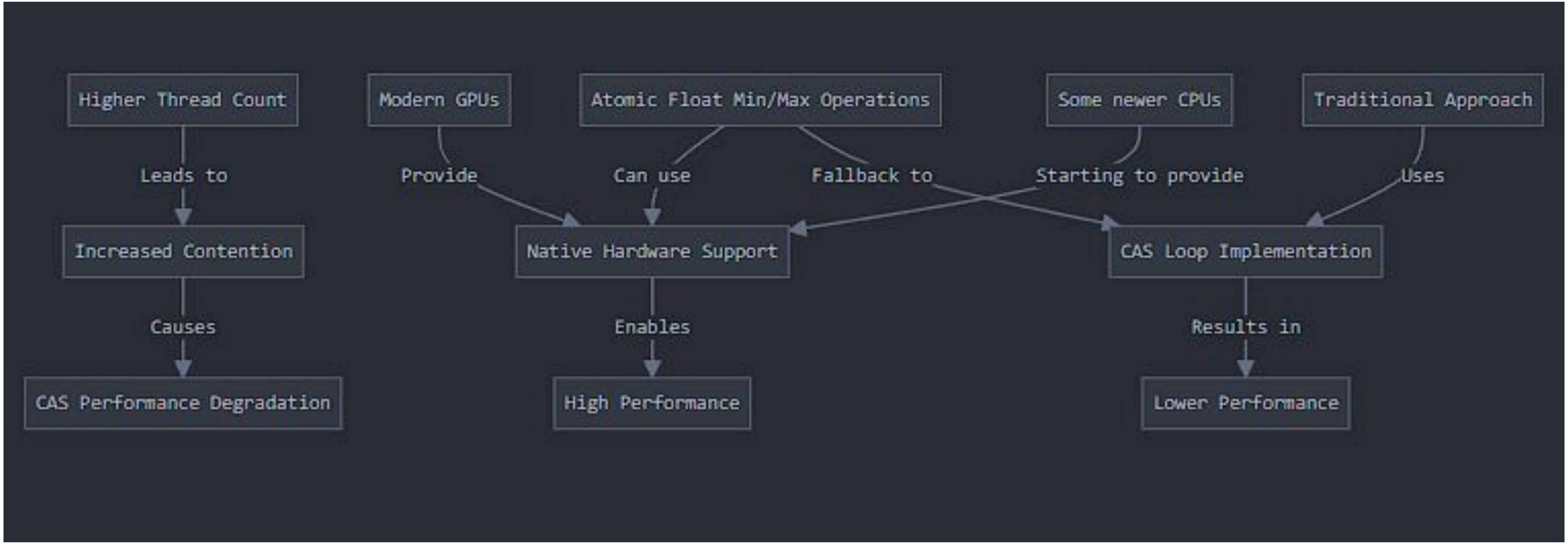
float fetch_fminimum_num(float, memory_order = memory_order::seq_cst) noexcept;

float fetch_fmaximum_num(float, memory_order = memory_order::seq_cst) noexcept;

# Performance Considerations

Native atomic operations vs. CAS loops.

Significant performance difference in highly concurrent systems.



## Code Examples

```
#include <atomic>
#include <cmath>

#include <iostream>

int main() {

  std::atomic<float> atomic_value(10.0f);

  float new_max = 15.0f;

  float result_max = atomic_value.fetch_max(new_max);

   std::cout << "Old max: " << result_max << ", New max: "
<< atomic_value << std::endl;

   float new_min = 5.0f;

   float result_min = atomic_value.fetch_min(new_min);

    std::cout << "Old min: " << result_min << ", New min: "
<< atomic_value << std::endl;

    return 0;

}
```

Expected output:

```
Old max: 10, New max: 15
 Old min: 15, New min: 15
```

# Conclusion

Atomic floating-point min/max: Essential for modern concurrent systems

Careful consideration of corner cases: NaNs and signed zeros

Performance implications: Native atomic operations vs. CAS loops

New atomic float min/max operations

Semantics close to hardware capabilities

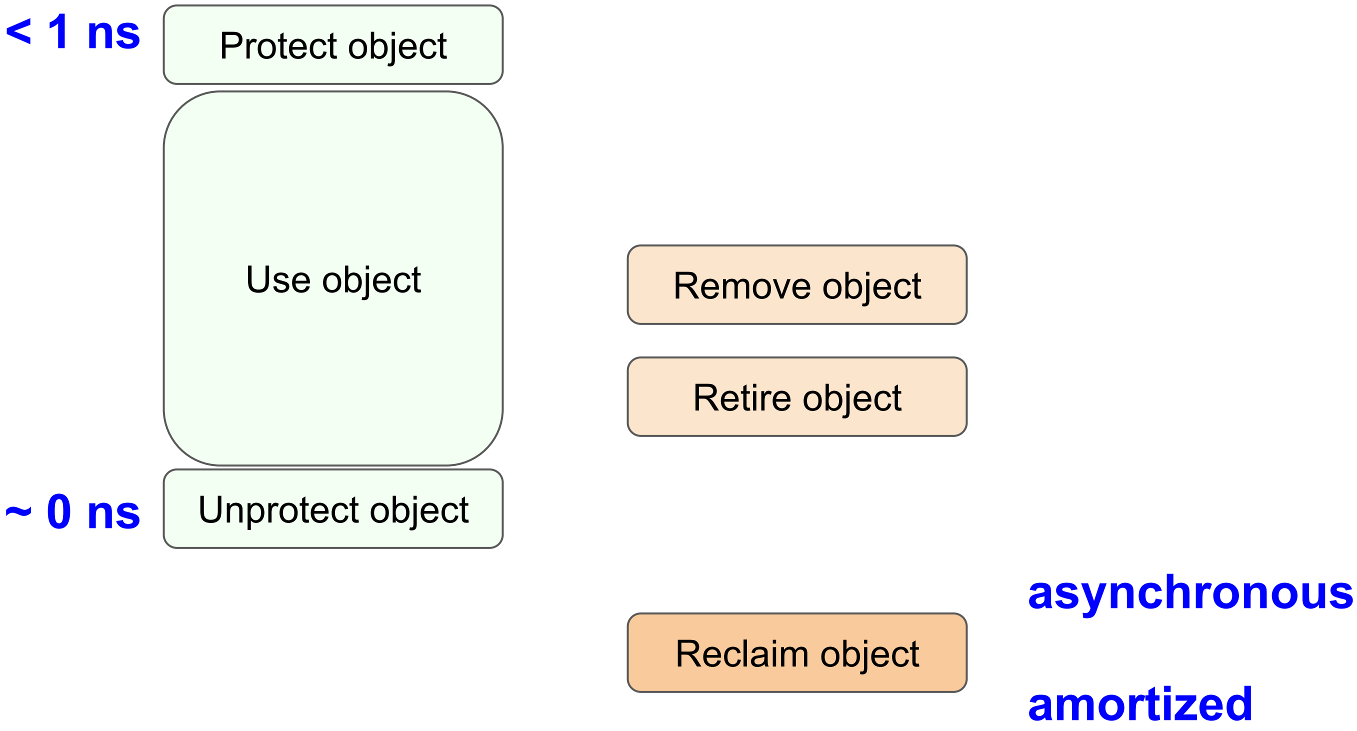
Significant performance benefits

Additional functions for more specific needs

# Hazard Pointer Extensions beyond C++26

# Hazard Pointers in C++26 -- Background

Hazard pointers **protect** dynamic **objects from being reclaimed**, allowing **safe access** to protected objects **without** additional **synchronization**.



# Hazard Pointers C++26

```
template <typename T, typename D = default_delete<T>>
class hazard_pointer_obj_base {
  void retire(D d = D()) noexcept;
};
```

```
class hazard_pointer {
  hazard_pointer() noexcept; // construct empty hazard_pointer
  hazard_pointer(hazard_pointer&&) noexcept;
  hazard_pointer& operator=(hazard_pointer&&) noexcept;
  ~hazard_pointer();
  [[nodiscard]] bool empty() const noexcept;
  template <typename T> T* protect(const atomic<T*>& src) noexcept;
  template <typename T> bool try_protect(T*& ptr, const atomic<T*>& src) noexcept;
  template <typename T> void reset_protection(const T* ptr) noexcept;
  void reset_protection(nullptr_t = nullptr) noexcept;
  void swap(hazard_pointer&) noexcept;
};
```

```
hazard_pointer make_hazard_pointer(); // construct nonempty hazard_pointer
void swap(hazard_pointer&, hazard_pointer&) noexcept;
```

# Example Using C++26 Hazard Pointers

```
class T : public hazard_pointer_obj_base<T> { /* T members */ };
```

```
std::atomic<T*> src_;
```

```
U readAndAccess(Func userFn) { // Called frequently
 hazard_pointer hp = make_hazard_pointer(); // Construct hazard pointer.
 T* ptr = hp.protect(src_);                 // Get pointer to a protected object.
 return userFn(ptr);
}

Void update(T* newptr) { // Called infrequently
  T* oldptr = src_.exchange(newptr);
  oldptr->retire();  // Pass to hazard pointer library for safe reclamation.
}
```

# P3135R1: Hazard Pointer Extensions (beyond C++26)

**P3135R1: Hazard Pointer Extensions** (`wg21.link/p3135r1`)

**No need for extending C++26:**

- **Protection Counting** (can be a topic for a future talk)
- **Execution of Asynchronous Reclamation**

**Proposed standard extensions**:

- **Synchronous reclamation**
- **Batch creation and destruction**

# Hazard Pointer
# Execution  of Asynchronous Reclamation

# Hazard Pointer Execution of Asynchronous Reclamation

## Using C++26

Possible inline asynchronous reclamation

```
void worker() {
  /* ... */
  obj->retire();
  // Possible inline reclamation.
}
```

Using a separate asynchronous reclamation executor

```
void worker() {
  /* ... */
  ex_.submit([obj] { obj->retire(); });
  // No inline reclamation.
}
```

# Hazard Pointer
# Synchronous Reclamation

# Hazard Pointer Synchronous Reclamation

## C++26 (Asynchronous Reclamation Only)

```
template <class T> class Container {
  class Obj : hazard_pointer_obj_base<Obj>
  { T data; /* etc */ };
  void insert(T data) {
    Obj* obj = new Obj(data); /* etc */ }
  void erase(Args args) {
    Obj* obj = find(args);
    /* Remove obj from container */
    obj->retire();
  }
};
class A {
  // Deleter does not depend on resources
  // with independent lifetime.
  ~A();
};

{ Container<A> container;
  container.insert(a);
  container.erase(a); }
// Obj containing 'a' may be not deleted yet.
```

**OK**

## Need Synchronous Reclamation

```
template <class T> class Container {
  class Obj : hazard_pointer_obj_base<Obj>
  { T data; /* etc */ };
  void insert(T data) {
    Obj* obj = new Obj(data); /* etc */ }
  void erase(Args args) {
    Obj* obj = find(args);
    /* Remove obj from container */
    obj->retire();
  }
};
class B {
  // Deleter may depend on resources
  // with independent lifetime.
  ~B() { use_resource_XYZ(); }
};
make_resource_XYZ();
{ Container<B> container;
  container.insert(b);
  container.erase(b); }
// Obj containing 'b' may be not deleted yet.
destroy_resource_XYZ();
```

**ERROR**

# Hazard Pointer Synchronous Reclamation

**Cohorts**:

- **Folly**: `folly/synchronization/Hazptr.h`
- **CPPCON 2021**: *Hazard Pointer Synchronous Reclamation*

## Possible API

```
class hazard_pointer_cohort {
  hazard_pointer_cohort() noexcept;
  hazard_pointer_cohort(const hazard_pointer_cohort&) = delete;
  hazard_pointer_cohort(hazard_pointer_cohort&&) = delete;
  hazard_pointer_cohort& operator=(const hazard_pointer_cohort&) = delete;
  hazard_pointer_cohort& operator=(hazard_pointer_cohort&&) = delete;
  ~hazard_pointer_cohort();
};
template <class T, class D = default_delete<T>>
class hazard_pointer_obj_base {
  void retire_to_cohort(hazard_pointer_cohort&, D d = D()) noexcept;      synchronous
};
void asynchronous_reclamation() noexcept;                                 asynchronous
```

# Hazard Pointer Synchronous Reclamation

## C++26 (Asynchronous Reclamation Only)

```
template <class T> class Container {
  class Obj : hazard_pointer_obj_base<Obj>
  { T data; /* etc */ };

  void insert(T data) {
    Obj* obj = new Obj(data); /* etc */ }
  void erase(Args args) {
    Obj* obj = find(args);
    /* Remove obj from container */
    obj->retire();

  }
};
class A {
  // Deleter does not depend on resources
  // with independent lifetime.
  ~A();
};

{ Container<A> container;
  container.insert(a);
  container.erase(a); }
// Obj containing 'a' may be not deleted yet.
```

**OK**

## Synchronous Reclamation

```
template <class T> class Container {
  class Obj : hazard_pointer_obj_base<Obj>
  { T data; /* etc */ };
  hazard_pointer_cohort cohort_;
  void insert(T data) {
    Obj* obj = new Obj(data); /* etc */ }
  void erase(Args args) {
    Obj* obj = find(args);
    /* Remove obj from container */
    obj->retire_to_cohort(cohort_);
    ex_.submit([] {asynchronous_reclamation(); });
  }
};
class B {
  // Deleter may depend on resources
  // with independent lifetime.
  ~B() { use_resource_XYZ(); }
};
make_resource_XYZ();
{ Container<B> container;
  container.insert(b);
  container.erase(b); }
// Obj containing 'b' must be already deleted.
destroy_resource_XYZ();
```

**OK**

# Batch Creation and Destruction

# Hazard Pointer Batch Creation and Destruction

## C++26 (one at a time)

```
{ hazard_pointer hp[3];
  /* Three hazard pointers are made nonempty
     separately. */
  hp[0] = make_hazard_pointer();
  hp[1] = make_hazard_pointer();
  hp[2] = make_hazard_pointer();
  assert(!hp[0].empty());
  assert(!hp[1].empty());
  assert(!hp[2].empty());
  /* src is atomic<T*> */
  T* ptr = hp[0].protect(src);
  /* etc */
} /* Three nonempty hazard pointers are
     destroyed separately. */
```

**e.g., ~6 ns**

## Batch creation and destruction

```
{ hazard_pointer hp[3];
  /* Three hazard pointers are made nonempty
     together. */
  make_hazard_pointer_batch(std::span{hp});
  SCOPE_EXIT {
    destroy_hazard_pointer_batch(std::span{hp}); };
  assert(!hp[0].empty());
  assert(!hp[1].empty());
  assert(!hp[2].empty());
  /* src is atomic<T*> */
  T* ptr = hp[0].protect(src);
  /* etc */
} /* Three nonempty hazard pointers are emptied
     together, and then destroyed separately. */
```

**e.g., ~2 ns**

## Possible API

```
void make_hazard_pointer_batch(std::span<hazard_pointer>);
void destroy_hazard_pointer_batch(std::span<hazard_pointer>) noexcept;
```

# Pointer Tagging

# Pointer Tagging

P3125R0: Pointer Tagging (author: Hana Dusíková) (`wg21.link/p3125r0`)

Pointer to aligned object `T` with **`alignof(T)` = 4**:

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 0 | 0 |

**Motivation (P3125R0):**

> *"Pointer tagging is not allowed in standard C++ as manipulating pointer bits is UB. Because of this limitation, some advanced data structures are not implementable or sub-optimally implementable.*
> *This technique is an existing practice, is widely used in the field, and standardising it would lower the bar for its safe usage among C++'s users."*

# P3125R0: Pointer Tagging API

```
template <typename T, size_t Alignment = alignof(T)> class tagged_pointer;
```

*available bits for tagging*

```
template <typename T, size_t Alignment = alignof(T)>
constexpr auto tag_bit_mask() noexcept -> uintptr_t;
```

*pointer --> tagged_pointer*

```
template <typename T, size_t Alignment = alignof(T)>
constexpr auto tag_pointer(T* original, uintptr_t tag) noexcept
-> tagged_pointer<T, Alignment>;
```

Precondition: `tag == (tag & tag_bit_mask<T, Alignment>)`

*tagged_pointer --> pointer*

```
template <typename T, size_t Alignment = alignof(T)>
constexpr auto untag_pointer(tagged_pointer<T, Alignment> ptr) noexcept -> T*;
```

*tagged_pointer --> tag*

```
template <typename T, size_t Alignment = alignof(T)>
constexpr auto tag_value(tagged_pointer<T, Alignment> ptr) noexcept -> uintptr_t;
```

# Pointer Tagging Example

```
using TaggedPointer = tagged_pointer<T,2>;
```

```
bool try_tag_untagged_pointer(atomic<TaggedPointer>& src) {
  TaggedPointer current = src.load();
  assert(tag_value(current) == 0);              // Assert src was untagged
  assert(1 & tag_bit_mask<T,2>() == 1);         // Assert that 1 is a valid tag
  T* ptr = untag_pointer(current);              // Extract pointer
  TaggedPointer newval = tag_pointer(ptr, 1);   // Tag the pointer
  return src.compare_exchange_weak(current, newval);  // CAS src
}
```

# Extending Hazard Pointers to Tagged Pointers

## C++26

```
template <typename T> T* protect(const atomic<T*>& src) noexcept;
template <typename T> bool try_protect(T*& ptr, const atomic<T*>& src) noexcept;
```

## Possible Extensions

```
template <typename T, size_t Alignment = alignof(T)>
tagged_pointer<T, Alignment> protect(
    const atomic<tagged_pointer<T, Alignment>>& src) noexcept;
```

```
template <typename T, size_t Alignment = alignof(T)>
bool try_protect(
    tagged_pointer<T, Alignment>& ptr,
    const atomic<tagged_pointer<T, Alignment>>& src) noexcept;
```

## Example

```
atomic<tagged_pointer<T>> src_;
```

```
hazard_pointer hp = make_hazard_pointer();
tagged_pointer<T> tagged = hp.protect(src_);
/* Safe to use ptr, where ptr == untag_pointer(tagged). */
```

# Bringing parallelism to std::ranges algorithms

# ISO C++ Parallelism/Concurrency Programming Language (based on Gonzalo's ISC C++ BoF)

**C++ 2017**

**Parallel Algorithms [many]**

**Concurrency++**
Memory Model++ [MichaelW, Maged, Paul M]
Forward Progress

**C++ 2020**

**Concepts**

**Ranges**

**Modules**

**Concurrency++ [Bryce, Gonzalo]**
**Coroutines**
atomic_ref, barriers, …

**C++ 2023**

**Ranges++** many

**Multi-dimensional Spans [Bryce, Christian]**
**operator[i, j, k]**

**2024**

**C++ 2026**

**Executors/Sender/Receiver [P2300]**

**Reflection**

**Algorithms++** asynchrony, ranges **[Ruslan, Alexey, Bryce]**

**Linear Algebra, submdspan, padded layouts [Bryce, Christian]**

**Concurrency++** RCU, HP **[MichaelW, Gonzalo, Maged, Paul M]**

**SIMD** [Matthias, Daniel, Ruslan]

# Bringing parallelism to std::ranges algorithms (P3179)

```
// C++03
template<class RandomAccessIterator, class Compare>
 void sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

// C++17
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
 void sort(ExecutionPolicy&& exec, RandomAccessIterator first, RandomAccessIterator last, Compare comp);

// C++20
template<random_access_range R, class Comp = ranges::less, class Proj = identity>
requires sortable<iterator_t<R>, Comp, Proj>
 constexpr borrowed_iterator_t<R> ranges::sort(R&& r, Comp comp = {}, Proj proj = {});

// C++26? (https://wg21.link/P3179)
template<class ExecutionPolicy, random_access_range R, class Comp = ranges::less, class Proj = identity>
requires sortable<iterator_t<R>, Comp, Proj>
 constexpr borrowed_iterator_t<R> ranges::sort(ExecutionPolicy&& exec, R&& r, Comp comp = {}, Proj proj = {});
```

# C++ Parallel Range Algorithms: Unifying Parallelism and Ranges

**Why Combine Parallelism with Ranges?**

- Ranges offer a productive API with opportunities for optimization.
- Users are already using ranges with non-range parallel algorithms; integrating execution policies simplifies and streamlines code.

**The Power of Ranges and Parallelism**

- The C++ Ranges library provides a powerful way to express and compose computations lazily.
- C++17 introduced parallel algorithms, but they don't integrate seamlessly with ranges.
- This paper proposes adding parallel algorithms that work directly with ranges, combining the benefits of both worlds.

**The Need for Parallel Range Algorithms**

- Users often combine ranges and parallel algorithms, but the current approach is verbose and error-prone.
- The proposed parallel range algorithms offer a more natural and expressive way to parallelize range-based computations

.

# Design Overview: Key Modifications

- Execution policy parameter added to range algorithms.
- Introduction of bounded ranges for better parallel performance.
- 
- **Execution policies:** Parallel range algorithms accept execution policies to control parallelism.
- **Random access ranges:** Algorithms require random-access ranges for efficient parallelization.
- **Bounded ranges:** At least one input and the output range must be bounded for safety and performance.
- **Algorithm return types:** Consistent with serial range algorithms for easy migration.
- 
- Enable single-call fusion of multiple operations
- Preserve the expressiveness of ranges

# Key Design Decisions

1. Return types match serial range algorithms
2. Require random_access_range (for now)
3. Take range as output
4. Require bounded ranges
5. Preserve callable requirements from C++17 parallel algorithms

```
template <class ExecutionPolicy,
random_access_range R,
          class Proj = identity,

indirectly_unary_invocable<projected<i
terator_t<R>, Proj>> Fun>

requires
sized_sentinel_for<ranges::sentinel_t<
R>, ranges::iterator_t<R>>

ranges::borrowed_iterator_t<R>

ranges::for_each(ExecutionPolicy&&
policy, R&& r, Fun f, Proj proj = {});
```

Differences to C++17 Parallel Algorithms

## Key Differences:

- Parallel range algorithms require random access ranges.
- Output can now be a range instead of just an iterator.

# Benefits

Parallel range algorithms offer a natural and efficient way to parallelize range-based computations.

The proposed design integrates seamlessly with the Ranges library and existing parallel algorithms.

This feature will enhance the expressiveness and performance of parallel code in C++.

More expressive code

Potential for better performance

Safer APIs (bounded ranges, range outputs)

Simplified migration from serial to parallel code

# C++ Parallel Algorithms

# P2500R2 C++ parallel algorithms and P2300

**Overview:**

- The evolution of parallelism in C++.
- Why P2300 is important for C++26.

**Goal of the Talk:**

- Discuss the integration of C++ parallel algorithms with the facilities introduced in P2300.
- https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2500r2.htm

# ISO C++ Parallelism/Concurrency Programming Language (based on Gonzalo’s ISC C++ BoF)

**Parallel Algorithms** [many]

**Concurrency++**
Memory Model++ [MichaelW, Maged, Paul M]
Forward Progress

**C++ 2017**

**C++ 2020**

**Concepts**

**Ranges**

**Modules**

**Concurrency++** [Bryce, Gonzalo]
**Coroutines**
atomic_ref, barriers, …

**Ranges++** many

**Multi-dimensional Spans** [Bryce, Christian]
**operator[i, j, k]**

**C++ 2023**

**2024**

**C++ 2026**

**Executors/Sender/Receiver [P2300]**

**Reflection**

**Algorithms++** asynchrony, ranges [Ruslan, Alexey, Bryce]

**Linear Algebra, submdspan, padded layouts** [Bryce, Christian]

**Concurrency++** RCU, HP [MichaelW, Gonzalo, Maged, Paul M]

**SIMD** [Matthias, Daniel, Ruslan]

# Further evolution of parallel algorithms (P2500)

```
//C++ 11/14
namespace stde = std::execution;
std::vector<int> vec{N};

// https://wg21.link/P3179: parallel range algorithms
std::ranges::generate( vec, std::minstd_rand{} );
std::ranges::sort( stde::par, vec );

// https://wg21.link/P2500: support schedulers to specify where to execute,  targeted to cover synchronous parallel algorithms integration with P2300.
std::ranges::generate(stde::execute_on(some_sched, stde::seq), vec, std::minstd_rand{} );
std::ranges::sort( stde::execute_on(some_sched, stde::par), vec );
```

# Motivation

**Why P2300?**

- The need for a flexible abstraction that answers "where" the code should be executed.
- Limitations of current execution policies in specifying hardware execution contexts.

C++17 parallel algorithms: A good start

Current limitation: No control over execution hardware

P2300 introduces flexible schedulers

Need: Integrate schedulers with parallel algorithms

The Need for Integration

- C++ parallel algorithms offer parallelism, but lack control over execution hardware ("where").
- P2300 introduces the "scheduler" concept, representing execution contexts, addressing the "where."
- The integration of these two is crucial for leveraging hardware capabilities effectively.

# Design Overview

The Proposed Solution in P2500

- The paper proposes extending C++ algorithms to accept a "policy-aware scheduler."
- This scheduler combines an execution policy ("how") and a scheduler ("where").
- The `execute_on` function facilitates the creation of such policy-aware schedulers.

**Design Goals:**

- Extending C++ parallel algorithms with policy-aware schedulers.
- Allowing customization for different execution contexts.
- 
- Preserve core semantics of algorithms and policies
- Cover both "classic" and range-based algorithms
- 
- Minimal API changes: The design aims to preserve the existing usage patterns of C++ algorithms.
- Flexibility: It allows execution semantics to be adjusted based on the capabilities of the execution context.
- Customization: Implementers of execution contexts can customize the implementation of standard algorithms for optimal performance.
- 

**Key Features:**

- Combining scheduler and policy.
- Minimal, incremental API changes.

# Combining Scheduler with Policy

**Why Use Schedulers?**

- Schedulers represent execution contexts and provide flexibility.
- API overview

Key Concepts

1. policy_aware_scheduler
2. execute_on
3. Customizable functions

- 
    - The `execution_policy` concept defines the requirements for execution policies.
    - The `policy_aware_scheduler` concept represents an entity combining a scheduler and an execution policy.
    - The `execute_on` customization point binds a scheduler and an execution policy.
    - Parallel algorithms are defined as customizable functions, allowing customization for specific policy-aware schedulers.

# policy_aware_scheduler Concept

## API

```
template <typename S>
concept policy_aware_scheduler =
scheduler<S> && requires (S s) {
   typename
S::base_scheduler_type;
   typename S::policy_type;
   { s.get_policy() } ->
execution_policy;
 };
```

## Usage Example

```
struct MyScheduler {

    using base_scheduler_type = /* some scheduler
type */;

    using policy_type = /* some execution policy
type */;

    policy_type get_policy() const {

        return /* return the associated policy
*/;

    }};

 static_assert(policy_aware_scheduler<MySchedul
 er>);
```

# execute_on Function API and Usage example

```cpp
inline namespace
__execute_on_fn_namespace
{
  inline constexpr
__detail::__execute_on_fn
execute_on;
 }
```

```cpp
auto
policy_aware_sched =
std::execute_on(my_sc
heduler,
std::execution::par);
```

# Proposed API (Example with for_each)

// Existing API

```
template<class ExecutionPolicy, class It, class Fun>
      constexpr void for_each(ExecutionPolicy&& policy, It first, It last, Fun f);
```

// New Policy-based API

```
template<execution_policy Policy, input_iterator I, sentinel_for<I> S, class Proj = identity,
         indirectly_unary_invocable<projected<I, Proj>> Fun>
constexpr ranges::for_each_result<I, Fun>
  ranges::for_each(Policy&& policy, I first, S last, Fun f, Proj proj = {});
```

// New Scheduler-based API

```
template<policy_aware_scheduler Scheduler, input_iterator I, sentinel_for<I> S,
         class Proj = identity, indirectly_unary_invocable<projected<I, Proj>> Fun>
constexpr ranges::for_each_result<I, Fun>
  ranges::for_each(Scheduler sched, I first, S last, Fun f, Proj proj = {}) /*customizable*/;
```

Allowing schedulers with C++ algorithms.

Blocking behavior similar to C++17 parallel algorithms.

```
template<policy_aware_scheduler Scheduler,
typename ForwardIterator, typename Function>
void for_each(Scheduler&& sched, ForwardIterator
first, ForwardIterator last, Function f) {
    // Implementation using scheduler and policy
    sched.execute([&]() {
        for (; first != last; ++first) {
            f(*first);
        }
    });
}
```

# Using the API

```
std::for_each(

std::execute_on(my_gpu_scheduler,
std::execution::par),
      begin(data),
      end(data),
      [](auto& item) { item.process();
}
 );
```

# Customization and Extensibility

**Parallel Algorithms as Customizable Functions:**

- Customization through policy-aware schedulers.
- Flexibility to support platform-specific optimizations.

**Example Customization:**

- Using CUDA-specific scheduler for `std::for_each`.

```
namespace cuda {
  struct scheduler {
    friend constexpr auto
    tag_invoke(std::tag_t<ranges::for_each>, scheduler, /*...*/) {
      // CUDA-optimized implementation
      cuda_kernel<<<blocks, threads>>>(/*...*/);
      return std::ranges::for_each_result{/*...*/};
    }
  };
 }
```

# Summary

P2300 offers significant flexibility and control over execution contexts.

P2500 Integration with parallel algorithms is crucial for modern C++ development.

The future of C++ parallelism lies in customizable and extensible algorithms.

# Conclusion and Questions

We covered a few interesting proposals that are coming in C++ parallelism and concurrency based on the last 3 C++ Std meetings since the last CPPCON:

1. Improving C++20 Atomic Min/Max(P0493, Michael)
2. Hazard pointer extensions (P3135; Maged)
3. Pointer tagging (P3125; Maged)
4. Parallel Range algorithms (P3179; Michael), maybe Parallel Algorithms (P2500),

1 and 4 are likely to make C++26. 2 and 3 will be beyond C++ 26.
These are just some of the likely features that have progressed sufficiently and fit in a talk format, though they still may enter the standard in slightly different forms.

If you like this talk, we will continue to cover more parallelism and concurrency proposals in future CPPCON talks to prepare you for the future, giving our take on their significance, and usefulness in parallel concurrent programming.