vollmann engineering gmbh

Concurrent Queues API

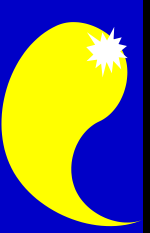
P0260R15 / P3537R1

WG21

Hagenberg, February 2025

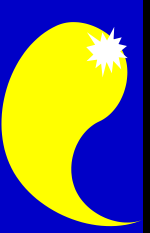
Detlef Vollmann

vollmann engineering gmbh



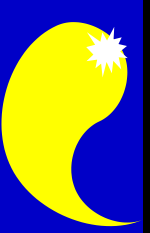
Overview

- Presentation
  - Queues
  - Example
  - Error Handling
  - Concurrent Concept
  - Async Concept
- Discussion
  - `expected`
  - Obsolete Error Facilities
  - Async Interface
  - Implementability
- SG1



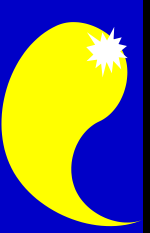
Concurrent Queues are not Containers

- Concurrent queues are concurrent data structures
- A communication mechanism
- A synchronization mechanism
  - consumers wait for producers
  - producers wait for consumers
- (Temporary) storage is a possible implementation detail
  - queues of size 0 sometimes make perfect sense



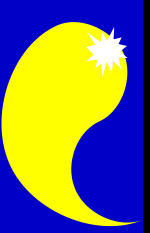
Design Space

- The design space for concurrent queues is pretty big
  - partly in interface design
  - more in semantics
- Single or multiple connections on producer and/or consumer side
- Lock-free vs. locking
  - separate for both ends
- Memory allocation
  - up-front, per push/pop, external
- Ordering guarantee
  - FIFO vs. priorities
- Non-blocking only vs. synchronous interface
- Single push/pop vs. two-phase
- Strongly typed vs. (dynamically sized) byte chunks



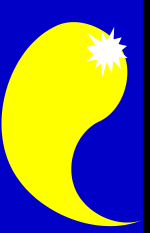
Design Space

- More interface
  - timed waits
  - asynchronous
  - debugging
  - single ended interfaces
- Efficiency vs. robust/portable interface
- Error handling (exceptions)
- Concurrency vs. parallelism vs. asynchronicity



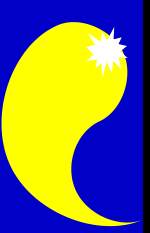
Concepts for Concurrent Queues

- No single queue implementation can cover all design aspects
- Provided concepts are expected to cover most design aspects
- Implementing both async and non-blocking interfaces has performance costs
  - and real challenges
- Concept is split into one base concept and two separate concepts for async and non-blocking
- Many different implementations for these concepts are expected
  - some of them may be standardized
- Possible single-ended adapter can use these concepts
- `bounded_queue` models all concepts



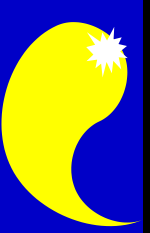
Basic Concept

```
enum class conqueue_errc { success, empty, full, closed, busy, busy_async };

void close() noexcept;
bool is_closed() const noexcept;

bool push(const T& x);
bool push(T&& x);
template <typename... Args> bool emplace(Args &&... as);

optional<T> pop();
```

Closing Queues

- The only queues that don't need `close` are
  - queues that are never closed
  - single producer, single consumer with inline close token
- For all other cases synchronization needs access to queue internals
  - as detailed in the paper
- So the basic concept provides `close`

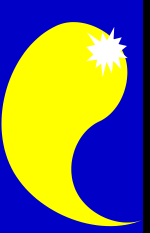


Synchronous `push`

- Push interface

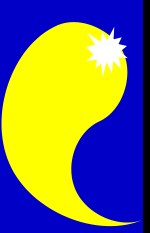
```
bool push(const T& x);
bool push(T&& x);
template <typename... Args> bool emplace(Args &&... as);
```

- Returns `true` on success and `false` on close



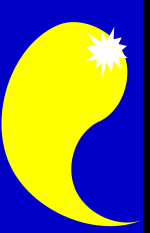
Synchronous `pop`

- Pop interface
  optional<T> pop();
- Returns `optional` with value on success and empty `optional` on close
- This is what LEWG voted for in Wroclaw



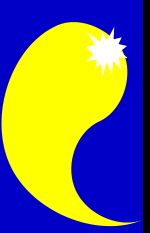
Example

- "Find files with string"
- One task/thread collects all the file paths in a directory and pushes them into a queue and then closes the queue
- Other tasks/threads (one or more) pop file paths from the queue and search them for a string
- Synchronous version with multiple threads
- Single-threaded Asynchronous version with coroutines
- Single-threaded Asynchronous version with native S/R
- Code available at `https://gitlab.com/cppzs/bounded-queue/-/tree/master/demo`



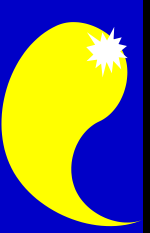
"Error" Handling

- "One person's exception is another person's expected result."
- The current proposal is to have no queue based errors.
- LEWG decided in Wroclaw to have `optional<T> pop()`
  - i.e. `closed` is not an error
- This leads to `bool push(T&& x)`
- For non-blocking functions (`try_*`) `empty` and `full` (and arguably `busy` and `busy_async`) are similar



Concurrent Queue Concept

```
conqueue_errc try_push(const T& x);
conqueue_errc try_push(T&& x);
template <typename... Args> conqueue_errc try_emplace(Args &&... as);

optional<T> try_pop(conqueue_errc &ec);
```

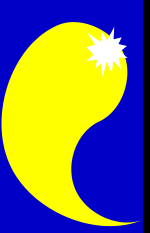



Non-Blocking `push`

- Push interface

```
conqueue_errc try_push(const T& x);
conqueue_errc try_push(T&& x);
template <typename... Args> conqueue_errc try_emplace(Args &&... as);
```

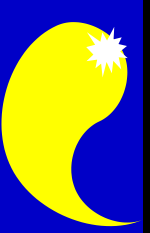
- This is the logical extension to blocking `push`



Non-Blocking `pop`

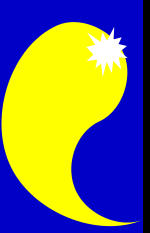
- Pop interface

  optional<T> try_pop(conqueue_errc &ec);



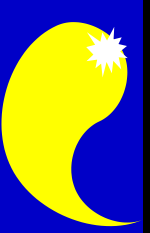
Logging Example

- Embedded system
- No blocking anywhere
- Debug messages are raised anywhere
  - pushed into queue
- Background task takes messages from the queue and sends them to a UART
  - no blocking either
  - `try_pop`



Async Queue Concept

```
sender auto async_push(const T&);
sender auto async_push(T&&);
template <typename... Args> sender auto async_emplace(Args &&... as);

sender auto async_pop();
```

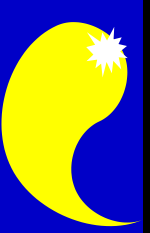
Asynchronous Interface

```
sender auto async_pop();
```

- Current proposal for `async_pop` calls `set_value(T)` on success and `set_error(conqueue_errc)` when closed.

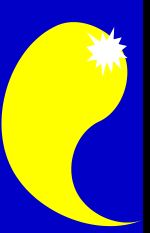
```
sender auto async_push(const T&);
sender auto async_push(T&&);
template <typename... Args> sender auto async_emplace(Args &&... as);
```

- Analogously `async_push` calls `set_value(void)` on success and `set_error(conqueue_errc)` when closed.



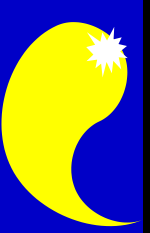
Example Using Coroutines

- "Find files with string"
- Single-threaded Asynchronous version with coroutines



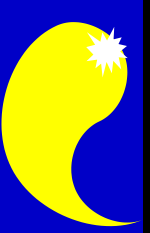
Example Sender/Receiver

- "Find files with string"
- Single-threaded Asynchronous version with native S/R



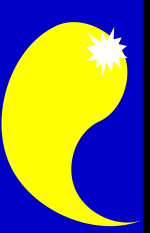
Discussion

Discussion



Non-Blocking pop

- Pop interface
  optional<T> try_pop(conqueue_errc &ec);
- Alternative versions would be
  expected<T, conqueue_errc> queue::try_pop();
- or even
  expected<optional<T>, conqueue_errc> queue::try_pop();



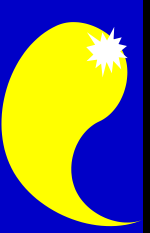
Non-Blocking `pop`

- Example from P2921R0:

```
conqueue_errc ec;
while (auto val = q.try_pop(ec))
   println("got {}", *val);
if (ec == conqueue_errc::closed)
  return;
// do something else.
```

- With `expected<T, conqueue_errc>`

```
auto val = q.try_pop();
while (val) {
  println("got {}", *val);
  val = q.try_pop();
}
if (val.error() == conqueue_errc::closed)
  return;
// do something else
```

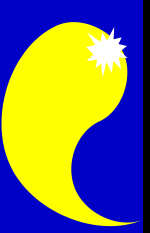



Non-Blocking `pop`

- With `expected<optional<T>, conqueue_errc>`

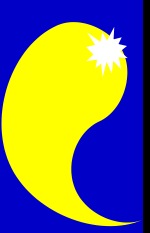
```
auto val = q.try_pop();
while (val && *val) {
  println("got {}", **val);
  val = q.try_pop();
}
if (val.error() == conqueue_errc::closed)
  return;
// do something else
```

- LEWG poll in St. Louis: "LEWG would like to add a `std::expected` interface for concurrent queues":
  `|SF|F|N|A|SA|  |0|2|5|3|2"`



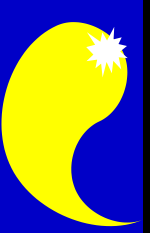
Obsolete Error Facilities

- Now `conqueue_error` and `conqueue_category` are not needed anymore and `conqueue_errc` should possibly renamed (was `queue_op_status` before R5).



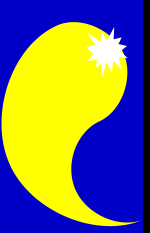
Async "Error" Handling

- In many cases calling push or pop operations on a closed queue is common and you either expect a value or a "closed" signal.
- For async operations calling `set_error` for closed queues feels intuitively wrong.
- Considering the closed signal as special value delivered through the `set_value` channel seems plausible.
- But if `async_pop` doesn't produce a value, calling `set_value` seems wrong either.
  - it clobbers the value channel
- The current proposal proposes to call `set_error(conqueue_errc)`
  - even if I still don't consider it an error
- For symmetry, `async_push` uses `set_error` as well



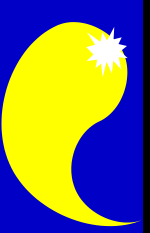
Async "Error" Handling

- LEWG voted strongly in favour in Wroclaw for the sender to call `set_value(optional<T>)`
- Sender/receiver are used via coroutines or native
- For coroutines, `set_value(optional<T>)` is probaly the perfect choice
- For native sender/receiver separating value and error channels is probably a much better choice
- Different interfaces for coroutines and native are akward
  - but wait for P3570
- With `set_value`/ `set_error` coroutines get an exception on closed queues
  - or use something like `error_as_optional`
- `async_pop` calls `set_value(T)` on success and `set_error()` when closed.
- `async_push` could call `set_value(bool)`



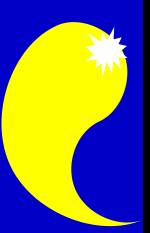
Continuation Scheduling

- Concurrent queues are used to separate different execution contexts or different execution agents
- `pop` continues on the execution agent it was called
- `async_pop` continues on the scheduler it was called
- Same for `async_push`
- A push operation never runs the the "continuation" of the pop
  - and vice versa



Implementability

- Consider the logging example
  - `try_push` from everywhere
  - using `async_pop` from the background task
- This should work
  - It doesn't
- `async_push` might need to schedule the continuation
  - this requires to enqueue the continuation to the execution context
  - this requires (possibly blocking) synchronization
- This isn't an implementation issue.
  - if you control the queue and execution context implementation
- The continuation might be on a user provided scheduler
- No existing facilities to co-ordinate with execution context
- SG1 decided to return `conqueue_errc::busy_async` in this case



SG1 Decision in Hagenberg

- Poll: in review of P0260R14:
  - In the sequential consistency specification, `pop1` and `pop2` should not be related by `strongly happens-before`, but merely be related "in that order".
  - We will do more work between meetings on whether "that order" is per queue, or it is the unique sequentially-consistent order itself.
  - It is correct for `set_value`/`set_error` to be called on the scheduler of `r`. Concurrent data structures aren't merely data structures, they are also control constructs.
  - Introduce `busy_async` and return that in the two specific cases when try functions would have process an async operation (`try_push`/`try_pop` → `async_pop`/`try_push`, respectively)
  - We still want this in C++26, because it is an important vocabulary type to use with S&R
- No objections to unanimous consent

