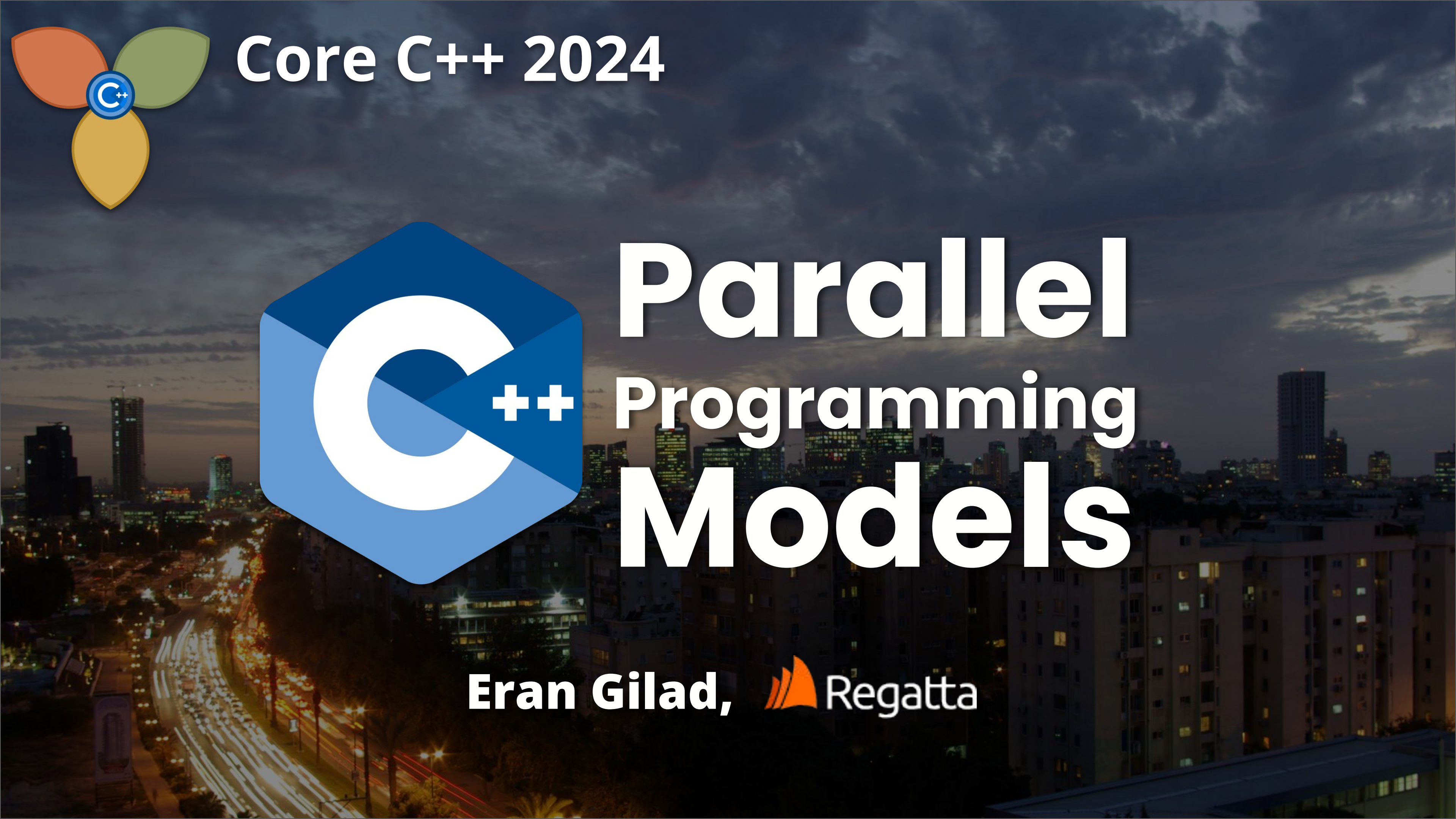


Core C++ 2024

Parallel Programming Models

Eran Gilad, Regatta

Nice to meet you!

Day job:

Database internals @

Regatta

Community activity:

Conference @ Core C++ 2024

Meetups @ haifa::c++

Chit-chat @ מדברים C++

bit.ly/cpp-wa-desc

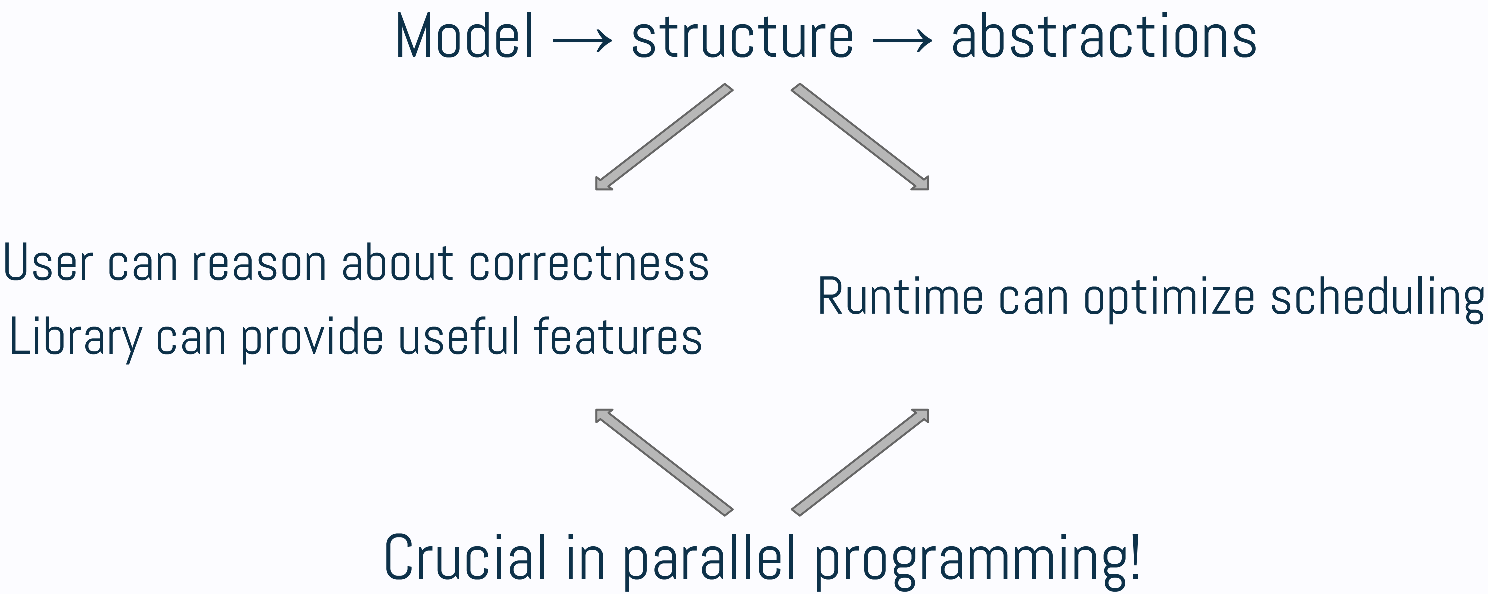
Why parallel programming?

1. Performance
2. Hide I/O
3. Responsiveness

Why is it hard?
non-deterministic execution, data races, deadlocks

Having *structure* makes things easier

Programming model



Model ⟶ structure ⟶ abstractions

User can reason about correctness

Library can provide useful features

Runtime can optimize scheduling

Crucial in parallel programming!

C++ parallel programming facilities

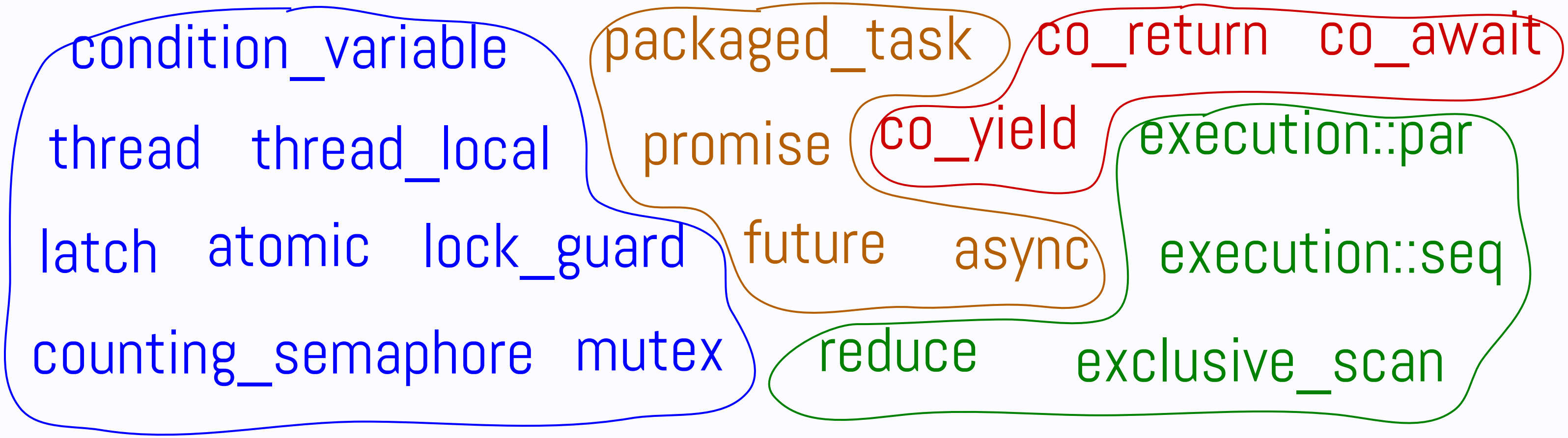
condition_variable packaged_task co_return co_await

thread thread_local promise co_yield execution::par

latch atomic lock_guard future async execution::seq

counting_semaphore mutex reduce exclusive_scan

C++ parallel programming facilities



condition_variable
thread thread_local
latch atomic lock_guard
counting_semaphore mutex

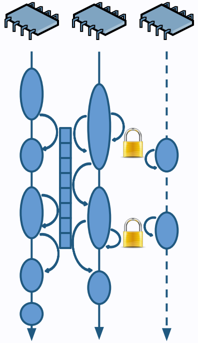
packaged_task
promise
future async

co_return co_await
co_yield

execution::par
execution::seq
reduce exclusive_scan

Not random facilities, not a single API:
4 parallel programming models

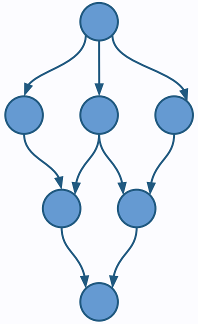
C++ parallel programming models



Unstructured

thread, mutex, atomics

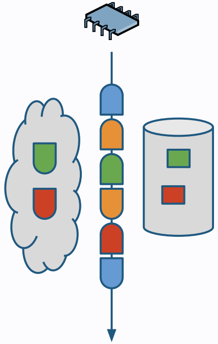
(C++11)



Task parallelism

async, promises, futures

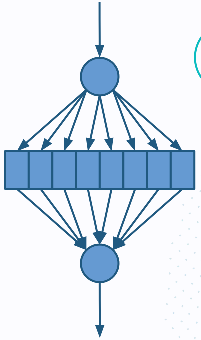
(C++11)



Cooperative tasks

coroutines

(C++20)



Data parallelism

parallel algorithms

(C++17)

Not different *abstraction levels* - different program structuring

Talk outline

Intro

Unstructured parallelism

Task-based parallelism

Cooperative multitasking

Data parallelism

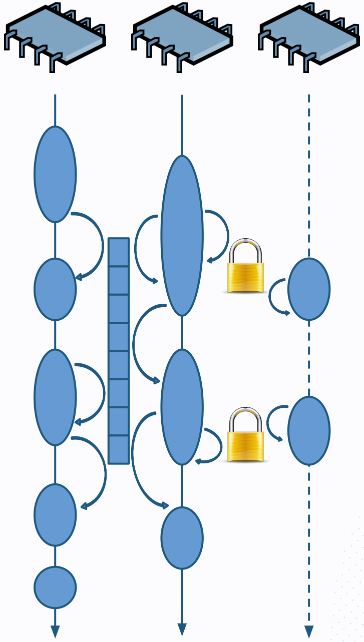
Mixing models

This talk is about the forest,

not about the trees

Model #1: Unstructured parallelism

- Ad hoc use of parallelization facilities
    - Solve a particular need
- Relies on lower-level abstractions
- Min overhead, max hardware utilization
- Min safety, so requires max proficiency
- Central C++11 feature



Major components (partial list)

C++11/14

C++20

Convenience utils

RAII lock wrappers, call_once, jthread

Threading and sync classes

thread, mutex, condition_variable, semaphore, latch, barrier

Memory model

atomic and friends, thread_local

Exclusive use cases

- Construct higher level facilities
  - E.g., thread pool, spin lock etc.
- Concurrent data structures
  - At least thread safe, usually better if lock free
- Long running services

Missing part - safe shared state

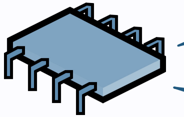
- Threads communicate via shared state
    - atomics are too fine-grained
    - locking complete containers is too coarse grained
- We need *concurrent data structures!*
    - Nothing yet
    - RCU and hazard pointers hopefully in C++26

Unstructured parallelism pros and cons

- Maximal control
- Usually maximal performance (when done right)
- Complicated memory model, hard to make ideal use
- Data races, deadlocks, non-determinism

Thread-level API shortcomings

Thread ≈



Get execution entry point,
no functional semantics

Synchronization, communication,
scheduling interleaved in code

Higher abstraction level + clever runtime

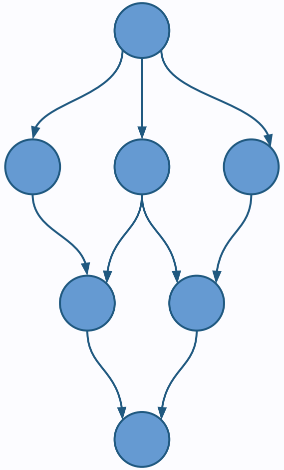
⇓

Less work, less bugs, probably better performance

Model #2: Task-based parallelism

- Task: limited computation providing single result
  - Function, lambda, loop iteration etc.
  - *Ideally* has inputs and/or output, no side effects

- Decouples functionality from execution
  - One thread can run many tasks
  - One task can migrate among many threads (?)



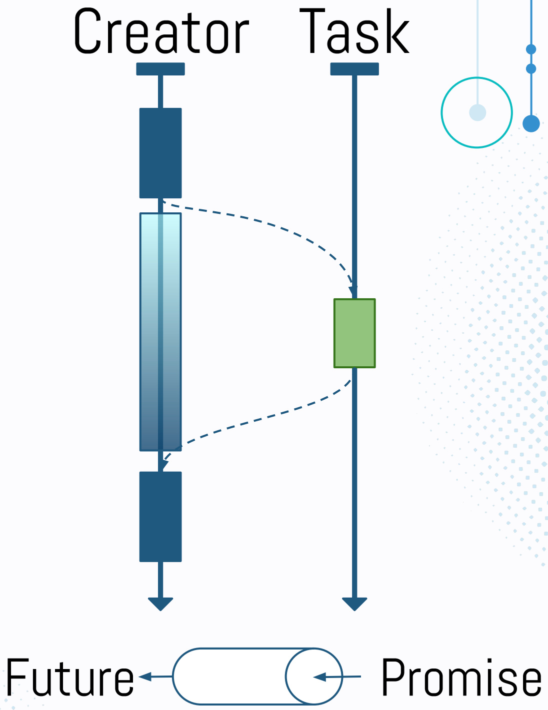
Tasks are asynchronous

Execution:

- Somewhere, sometime
- Creator proceeds till outcome needed

Communication channel:

- *Future*: A handle to the task outcome
- *Promise*: the task's end of the future



C++ tasks

```
future<int> answer = async(..., []{ return 42; });
pretend_to_work();
cout << “found the answer: “ << answer.get() << endl;
```

- Spawn tasks using std::async
- Obtain results using std::future
- The runtime assigns tasks to worker threads
    - Yes, the C++ runtime can create and manage threads without user intervention!

future and promise

- future<T>
  - Returned by a call to async
  - Provides access to the task's result
  - The "pull" end of a communication channel
- promise<T>
  - The "push" end of the communication channel
  - Encapsulated by async

async execution

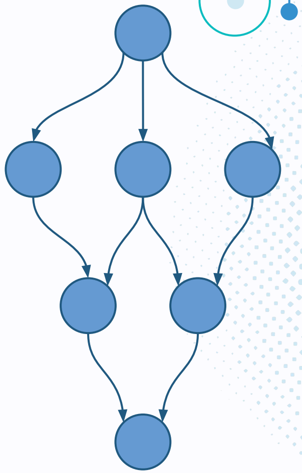
```
future<int> answer = async(when, []{ return 42; });
pretend_to_work();
cout << “found the answer: “ << answer.get() << endl;
```

Task execution determined by ***`when`***:

1. **`launch::async`** - on a new thread (“as if”!)
2. **`launch::deferred`** - right before the results are needed
3. **`async | deferred`** - implementation dependent (default)

std::async tasks limitations

- No composition - e.g., then()
- No execution config - e.g., thread pool
- No scheduling options - e.g., yield
- No changes since C++11 - 🤷



Model #3: Cooperative Multitasking

- Blocked tasks should not block the CPU
    - Assuming #tasks >> #cores
- Tasks are a user-mode concept
    - The kernel can't help
- Async flows based on callbacks are cumbersome
    - Not the *programming model* we want

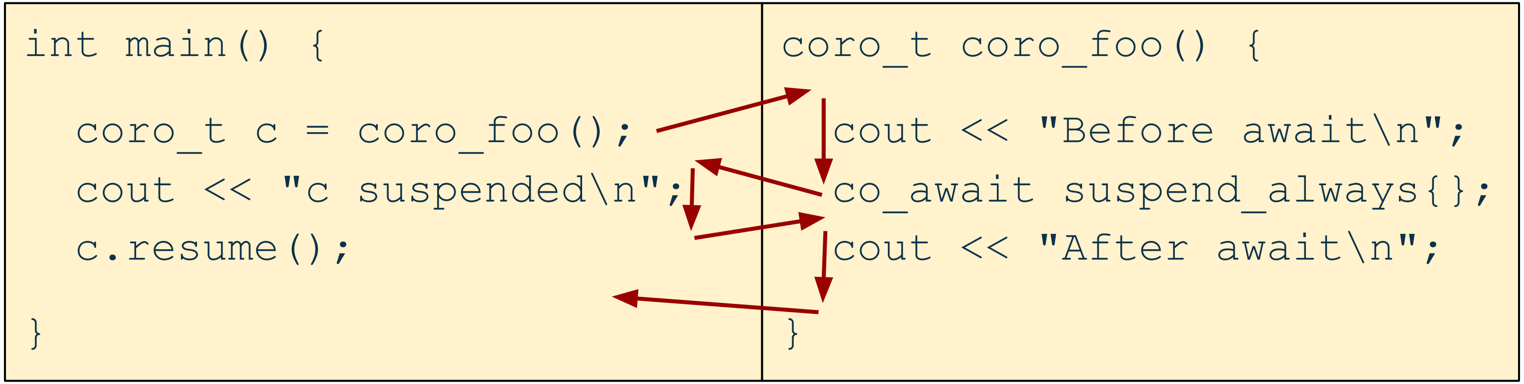
Enter coroutines!

Coroutines

C++20 coroutine: a function that can be explicitly suspended and resumed by the C++ code

- No operating system involvement
- No assembly
- Full language (and compiler) support
- Well defined state management and error handling

Example



```
int main() {

  coro_t c = coro_foo();
  cout << "c suspended\n";
  c.resume();

}
```

```
coro_t coro_foo() {

  cout << "Before await\n";
  co_await suspend_always{};
  cout << "After await\n";

}
```

coro_t: define coroutine behavior and resume handle

co_await *expr*

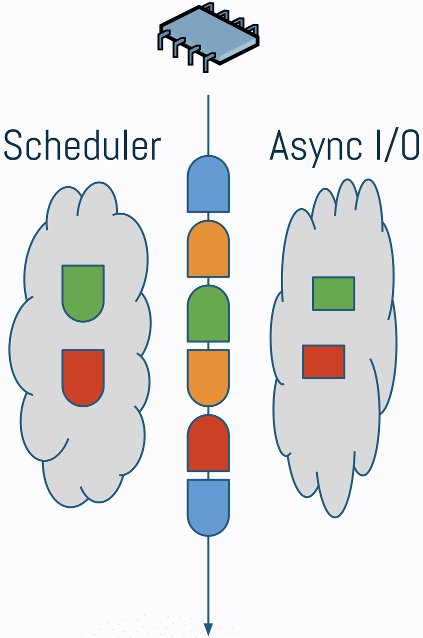
- *expr* should evaluate to an *awaitable object*
- The awaitable is a scheduling point:
  - await_ready - should the coroutine be suspended
  - await_suspend - get the suspended coroutine handle
  - await_resume - after coroutine resumed
- co_await foo():
  - execute foo, evaluate its return value (awaitable)
  - suspend the current coroutine if needed

Coroutine scheduling

- co_await returns control to caller or another coroutine
  - It *does not* create any parallelism!
- The program must:
  - Track suspended coroutines
  - Resume ready coroutines
- Requirements:
  - Some source of parallelism (async I/O, worker threads, etc.)
  - Scheduler

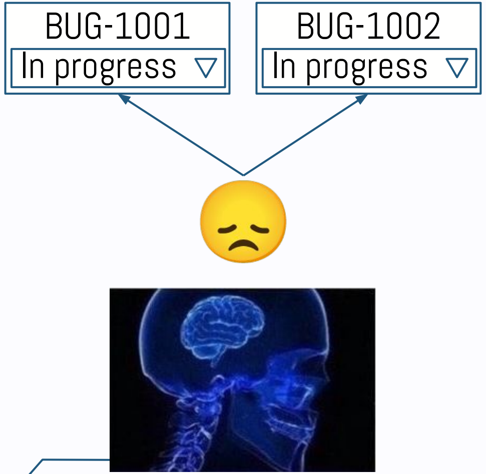
Coroutine-based multitasking example

1. The scheduler starts a coroutine task
2. The task invokes async I/O
3. The task **suspends itself**
4. The scheduler switches to another task
5. The async I/O completes
6. The awaiting task marked as ready
7. The scheduler **resumes it** when possible



Concurrency and parallelism

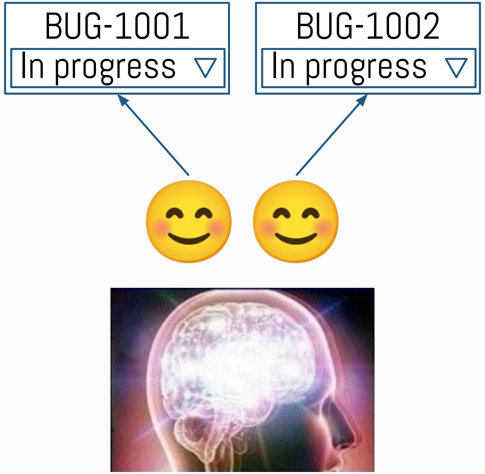
Task concurrency,
no parallelism



BUG-1001
In progress ▽

BUG-1002
In progress ▽

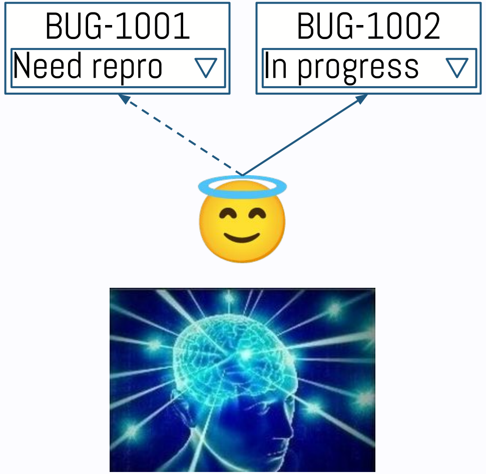
Task concurrency,
multi-CPU parallelism



BUG-1001
In progress ▽

BUG-1002
In progress ▽

Task concurrency,
CPU/IO parallelism



BUG-1001
Need repro ▽

BUG-1002
In progress ▽

Parallel coroutines execution

- Commonly one thread per core
- Suspended tasks can move among cores
  - Can you use thread_local in this scenario?
- Sync primitives must be coroutine-aware
  - co_await coro_lock() instead of std::mutex.lock()
  - A coroutine can't switch if its thread is blocked!
- Expected scheduler complexity - races, load balancing, task affinity, etc.

Threads vs. tasks

Threads use cases	Tasks use cases
Parallelism abstraction (CPU)	Concurrency abstraction (function)
Background services/workers	Many short and independent ops
Long complex parallel operations	“local”/ad-hoc (async I/O)
Equal amount of per-thread work	Dynamic load balancing

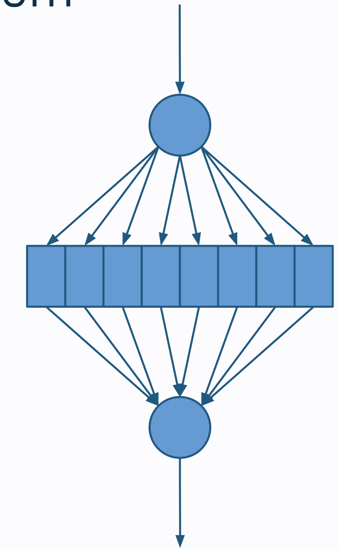
Fibers / user threads / green threads

- Stackful coroutines
  - A full stack must be allocated for each fiber
- Any function running on a fiber can be suspended
  - Not just the coroutine body
- Decoupled executor (fiber) and work (function)
  - Allows fiber pools etc.
- Requires an "ecosystem" - scheduler and sync objects
- Not supported in C++ (yet?)

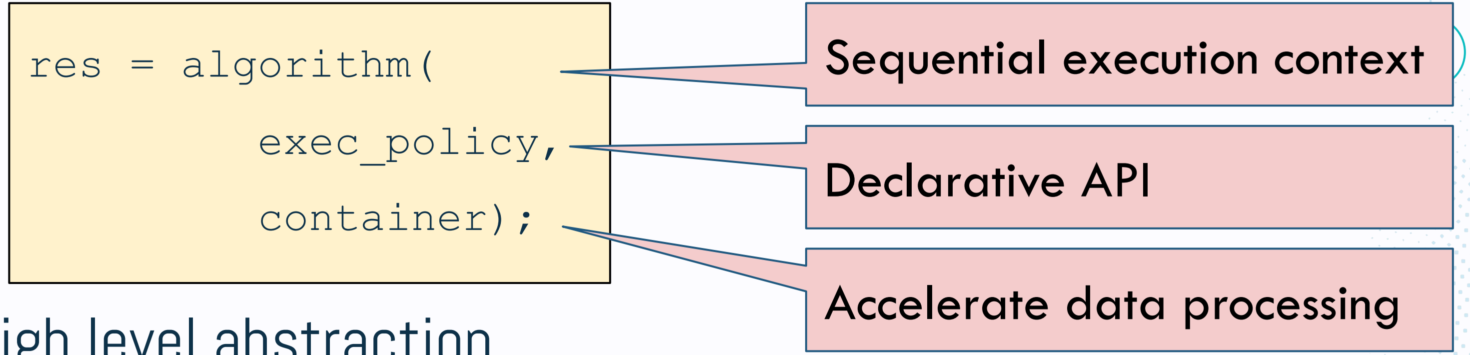
Model #4: Data parallelism

A programming model in which parallelism stems from the individual computation associated with every element in a collection.

- Requires a lot of non-dependent data
- Procedure:
  - Divide element processing among processors
  - Short and simple operation on each element



C++ Parallel algorithms



```
res = algorithm(
          exec_policy,
          container);
```

Sequential execution context

Declarative API

Accelerate data processing

- High level abstraction
  - Few customization points for user
  - No control over parallelism, scheduling, work distribution etc.
  - Library + runtime can be very efficient
  - When used properly: no data races, deadlocks etc.

Using parallel algorithms

```
vector<int> v = {1, 2, 3, 5, 11, 20};
int res = reduce(execution::par, v.begin(), v.end());
assert(res == 42);
```

- Most standard algorithms have a parallel overload
  - First parameter: *ExecutionPolicy*
  - *ForwardIterator* instead of InputIterator / OutputIterator
- Complexity requirements more lax
- Implementation isn't specified – in theory, can use GPUs

sequenced_policy

- Forces execution to take place on the calling thread
- Differs from no-policy call:
  1. exceptions invoke std::terminate()
  2. ForwardIterator
- execution::seq is an *instance of* sequence_policy
  - Algorithms can be overloaded by policy
  - Policy is a *compile time* decision!

```
reduce(execution::seq, ..., ...);
```

parallel_policy

- Execution on caller or another thread (runtime pool)
- Per thread, semantics are similar to sequenced_policy - unspecified order, no interleaving
- Data races are now possible if multiple operations write to unprotected data

```
reduce(execution::par, ..., ...);
```

parallel_unsequenced_policy

- Operations can now be interleaved and moved from thread to thread during execution
  - Operations must not use any locks
  - Cannot assume a thread executes a single operation at a time
- More user restrictions => more library options
  - Vectorization can now be used
  - finer grained scheduling

```
reduce(execution::par_unseq, ..., ...);
```

unsequenced_policy

- Operations can be interleaved on a single thread
  - Not a multithreaded context
  - But vectorization can still be used!

- C++20 addition

```
reduce(execution::unseq, ..., ...);
```

Non-standard policies

- Vendor-specific
- Can allow the use of accelerators
    - GPU
    - FPGA
    - ASIC

Parallel algorithms != parallel containers

- C++ separates algorithms from containers
    - Thread-safe containers by default? No - zero-overhead!
- Parallel algorithms can modify data but not structure
    - Unlike sequential algorithms, which can modify both
- Accessing a container processed by a parallel algorithm from another thread? Possible race!

Mixing models - unstructured context

Most cores are used:

- E.g., server
- Use coroutines for async I/O
- No resources for parallel algorithms or async compute

Most cores are unused:

- E.g., background service
- Tasks and parallel algorithms can be used
  - Impact? depends
- Careful with shared state!

Reminder: runtime schedulers aren't aware of user threads

Mixing models - std::async tasks context

Most cores are used:

- Namely, many async tasks created
- No point in creating threads or using parallel algorithms
- Awkward context for coroutine execution

Most cores are unused:

- E.g., ad-hoc work, UI worker
- Can use parallel algorithms from tasks
- Creating threads less suitable - spawn tasks instead
- Using sync mechanism (mutex etc.) doesn't fit the model

Mixing models - coroutine tasks context

Most cores are used:

- Namely, the scheduler uses many cores
- No point in using other models from within coroutines

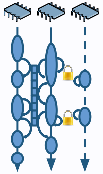
Most cores are unused:

- E.g., limited I/O needs
- Parallel algorithms from coroutine will block it
- std::async tasks somewhat match (~allow awaiting)
- Tasks too short for thread creation

Mixing models - parallel algorithms

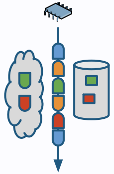
- Within a parallel algorithm (e.g., user lambda):
    - No point in spawning async tasks or creating threads
    - Inappropriate context for coroutines
    - Using sync mechanisms to access external state possible but might kill concurrency

Summary



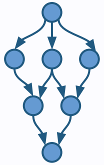
Unstructured

Low-level building blocks



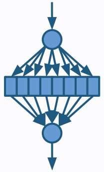
Cooperative multitasking

great for async I/O,
missing scheduler



Task parallelism

functional decomposition,
incomplete feature



Data parallelism

declarative, no executors
control (yet!)

Mixing models hardly works. Parallel programming is hard.