concore

Lucian Radu Teodorescu

Aug 15, 2021

CONTENTS

concore is a C++ library that aims to raise the abstraction level when designing concurrent programs. It allows the user to build complex concurrent programs without the need of (blocking) synchronization primitives. Instead, it allows the user to "describe" the existing concurrency, pushing the planning and execution at the library level.

We strongly believe that the user should focus on describing the concurrency, not fighting synchronization problems.

The library also aims at building highly efficient applications, by trying to maximize the throughput.

concore is built around the concept of tasks. A task is an independent unit of work. Tasks can then be executed by so-called *executors*. With these two main concepts, users can construct complex concurrent applications that are safe and efficient.

concore concurrency core

> variation on *concord* – agreement or harmony between people *threads* or groups (of threads); a chord that is pleasing or satisfactory in itself.

CHAPTER

ONE

TABLE OF CONTENT

1.1 Quick-start

1.1.1 Building the library

The following tools are needed:

- conan
- CMake

Perform the following actions:

```
mkdir -p build
pushd build

conan install .. --build=missing -s build_type=Release

cmake -G<gen> -D CMAKE_BUILD_TYPE=Release -D concore.testing=ON ..
cmake --build .

popd build
```

Here, <gen> can be `Ninja`, `make`, `XCode`, `"Visual Studio 15 Win64"`, etc.

1.1.2 Tutorial

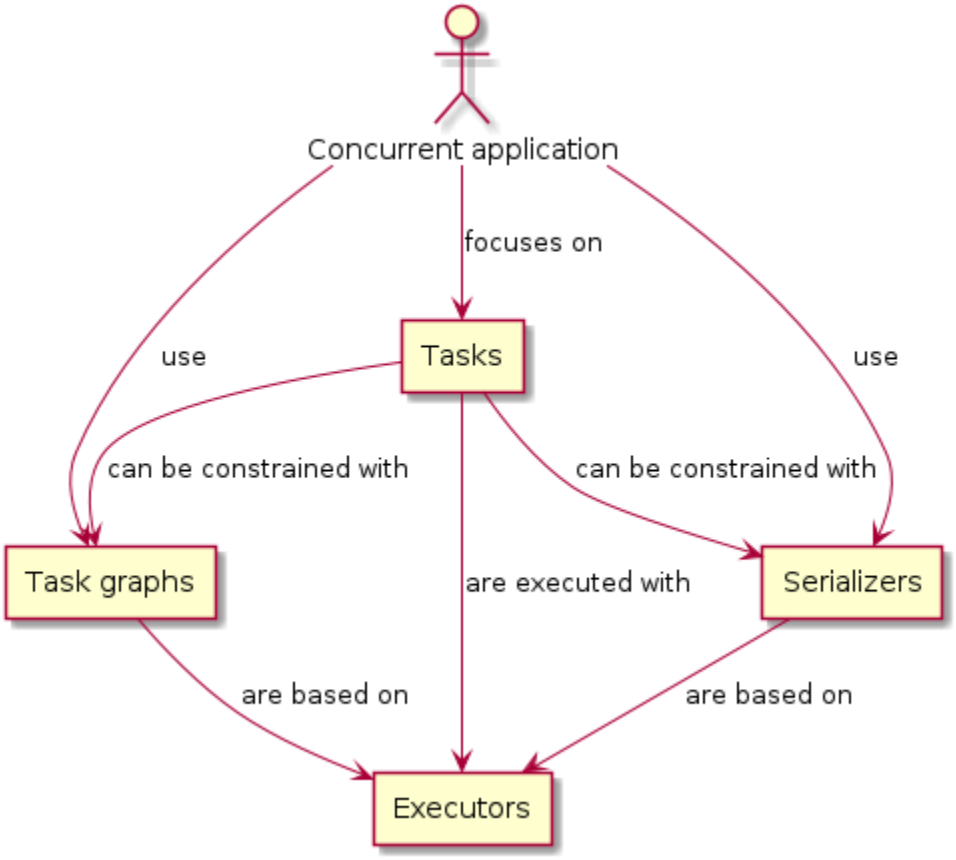
TODO

1.1.3 Examples

TODO

1.2 Concepts

Overview of main concepts:



1.2.1 Building concurrent applications with concore

Traditionally, applications are using manually specified threads and manual synchronization to support concurrency. With many occasions this method has been proven to have a set of limitations:

- performance is suboptimal due to synchronization
- understandability is compromised
- thread-safety is often a major issue
- composability is not achieved

concore aims at alleviating these issues by implementing a *Task-Oriented Design* model for expressing concurrent applications. Instead of focusing on manual creation of threads and solving synchronization issues, the user should focus on decomposing the application into smaller units of work that can be executed in parallel. If the decomposition is done correctly, the synchronization problems will disappear. Also, assuming there is enough work, the performance of the application can be close-to-optimal (considering throughput). Understandability is also improved as the concurrency is directly visible at the design level.

The main focus of this model is on the design. The users should focus on the design of the concurrent application, and leave the threading concerns to the concore library. This way, building good concurrent applications becomes a far easier job.

Proper design should have two main aspects in mind:

1. the total work needs to be divided into manageable units of work
2. proper constraints need to be placed between these units of work

concore have tools to help with both of these aspects.

For breaking down the total work, there are the following rules of thumb:

- at any time there should be enough unit of works that can be executed; if one has *N* cores on the target system the application should have 2*N units of works ready for execution
- too many units of execution can make the application spend too much time in bookkeeping; i.e., don't create thousands or millions of units of work upfront.
- if the units of work are too small, the overhead of the library can have a higher impact on performance
- if the units of work are too large, the scheduling may be suboptimal
- in practice, a good rule of thumb is to keep as much as possible the tasks between 10ms to 1 second – but this depends a lot on the type of application being built

For placing the constraints, the user should plan what types of work units can be executed in parallel to what other work units. concore then provides several features to help managing the constraints.

If these are followed, fast, safe and clean concurrent applications can be built with relatively low effort.

1.2.2 Tasks

Instead of using the generic term *work*, concore prefers to use the term *task* defined the following way:

task An independent unit of work

The definition of *task* adds emphasis on two aspects of the work: to be a *unit* of work, and to be *independent*.

We use the term *unit of work* instead of *work* to denote an appropriate division of the entire work. As the above rules of thumb stated, the work should not be too small and should not be too big. It should be at the right size, such as dividing it any further will not bring any benefits. Also, the size of a task can be influenced by the relations that it needs to have with other tasks in the application.

The *independent* aspect of the tasks refers to the context of the execution of the work, and the relations with other tasks. Given two tasks *A* and *B*, there can be no constraints between the two tasks, or there can be some kind of execution constraints (e.g., "*A* needs to be executed before *B*", "*A* needs to be executed after *B*", "*A* cannot be executed in parallel with *B*", etc.). If there are no explicit constraints for a task, or if the existing constraints are satisfied at execution time, then the execution of the task should be safe, and not produce undefined behavior. That is, an *independent* unit of work should not depend on anything else but the constraints that are recognized at design time.

Please note that the *independence* of tasks is heavily dependent on design choices, and maybe less on the internals of the work contained in the tasks.

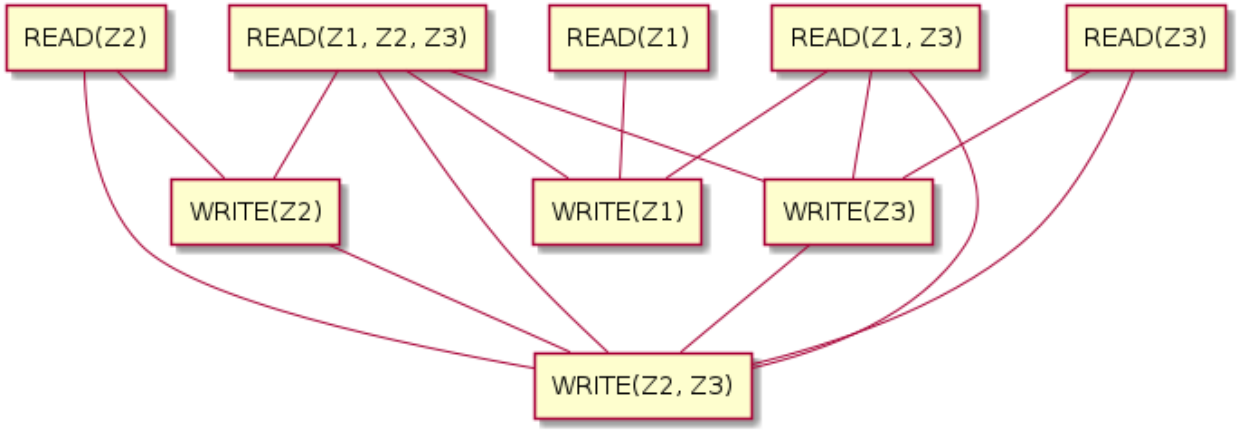
Let us take an example. Let's assume that we have an application with a central data storage. This central data storage has 3 zones with information that can be read or written: *Z1*, *Z2* and *Z3*. One can have tasks that read data, tasks that write data, and tasks that both read and write data. These operations can be specific to a zone or multiple zones in the central data storage. We want to create a task for each of these operations. Then, at design time, we want to impose the following constraints:

- No two tasks that write in the same zone of the data storage can be executed in parallel.
- A task that writes in a zone cannot be executed in parallel with a task that reads from the same zone.
- A task that reads from a zone can be executed in parallel with another task that reads from the same zone (if other rules don't prevent it)

- Tasks in any other combination can be safely executed in parallel

These rules mean that we can execute the following four tasks in parallel: *READ(Z1)*, *READ(Z1, Z3)*, *WRITE(Z2)*, *READ(Z3)*. On the other hand, task *READ(Z1, Z3)* cannot be executed in parallel with *WRITE(Z3)*.

Graphically, we can represent these constraints with lines in a graph that looks like the following:



One can check by looking at the figure what are all the constraints between these tasks.

In general, just like we did with the example above, one can define the constraints in two ways: synthetically (by rules) or by enumerating all the legal/illegal combinations.

In code, concore models the tasks by using the `concore::v1::task` class. They can be constructed using arbitrary work, given in the form of a `std::function<void()>`.

1.2.3 Executors

Creating tasks is just declaring the work that needs to be done. There needs to be a way of executing the tasks. In concore, this is done through the *executors*.

executor An abstraction that takes a task and schedules its execution, typically at a later time, and maybe with certain constraints.

Concore has defined the following executors:

- `global_executor`
- `spawn_executor`
- `spawn_continuation_executor`
- `inline_executor`
- `delegating_executor`
- `dispatch_executor`
- `tbb_executor`
- `any_executor`

Executors execute tasks. Thus, for a given object *t* of type `concore::v1::task`, and an executor *ex*, one can call: `concore::execute(ex, t)`. This will ensure that the task will be executed in the context of the executor.

An executor can always be stored into a `any_executor`, which is a polymorphic executor that can hold another executor.

For most of the cases, using a `global_executor` is the right choice. This will add the task to a global queue from which concore's worker threads will extract and execute tasks.

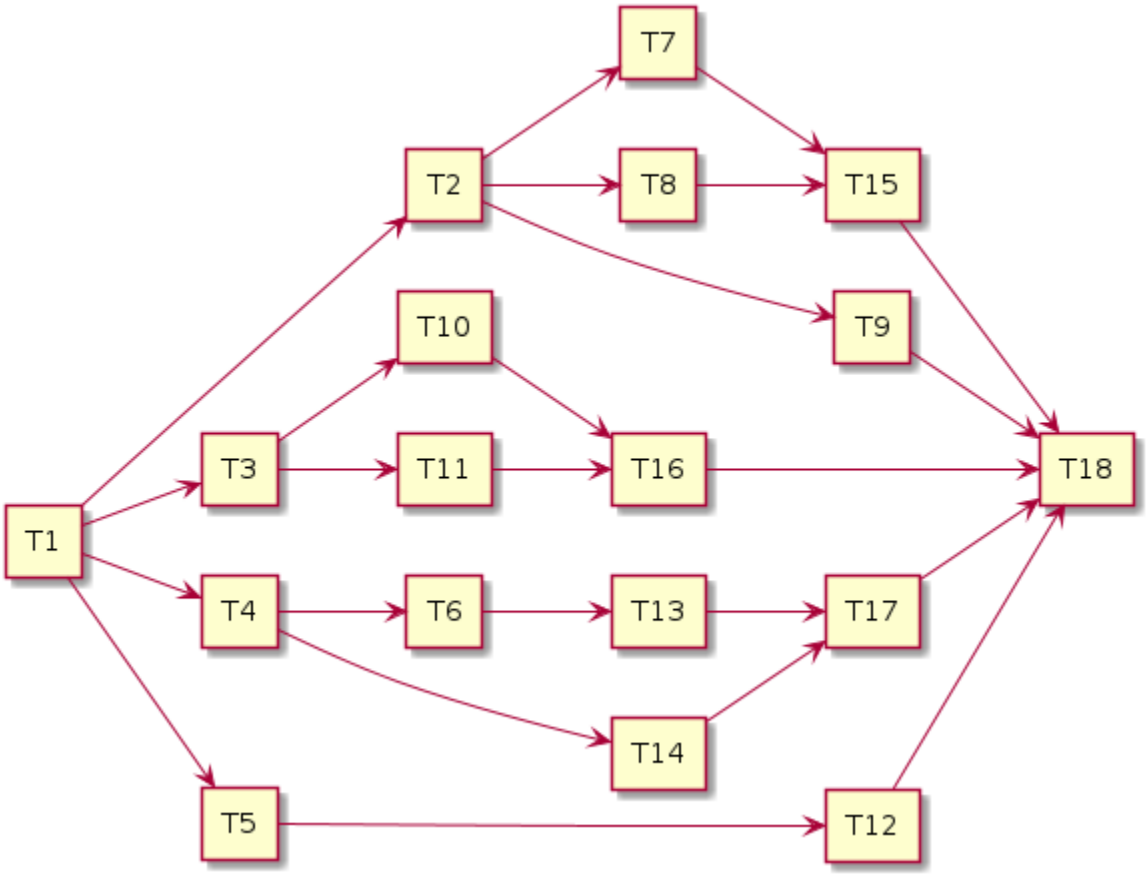
Another popular alternative is to use the *spawn* functionality (either as a free function `spawn()`, or through `spawn_executor`). This should be called from within the execution of a task and will add the given task to the local queue of the current worker thread; the thread will try to pick up the last task with priority. If using `global_executor` favors fairness, `spawn()` favors locality.

Using tasks and executors will allow users to build concurrent programs without worrying about threads and synchronization. But, they would still have to manage constraints and dependencies between the tasks manually. concore offers some features to ease this job.

1.2.4 Task graphs

Without properly applying constraints between tasks the application will have thread-safety issues. One needs to properly set up the constraints before enqueueing tasks to be executed. One simple way of adding constraints is to add dependencies; that is, to say that certain tasks need to be executed before other tasks. If we chose the encode the application with dependencies the application becomes a directed acyclic graph. For all types of applications, this organization of tasks is possible and it's safe.

Here is an example of how a graph of tasks can look like:



Two tasks that don't have a path between them can be executed in parallel.

This graph, as well as any other graph, can be built manually while executing it. One strategy for building the graph is the following:

- tasks that don't have any predecessors or for which all predecessors are completely executed can be enqueued for execution
- tasks that have predecessors that are not run should not be scheduled for execution
- with each completion of a task, other tasks may become candidates for execution: enqueue them
- as the graph is acyclic, in the end, all the tasks in the graph will be executed

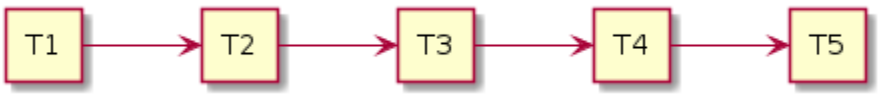
Another way of building task graph is to use concore's abstractions. The nodes in the graph can be modeled with `chained_task` objects. Dependencies can be calling `add_dependency()` of `add_dependencies()` functions.

1.2.5 Serializers

Another way of constructing sound concurrent applications is to apply certain execution patterns for areas in the application that can lead to race conditions. This is analogous to adding mutexes, read-write mutexes, and semaphores in traditional multi-threaded applications.

In the world of tasks, the analogous of a mutex would be a `serializer`. This behaves like an executor. One can enqueue tasks into it, and they would be *serialized*, i.e., executed one at a time.

For example, it can turn 5 arbitrary tasks that are enqueued roughly at the same time into something for which the execution looks like:



A serializer will have a waiting list, in which it keeps the tasks that are enqueued while there are tasks that are in execution. As soon as a serializer task finishes a new task is picked up.

Similar to a `serializer`, is an `n_serializer`. This corresponds to a semaphore. Instead of allowing only one task to be executing at a given time, this allows *N* tasks to be executed at a given time, but not more.

Finally, corresponding to a read-write mutex, concore offers `rw_serializer`. This is not an executor, but a pair of two executors: one for *READ* tasks and one for *WRITE* tasks. The main idea is that the tasks are scheduled such as the following constraints are satisfied:

- no two *WRITE* tasks can be executed at the same time
- a *WRITE* task and a *READ* tasks cannot be executed at the same time
- multiple *READ* tasks can be executed at the same time, in the absence of *WRITE* tasks

As working with mutexes, read-write mutexes and semaphores in the traditional multi-threaded applications are covering most of the synchronization cases, the `serializer`, `rw_serializer` and `n_serializer` concepts should also cover a large variety of constraints between the tasks.

1.2.6 Others

Manually creating constraints

One doesn't need concore features like task graphs or serializers to add constraints between the tasks. They can easily be added on top of the existing tasks by some logic at the end of each task.

First, a constraint is something that acts to prevent some tasks to be executed while other tasks are executed. So, most of our logic is added to prevent tasks from executing.

To simplify things, we assume that a task starts executing immediately after it is enqueued; in practice, this does not always happen. Although one can say that implementing constraints based on this assumption is suboptimal, in practice

it's not that bad. The assumption is not true when the system is busy; then, the difference between enqueue time and execution time is not something that will degrade the throughput.

Finally, at any point in time we can divide the tasks to be executed in an application in three categories:

1. tasks that can be executed right away; i.e., tasks without constraints
2. tasks that can be executed right away, but they do have constraints with other tasks in this category; i.e., two tasks that want to WRITE at the same memory location, any of them can be executed, but not both of them
3. tasks for which the constraints prevent them to be executed at the current moment; i.e., tasks that depend on other tasks that are not yet finished executing

At any given point, the application can enqueue tasks from the first category, and some tasks from the second category. Enqueueing tasks from the second category must be done atomically with the check of the constraint; also, other tasks that are related to the constraint must be prevented to be enqueued, as part of the same atomic operation.

While the tasks are running without any tasks completing, or starting, the constraints do not change – we set the constraints between tasks, not between parts of tasks. That is, there is no interest for us to do anything while the system is in steady-state executing tasks. Whenever a new task is created, or whenever we complete a task we need to consider if we can start executing a task, and which one can we execute. At those points, we should evaluate the state of the system to see which tasks belong to the first two categories. Having the tasks placed in these 3 categories we know which tasks can start executing right away – and we can enqueue these in the system.

If the constraints of the tasks are properly set up, i.e., we don't have circular dependencies, then we are guaranteed to make progress eventually. If we have enough tasks from the first two categories, then we can make progress at each step.

Based on the above description it can be assumed that one needs to evaluate all tasks in the system at every step. That would obviously not be efficient. But, in practice, this can easily be avoided. Tasks don't necessarily need to sit in one big pool and be evaluated each time. They are typically stored in smaller data structures, corresponding to different parts of the application. And, furthermore, most of the time is not needed to check all the tasks in a pool to know which one can be started. In practice evaluating which tasks can be executed can be done really fast. See the serializers above.

Task groups

Task groups can be used to control the execution of tasks, in a very primitive way. When creating a task, the user can specify a `concore::v1::task_group` object, making the task belong to the task group object passed in.

Task groups are useful for canceling tasks. One can tell the task group to cancel, and all the tasks from the task group are canceled. Tasks that haven't started executed yet will not be executed. Tasks that are in progress can query the cancel flag of the group and decide to finish early.

This is very useful in *shutdown* scenarios when one wants to cancel all the tasks that access an object that needs to be destroyed. One can place all the tasks that operate on that object in a task group and cancel the task group before destroying the object.

Another important feature of task groups is the ability to wait on all the tasks in the group to complete. This is also required to the *shutdown* scenario above. concore does not block the thread while waiting; instead, it tries to execute tasks while waiting. The hope is to help in getting the tasks from the arena done faster.

Note that, while waiting on a group, tasks outside of the group can be executed. That can also mean that waiting takes more time than it needs to. The overall goal of maximizing throughput is still maintained.

Details on the task system

This section briefly describes the most important implementation details of the task system. Understanding these implementation details can help in creating more efficient applications.

If the processor on which the application is run has *N* cores, then concore creates *N* worker threads. Each of these worker threads has a local list of tasks to be executed can execute one task at a given time. That is, the library can only execute a maximum of *N* tasks at the same time. Increasing the number of tasks in parallel will typically not increase the performance, but on the contrary, it can decrease it.

Besides the local list of tasks for each worker, there is a global queue of tasks. Whenever a task is enqueued with `global_executor` it will reach in this queue. Tasks in this queue will be executed by any of the workers. They are extracted by the workers in the order in which they are enqueued – this maintains fairness for task execution.

If, from inside a worker thread one calls `spawn()` with a task, that task will be added to the local list of tasks corresponding to the current worker thread. This list of tasks behaves like a stack: last-in-first-out. This way, the local task lists aim to improve locality, as it's assumed that the latest tasks added are *closer* to the current tasks.

A worker thread favors tasks from the local task list to tasks from the global queue. Whenever the local list runs out of tasks, the worker thread tries to get tasks from the central queue. If there are no tasks to get from the central queue, the worker will try to steal tasks from other worker thread's local list. If that fails too, the thread goes to sleep.

When stealing tasks from another worker, the worker is chosen in a round-robin fashion. Also, the first task in the local list is extracted, that is, the furthest task from the currently executing task in that worker. This is also done like that to improve locality.

So far we mentioned that there is only one global queue. There are in fact multiple global queues, one for each priority. Tasks with higher priorities are extracted before the tasks with lower priority, regardless of the order in which they came in.

All the operations related to task extraction are designed to be fast. The library does not traverse all the tasks when choosing the next task to be executed.

1.2.7 Extra concore features

Pipeline

Concore provides an easy way to build pipelines that can implement different transformations on a data stream. One can create an instance of the `pipeline` class, set up stages, set up the maximum allowed parallelism and let elements flow through the pipeline.

Finish events

Sometimes the user is interested in the finalization of a specific task or set of tasks. One can use `finish_event` to notify when the task/chain of tasks is finished. In conjunction with that, `finish_task` can be used to start a task whenever the finish event was notified, or `finish_wait` to wait for that finalization condition.

Algorithms

Concore provides a couple concurrent algorithms that of general use:

- `conc_for()`
- `conc_reduce()`
- `conc_scan()`
- `conc_sort()`

A concurrent application typically means much more than transforming some STL algorithms to concurrent algorithms. Moreover, the performance of the concurrent algorithms and the performance gain in multi-threaded environments depends a lot on the type of data they operate on. Therefore, concore doesn't focus on providing an exhaustive list of concurrent algorithms like Parallel STL.

Probably the most useful of the algorithms is `conc_for()`, which is highly useful for expressing embarrassing parallel problems.

C++23 executors

Concore provides an implementation of the executors proposal that targets C++23. It's not a complete implementation, but it covers the most important parts:

- concepts
- customization point objects
- thread pool
- type wrappers

Here are a list of things that are not yet supported:

- properties and requirements – they seem too complicated to be actually needed
- extra conditions for customization point object behavior; i.e., a scheduler does not automatically become a sender – the design for this is too messy, with too many circular dependencies

1.3 C++23 executors

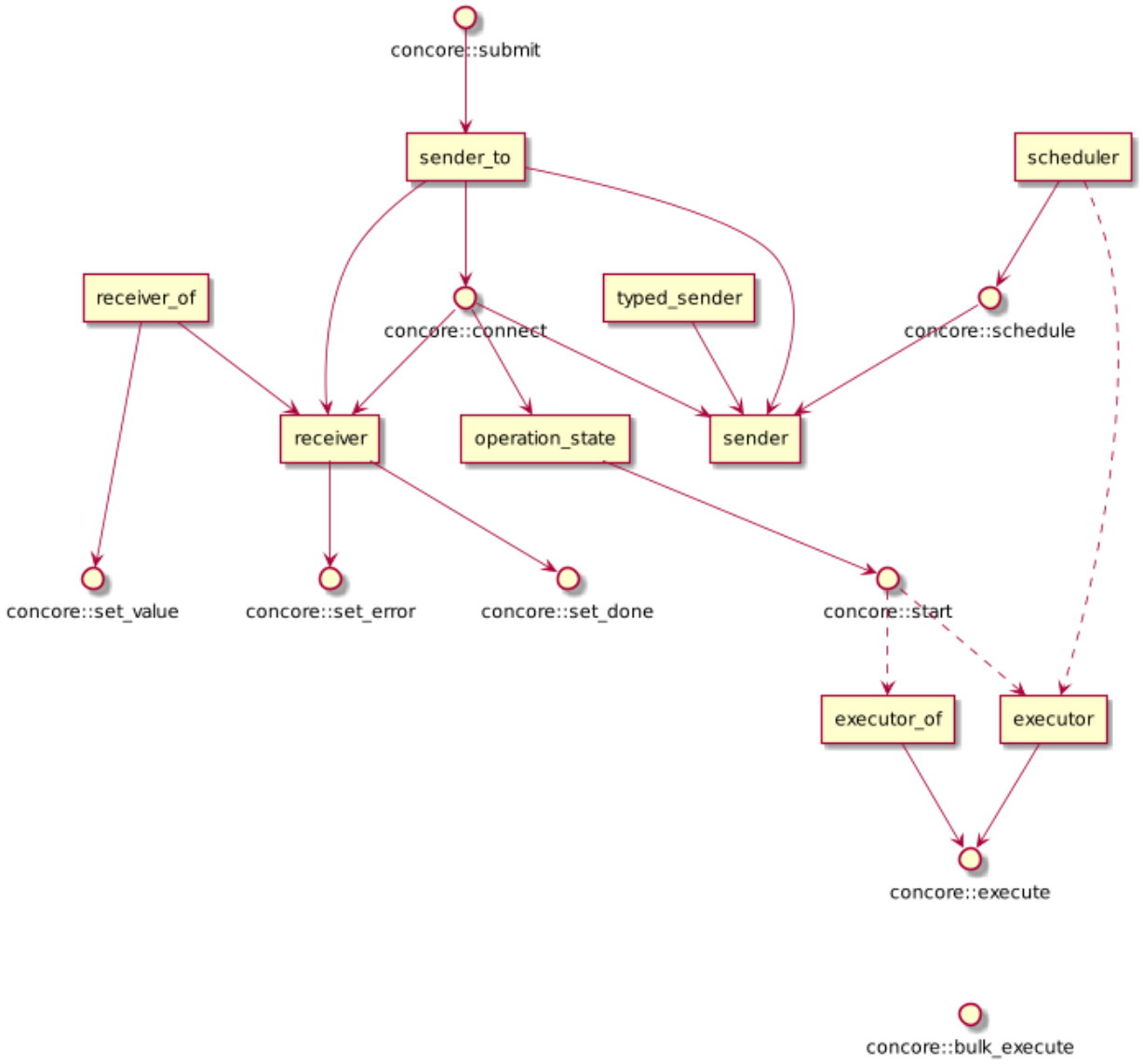
concore implements C++23 executors, as defined by P0443: A Unified Executors Proposal for C++. It is not a complete implementation, but the main features are present. Concore's implementation includes:

- concepts
- customization point objects
- thread pool
- type wrappers

However, it does not include:

- properties and requirements – they seem too complicated to be actually needed
- extra conditions for customization point object behavior; i.e., a scheduler does not automatically become a sender – the design for this is too messy, with too many circular dependencies

1.3.1 Concepts and customization-point objects



The following table lists the customization-point objects (CPOs) defined:

CPO	Description
`void set_value(R&&, Vs&&...)`	Given a receiver `R`, signals that the sender operation has completed (with zero or more values)
`void set_done(R&&)`	Given a receiver `R`, signals that the sender operation was canceled
`void set_error(R&&, E&&)`	Given a receiver `R`, signals that the sender operation has an error
`void execute(E&&, F&&)`	Executes a functor in an executor
`auto connect(S&&, R&&)`	Connects the given sender with the given receiver, resulting an `operation_state` object
`void start(O&&)`	Starts an `operation_state` object
`void submit(S&&, R&&)`	Submit a sender and a receiver for execution
`auto schedule(S&&)`	Given a scheduler, returns a sender that can kick-off a chain of processing
`void bulk_execute(E&&, F&&, N)`	Bulk-executes a functor N times in the context of an executor.

The following table lists the concepts defined:

Concept	Description
`executor<E>`	Indicates that the given type can execute work of type `void()`. It has a corresponding `execute()` CPO defined.
`executor_of<E, F>`	Indicates that the given type can execute work of the given type. It has a corresponding `execute()` CPO defined.
`receiver<R, E=exception_ptr>`	Indicates that the given type is a bare-bone receiver. That is, it supports `set_done` and `set_error` (with the given error type)
`receiver_of<R, E=exception_ptr, Vs...>`	Indicates that the given type is a receiver. That is, it supports `set_done` and `set_error` (with the given error type) and `set_value` with the given values types
`sender<S>`	Indicates that the given type is a sender.
`typed_sender<S>`	Indicates that the given type is a typed sender.
`sender_to<S, R>`	Indicates that the given type `S` is a sender compatible with the given receiver type. That is `connect(S, R)` is valid.
`operation_state<O>`	Indicates that the given type is an operation state. That is `start(O)` is valid.
`scheduler<S>`	Indicate that the given type is a scheduler. That is `schedule(S)` is valid and returns a valid sender type.

1.3.2 Concepts, explained

executor

A C++23 `executor` concept matches the way concore looks at an executor: it is able to schedule work. To be noted that all concore executors (`global_executor`, `spawn_executor`, `inline_executor`, etc.) fulfill the `executor` concept.

The way that P0443 defines the concept, an `executor` is able to execute any type of functor compatible with `void()`. While a `task` is a type compatible with `void()`, concore ensures that all the executors have a specialization that takes directly `task`. This is done mostly for type erasure, helping compilation times.

If `ex` is of type `E` that models concept `executor`, then the one can perform work on that executor with a code similar to:

```
concore::execute(ex, [](){ do_work(); });
```

operation_state

An `operation_state` object is essentially a pair between an `executor` object and a `task` object.

Given an operation `op` of type `Oper`, one can start executing it with a code like:

```
concore::start(op);
```

An operation is typically obtained from a `sender` object and a `receiver` object by calling the `connect` CPO:

```
operation_state auto op = concore::connect(snd, recv);
```

scheduler, sender

A `scheduler` is an agent that is capable of starting asynchronous computations. Most often a scheduler is created out of an `executor` object, but there is no direct linkage between the two.

A `scheduler` object can start asynchronous computations by creating a `sender` object. Given a `sched` object that matches the z``scheduler`` concept, then one can obtain a sender in the following way:

```
sender auto snd = concore::schedule(sched);
```

A `sender` object is an object that performs some asynchronous operation in a given execution context. To use a `sender`, one must always pair it with are `receiver`, so that somebody knows about the operation being completed. As shown, above, this pairing can be done with the `connect` function. Thus, putting them all together, one one can start a computation from a `scheduler` if there is a `receiver` object to collect the results, as shown below:

```
receiver recv = ...
sender auto snd = concore::schedule(sched);
operation_state auto op = concore::connect(snd, recv);
concore::start(op);
```

To skip the intermediate step of creating an `operation_state`, one might call `submit`, that essentially combines `connect` and `start`:

```
receiver recv = ...
sender auto snd = concore::schedule(sched);
concore::submit(snd, recv);
```

Also, a `sender` can be directly created from an `executor` by using the `as_sender` type wrapper:

```
receiver recv = ...
sender auto snd = concore::as_sender(ex);
concore::submit(snd, recv);
```

receiver

A `receiver` is the continuation of an asynchronous task. It is aways used to consume the results of a `sender`.

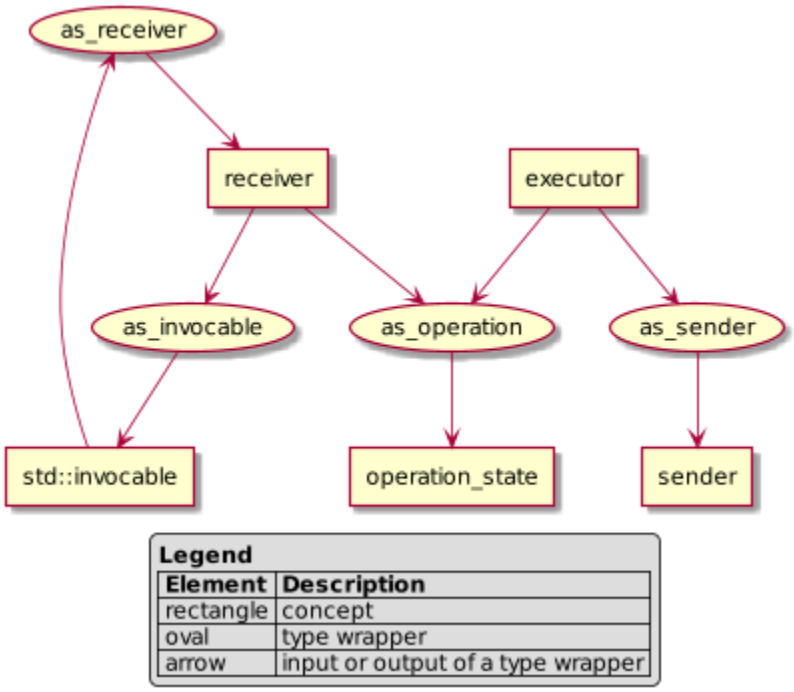
One can create a receiver from an invocable object by using the `as_receiver` wrapper:

```
auto my_fun = []() { on_task_done(); }
receiver recv = concore::as_receiver(my_fun)
```

The division between a `receiver` and a `sender` is a bit blurry. One can add computations that need to be executed asynchronously in any of them. Moreover, one can construct objects that are both `receiver` and `sender` at the same type. This is useful to create chains of computations.

1.3.3 Type wrappers

The following diagrams shows the type wrappers and how they transform different types of objects:



1.4 API

1.4.1 Class Hierarchy

1.4.2 File Hierarchy

1.4.3 Full API