concore

Lucian Radu Teodorescu

Jan 05, 2021

CONTENTS

1 Table of content 3
1.1 Quick-start . . . 3
1.2 Concepts . . . 4
1.3 C++23 executors . . . 12
1.4 API . . . 16

Index 105

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

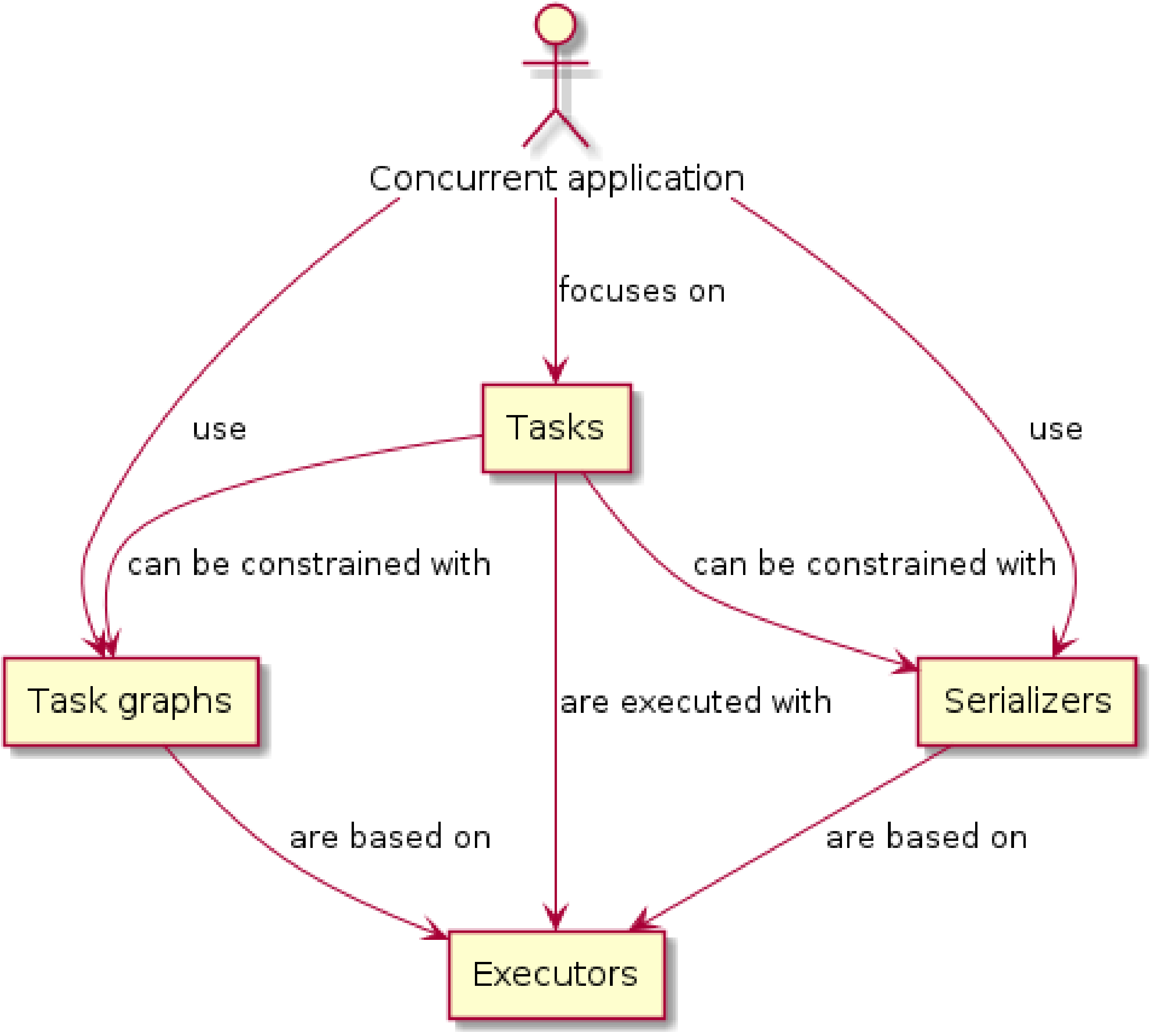
TODO

1.1.3 Examples

TODO

1.2 Concepts

Overview of main concepts:


Concurrent application
focuses on
use
use
Tasks
can be constrained with
can be constrained with
Task graphs
are executed with
Serializers
are based on
are based on
Executors

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.

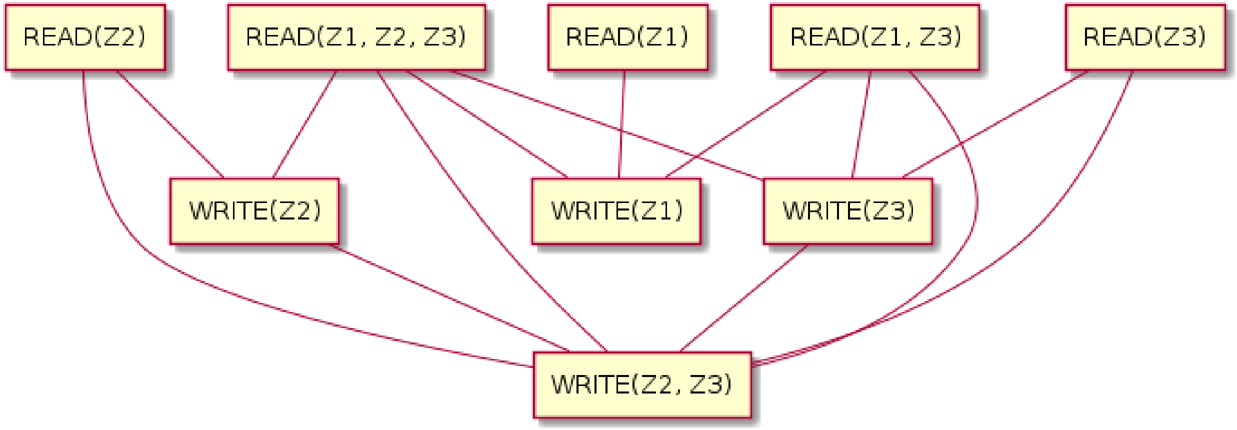
Please note that the *independence* of tasks is heavily dependent on design choices, and maybe less on the internals of the work contained in the tasks.

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.

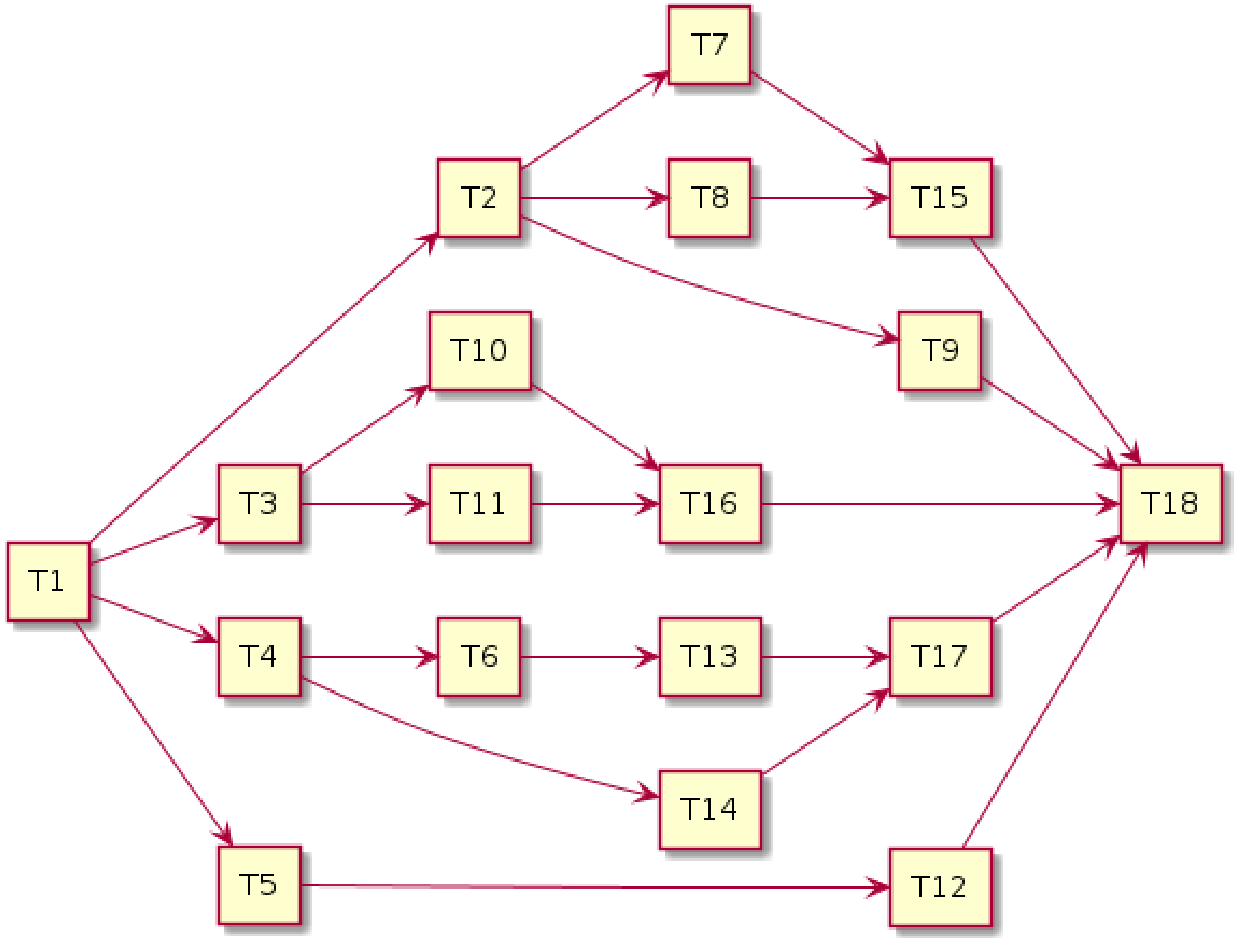
Another popular alternative is to use the *spawn* functionality (either as a free function *`spawn()`*, or through *`spawn_executor`*). This should be called from within the execution of a task and will add the given task to the local queue of the current worker thread; the thread will try to pick up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

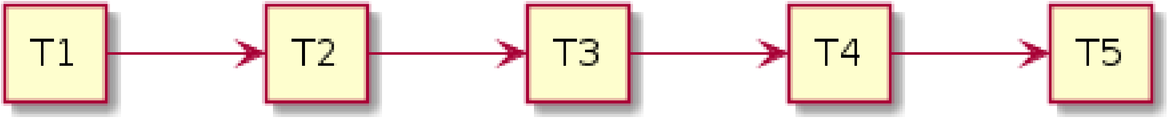
Another way of building task graph is to use concore's abstractions. The nodes in the graph can be modeled with *`chained_task`* objects. Dependencies can be calling *`add_dependency()`* of *`add_dependencies()`* functions.

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

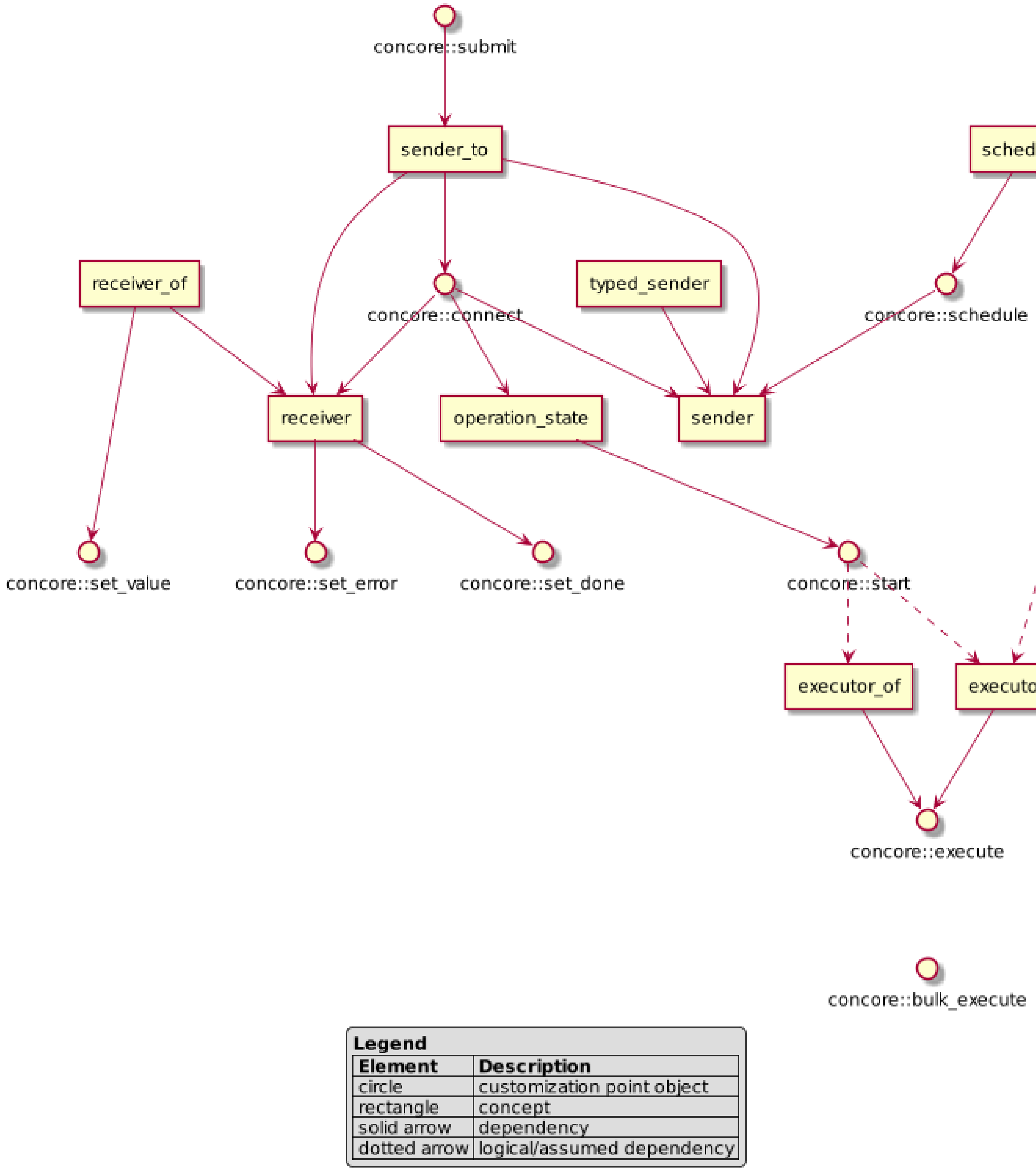
concore implements C++23 executors, as defined by P0443: A Unified Executors Proposal for C++. It is not a complete implementation, but the main features are present. Concore's implementation includes:

- 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


concore::submit
sender_to
scheduler
receiver_of
concore::connect
typed_sender
concore::schedule
receiver
operation_state
sender
concore::set_value
concore::set_error
concore::set_done
concore::start
executor_of
executor
concore::execute
concore::bulk_execute

Legend

Element	Description
circle	customization point object
rectangle	concept
solid arrow	dependency
dotted arrow	logical/assumed dependency

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`.

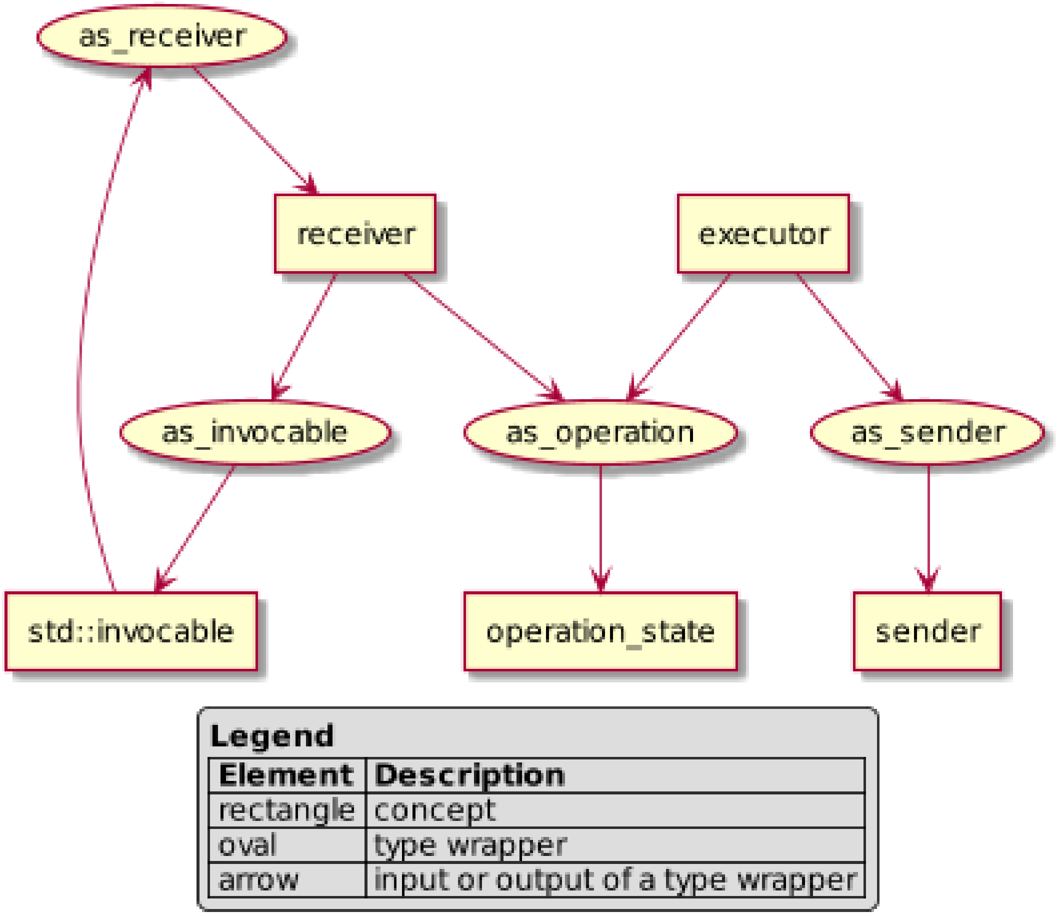
One can create a receiver from an invocable object by using the `as_receiver` wrapper:

```
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:


as_receiver
receiver
executor
as_invocable
as_operation
as_sender
std::invocable
operation_state
sender

Legend

Element	Description
rectangle	concept
oval	type wrapper
arrow	input or output of a type wrapper

1.4 API

1.4.1 Class Hierarchy

1.4.2 File Hierarchy

1.4.3 Full API

Namespaces

Namespace concore

Contents

- *Namespaces*
- *Functions*
- *Typedefs*

Namespaces

- *Namespace concore::v1*

Functions

- *Template Function concore::conc_sort(It, It)*
- *Template Function concore::conc_sort(It, It, task_group)*
- *Template Function concore::conc_sort(It, It, const Comp&, task_group)*
- *Template Function concore::conc_sort(It, It, const Comp&)*

Typedefs

- *Typedef concore::except_fun_t*

Namespace concore::v1

Contents

- *Classes*
- *Enums*
- *Functions*
- *Typedefs*
- *Variables*

Classes

- *Struct already_initialized*
- *Template Struct as_invocable*
- *Template Struct as_operation*
- *Template Struct as_receiver*
- *Template Struct as_sender*
- *Struct bulk_execute_t*
- *Struct connect_t*
- *Struct delegating_executor*
- *Struct dispatch_executor*
- *Struct execute_t*
- *Template Struct executor*
- *Template Struct executor_of*
- *Struct finish_event*
- *Struct finish_task*
- *Struct finish_wait*
- *Struct global_executor*
- *Struct init_data*
- *Struct inline_executor*
- *Struct invocable_archetype*
- *Template Struct operation_state*
- *Struct partition_hints*
- *Struct pipeline_end_t*
- *Template Struct receiver*
- *Struct receiver_invocation_error*
- *Template Struct receiver_of*
- *Struct schedule_t*
- *Template Struct scheduler*
- *Template Struct sender*
- *Template Struct sender_to*
- *Struct set_done_t*
- *Struct set_error_t*
- *Struct set_value_t*
- *Struct spawn_continuation_executor*
- *Struct spawn_executor*
- *Struct start_t*

- *Struct submit_t*
- *Struct tbb_executor*
- *Template Struct typed_sender*
- *Class any_executor*
- *Class binary_semaphore*
- *Class chained_task*
- *Template Class concurrent_dequeue*
- *Template Class concurrent_queue*
- *Class n_serializer*
- *Template Class pipeline*
- *Template Class pipeline_builder*
- *Class rw_serializer*
- *Class rw_serializer::reader_type*
- *Class rw_serializer::writer_type*
- *Class semaphore*
- *Class serializer*
- *Class shared_spin_mutex*
- *Class spin_backoff*
- *Class spin_mutex*
- *Class static_thread_pool*
- *Class task*
- *Class task_group*

Enums

- *Enum partition_method*
- *Enum queue_type*
- *Enum stage_ordering*

Functions

- *Function concore::v1::add_dependencies(std::initializer_list<chained_task>, chained_task)*
- *Function concore::v1::add_dependencies(chained_task, std::initializer_list<chained_task>)*
- *Function concore::v1::add_dependency*
- *Template Function concore::v1::bulk_execute*
- *Template Function concore::v1::conc_for(It, It, const UnaryFunction&, const task_group&, partition_hints)*
- *Template Function concore::v1::conc_for(It, It, const UnaryFunction&, const task_group&)*

- *Template Function concore::v1::conc_for(It, It, const UnaryFunction&, partition_hints)*
- *Template Function concore::v1::conc_for(It, It, const UnaryFunction&)*
- *Template Function concore::v1::conc_for(typename WorkType::iterator, typename WorkType::iterator, WorkType&, const task_group&, partition_hints)*
- *Template Function concore::v1::conc_for(typename WorkType::iterator, typename WorkType::iterator, WorkType&, const task_group&)*
- *Template Function concore::v1::conc_for(typename WorkType::iterator, typename WorkType::iterator, WorkType&, partition_hints)*
- *Template Function concore::v1::conc_for(typename WorkType::iterator, typename WorkType::iterator, WorkType&)*
- *Template Function concore::v1::conc_reduce(It, It, Value, const BinaryOp&, const ReductionOp&, task_group, partition_hints)*
- *Template Function concore::v1::conc_reduce(It, It, Value, const BinaryOp&, const ReductionOp&, task_group)*
- *Template Function concore::v1::conc_reduce(It, It, Value, const BinaryOp&, const ReductionOp&, partition_hints)*
- *Template Function concore::v1::conc_reduce(It, It, Value, const BinaryOp&, const ReductionOp&)*
- *Template Function concore::v1::conc_reduce(typename WorkType::iterator, typename WorkType::iterator, WorkType&, const task_group&, partition_hints)*
- *Template Function concore::v1::conc_reduce(typename WorkType::iterator, typename WorkType::iterator, WorkType&, const task_group&)*
- *Template Function concore::v1::conc_reduce(typename WorkType::iterator, typename WorkType::iterator, WorkType&, partition_hints)*
- *Template Function concore::v1::conc_reduce(typename WorkType::iterator, typename WorkType::iterator, WorkType&)*
- *Template Function concore::v1::conc_scan(It, It, It2, Value, const BinaryOp&, task_group, partition_hints)*
- *Template Function concore::v1::conc_scan(It, It, It2, Value, const BinaryOp&, task_group)*
- *Template Function concore::v1::conc_scan(It, It, It2, Value, const BinaryOp&, partition_hints)*
- *Template Function concore::v1::conc_scan(It, It, It2, Value, const BinaryOp&)*
- *Template Function concore::v1::connect*
- *Template Function concore::v1::execute*
- *Function concore::v1::init*
- *Function concore::v1::is_initialized*
- *Template Function concore::v1::schedule*
- *Template Function concore::v1::set_done*
- *Template Function concore::v1::set_error*
- *Template Function concore::v1::set_value*
- *Function concore::v1::shutdown*
- *Function concore::v1::spawn(task&&, bool)*
- *Function concore::v1::spawn(std::initializer_list<task_function>&&, bool)*

- *Template Function concore::v1::spawn(F&&, bool)*
- *Template Function concore::v1::spawn(F&&, task_group, bool)*
- *Function concore::v1::spawn(std::initializer_list<task_function>&&, task_group, bool)*
- *Template Function concore::v1::spawn_and_wait(F&&)*
- *Function concore::v1::spawn_and_wait(std::initializer_list<task_function>&&, bool)*
- *Template Function concore::v1::start*
- *Template Function concore::v1::submit*
- *Function concore::v1::wait*

Typedefs

- *Typedef concore::v1::task_function*

Variables

- *Variable concore::v1::pipeline_end*

Classes and Structs

Struct already_initialized

- Defined in file_include_concore_init.hpp

Inheritance Relationships

Base Type

- `public runtime_error`

Struct Documentation

`struct` `concore::v1::`**`already_initialized`** : **`public`** runtime_error

Exception thrown when attempting to initialize the library more than once.

Thrown when manually initializing after the library was already initialized, either automatically or by explicitly calling *init()*.

See *init()*, *is_initialized()*

Public Functions

`already_initialized``()`

Default constructor.

Template Struct as_invocable

- Defined in file_include_concore_as_invocable.hpp

Struct Documentation

template<typename **R**>
`struct` `concore::v1::`**`as_invocable`**

Wrapper that transforms a receiver into a functor.

The receiver should model `receiver_of`<>.

Template Parameters

- `R`: The type of the receiver

This will store a reference to the receiver; the receiver must not get out of scope.

When this functor is called *set_value()* will be called on the receiver. If an exception is thrown, the *set_error()* function is called.

If the functor is never called, the destructor of this object will call *set_done()*.

See *as_receiver*

Public Functions

`as_invocable``(`*R* &*r*`)` **`noexcept`**

Constructor from receiver.

`as_invocable``(`*as_invocable* &&*other*`)` **`noexcept`**

Move constructor.

as_invocable &**`operator=`**`(`*as_invocable* &&*other*`)` **`noexcept`**

Move assignment.

`as_invocable``(`**`const`** *as_invocable*&`)` = delete

Copy constructor is DISABLED.

as_invocable &**`operator=`**`(`**`const`** *as_invocable*&`)` = delete

Copy assignment is DISABLED.

`~as_invocable``()`

Destructor.

void **`operator()`**`()` **`noexcept`**

Call operator.

This forwards the call to the receiver by calling *set_value()*. If an exception is thrown during this operation, then *set_error()* is called passing the exception.

See *set_value()*, *set_error()*

Template Struct as_operation

- Defined in file_include_concore_as_operation.hpp

Struct Documentation

template<typename **E**, typename **R**>
`struct concore::v1::as_operation`

Wrapper that transforms an executor and a receiver into an operation.

This is a convenience wrapper to shortcut the usage of scheduler and sender.

Template Parameters

- `E`: The type of the executor; should model *executor*
- `R`: The type of the receiver

This types models the *operation_state* concept

See *operation_state*, *executor*, *receiver*, *as_invocable*, *as_sender*

Public Types

`using executor_type = concore::detail::remove_cvref_t<E>`

The type of executor to be used (remove cvref)

`using receiver_type = concore::detail::remove_cvref_t<R>`

The type of receiver to be used (remove cvref)

Public Functions

`as_operation(executor_type e, receiver_type r) noexcept`

Constructor.

`void start() noexcept`

Starts the asynchronous operation.

Template Struct as_receiver

- Defined in file_include_concore_as_receiver.hpp

Struct Documentation

template<typename **F**>
`struct concore::v1::as_receiver`

Wrapper that transforms a functor into a receiver.

This will implement the operations specific to a receiver given a functor. The receiver will call the functor whenever *set_value()* is called. It will not do anything on *set_done()* and it will terminate the program if *set_error()* is called.

Template Parameters

- `F`: The type of the functor

This types models the `receiver_of`<> concept

See *receiver_of*, *set_value()*, *set_done()*, *set_error()*

Public Functions

`as_receiver`(*F* &&*f*) **`noexcept`**
Constructor.

void **`set_value`**() **`noexcept`**(noexcept(**f_**()))
Called whenever the sender completed the work with success.

void **`set_done`**() **`noexcept`**
Called whenever the work was canceled.

template<typename **E**>
void **`set_error`**(*E*) **`noexcept`**
Called whenever there was an error while performing the work in the sender.

Template Struct as_sender

- Defined in file_include_concore_as_sender.hpp

Struct Documentation

template<typename **E**>
`struct` `concore::v1::`**`as_sender`**
Wrapper that transforms a receiver into a functor.

The receiver should model `receiver_of`<>.

Template Parameters

- `R`: The type of the receiver

This will store a reference to the receiver; the receiver must not get out of scope.

When this functor is called *set_value()* will be called on the receiver. If an exception is thrown, the *set_error()* function is called.

If the functor is never called, the destructor of this object will call *set_done()*.

This types models the *sender* concept.

See *sender*, *receiver_of*, *as_operation*

Public Types

using value_types = Variant<Tuple<>>

The value types that defines the values that this sender sends to receivers.

using error_types = Variant<std::exception_ptr>

The type of error that this sender sends to receiver.

Public Functions

as_sender(*E e*) **noexcept**

Constructor.

template<typename **R**>
as_operation<*E*, *R*> **connect**(*R* &&*r*) &&

The connect CPO that returns an operation state object.

template<typename **R**>
as_operation<*E*, *R*> **connect**(*R* &&*r*) **const** &

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Public Static Attributes

constexpr bool **sends_done** = false

Indicates that this sender never sends a done signal.

Struct bulk_execute_t

- Defined in file_include_concore_execution.hpp

Struct Documentation

struct bulk_execute_t

Customization point object tag for *concore::bulk_execute()*

To add support for *concore::bulk_execute()* to a type `T`, for given functor type `F` and a numeric type `N`, one can define:

```
void tag_invoke(bulk_execute_t, T, F, N);
```

See *concore::bulk_execute()*

Struct connect_t

- Defined in file_include_concore_execution.hpp

Struct Documentation

`struct connect_t`

Customization point object tag for *concore::connect()*

To add support for *concore::connect()* to a type `S`, with the receiver `R`, one can define:

```
void tag_invoke(connect_t, S, R);
```

See *connect()*

Struct delegating_executor

- Defined in file_include_concore_delegating_executor.hpp

Struct Documentation

`struct concore::v1::delegating_executor`

Executor type that forwards the execution to a give functor.

All the functor objects passed to the *execute()* method will be delegated to a function that takes a *task* parameter. This function is passed to the constructor of the class.

Public Types

`using fun_type` = std::function<void (*task*) >

The type of the function to be used to delegate execution to.

Public Functions

`delegating_executor` (*fun_type f*)

Constructor.

void **`execute`** (*task t*) **`const`**

Method called to execute work in this executor.

template<typename **F**>
void **`execute`** (*F &&f*) **`const`**

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Friends

`friend` bool **`operator==`**(*delegating_executor l*, *delegating_executor r*)
Equality operator; always false.

`friend` bool **`operator!=`**(*delegating_executor l*, *delegating_executor r*)
Inequality operator; always true.

Struct dispatch_executor

- Defined in file_include_concore_dispatch_executor.hpp

Struct Documentation

`struct` `concore::v1::`**`dispatch_executor`**
Executor that sends tasks to libdispatch.

This executors wraps the task execution from libdispatch.

This executor provides just basic support for executing tasks, but not other features like cancellation, waiting for tasks, etc.

The executor takes as constructor parameter the priority of the task to be used when enqueueing the task.

Two executor objects are equivalent if their priorities match.

See *global_executor*

Public Types

`enum priority`
The priority of the task to be used.

Values:

`enumerator prio_high`
High-priority tasks.

`enumerator prio_normal`
Tasks with normal priority.

`enumerator prio_low`
Tasks with low priority.

Public Functions

`dispatch_executor`(*priority prio* = *prio_normal*)
Constructor.

template<typename **`F`**>
void **`execute`**(*F &&f*) **`const`**
Method called to execute work in this executor.

Friends

friend bool **operator==**(*dispatch_executor l*, *dispatch_executor r*)
Equality operator.

friend bool **operator!=**(*dispatch_executor l*, *dispatch_executor r*)
Inequality operator.

Struct execute_t

- Defined in file_include_concore_execution.hpp

Struct Documentation

struct execute_t
Type to use for customization point for *concore::execute()*

To add support for *concore::execute()* to a type `T`, for given functor type `F`, one can define:

```
void tag_invoke(execute_t, T, F);
```

See *concore::execute()*

Template Struct executor

- Defined in file_include_concore_execution.hpp

Struct Documentation

template<typename **E**>
struct executor
Concept that defines an executor.

An executor object is an object that can "execute" work. Given a functor compatible with *invocable_archetype*, the executor will be able to execute that function, in some specified manner.

Template Parameters

- `E`: The type that we want to model the concept

Properties that a type needs to have in order for it to be an executor:

- it's copy-constructible
- the copy constructor is nothrow
- it's equality-comparable
- one can call `execute(obj, invocable_archetype{})`, where `obj` is an object of the type

To be able to call *execute()* on an executor, the executor type must have one the following:

- an inner method `execute` that takes a functor
- an associated `execute` free function that takes the executor and a functor
- an customization point `tag_invoke(execute_t, Ex, Fn)`

See *executor_of*, *execute_t*, *execute()*

Template Struct executor_of

- Defined in file_include_concore_execution.hpp

Struct Documentation

template<typename **E**, typename **F**>
`struct executor_of`

Defines an executor that can execute a given functor type.

This is similar to the *executor* concept, but instead of being capable of executing `void()` functors, this can execute functors of the given type `F`

Template Parameters

- `E`: The type that we want to model the concept
- `F`: The type functor that can be called by the executor

See *executor*

Struct finish_event

- Defined in file_include_concore_finish_task.hpp

Struct Documentation

`struct` `concore::v1::`**`finish_event`**

A finish event.

This can be passed to various tasks that would notify this whenever the task is complete. Depending on how the event is configured, after certain number of tasks are completed this triggers an event. This can be used to join the execution of multiple tasks that can run in parallel.

The *notify_done()* function must be called whenever the task is done.

This is created via *finish_task* and *finish_wait*.

Once a finish even is triggered, it cannot be reused anymore.

See *finish_task*, *finish_wait*

Public Functions

void **notify_done**() **const**

Called by other tasks to indicate their completion.

When the right number of tasks have called this, then the event is trigger; that is, executing a task or unblocking some wait call.

Struct finish_task

- Defined in file_include_concore_finish_task.hpp

Struct Documentation

struct concore::v1::**finish_task**

Abstraction that allows executing a task whenever multiple other tasks complete.

This is created with a task (and an executor) and a count number. If the specific number of other tasks will complete, then the given task is executed (in the given executor).

With the help of this class one might implement a concurrent join: when a specific number of task are done, then a continuation is triggered.

This abstraction can also be used to implement simple continuation; it's just like a join, but with only one task to wait on.

This will expose a *finish_event* object with the *event()* function. Other tasks will call *finish_event::notify_done()* whenever they are completed. When the right amount of tasks call it, then the task given at the constructor will be executed.

After constructing a *finish_task* object, and passing the corresponding *finish_event* to the right amount of other tasks, this object can be safely destructed. Its presence will not affect in any way the execution of the task.

Example usage:

```cpp
concore::finish_task done_task([]{
    printf("done.");
    system_cleanup();
}, 3);
// Spawn 3 tasks
auto event = done_task.event();
concore::spawn([event]{
    do_work1();
    event.notify_done();
});
concore::spawn([event]{
    do_work2();
    event.notify_done();
});
concore::spawn([event]{
    do_work3();
    event.notify_done();
});
// When they complete, the first task is triggered
```

See *finish_event*, *finish_wait*

Public Functions

finish_task (*task* &&*t*, *any_executor* *e*, int *count* = 1)
Constructor with a task and executor.

finish_task (*task* &&*t*, int *count* = 1)
Constructor with a task.

template<typename **F**>
finish_task (*F* *f*, *any_executor* *e*, int *count* = 1)
Constructor with a functor and executor.

template<typename **F**>
finish_task (*F* *f*, int *count* = 1)
Constructor with a functor.

finish_event **event** () **const**
Getter for the *finish_event* object that should be distributed to other tasks.

Struct finish_wait

- Defined in file_include_concore_finish_task.hpp

Struct Documentation

struct concore::v1::**finish_wait**
Abstraction that allows waiting on multiple tasks to complete.

This is similar to *finish_task*, but instead of executing a task, this allows the user to wait on all the tasks to complete. This wait, as expected, is a busy-way: other tasks are executed while waiting for the finish event.

Similar to *finish_task*, this can also be used to implement concurrent joins. Instead of spawning a task whenever the tasks are complete, this allows the current thread to wait on the tasks to be executed

This will expose a *finish_event* object with the *event()* function. Other tasks will call *finish_event::notify_done()* whenever they are completed. When the right amount of tasks call it, then the *wait()* method will be 'unblocked'.

Example usage:

```
concore::finish_wait done(3);
auto event = done_task.event();
// Spawn 3 tasks
concore::spawn([event]{
    do_work1();
    event.notify_done();
});
concore::spawn([event]{
    do_work2();
    event.notify_done();
});
concore::spawn([event]{
    do_work3();
    event.notify_done();
});

// Wait for all 3 tasks to complete
done.wait();
```

See *finish_event*, *finish_task*

Public Functions

`finish_wait` (int *count* = 1)

Constructor.

finish_event **`event`** () **`const`**

Getter for the *finish_event* object that should be distributed to other tasks.

void **`wait`** ()

Wait for all the tasks to complete.

This will wait for the right number of calls to the *finish_event::notify_done()* function on the exposed event. Until that, this will attempt to get some work from the system and execute it. Whenever the right number of tasks are completed (i.e., the right amount of calls to *finish_event::notify_done()* are made), then this will be unblocked; it will return as soon as possible.

This can be called several times, but after the first time this is unblocked, the subsequent calls will exit immediately.

Struct global_executor

- Defined in file_include_concore_global_executor.hpp

Struct Documentation

`struct` `concore::v1::`**`global_executor`**

The default global executor type.

This is an executor that passes the tasks directly to concore's task system. Whenever there is a core available, the task is executed.

The executor takes as constructor parameter the priority of the task to be used when enqueueing the task.

Two executor objects are equivalent if their priorities match.

See *inline_executor*, *spawn_executor*

Public Types

`using priority` = detail::task_priority

The priority of the task to be used.

Public Functions

global_executor(*priority prio = prio_normal*)
Constructor.

template<typename **F**>
void **execute**(*F &&f*) **const**
Method called to execute work in the global executor.

void **execute**(*task t*) **const**
This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Public Static Attributes

constexpr auto **prio_critical** = detail::task_priority::critical
Critical-priority tasks.

constexpr auto **prio_high** = detail::task_priority::high
High-priority tasks.

constexpr auto **prio_normal** = detail::task_priority::normal
Tasks with normal priority.

constexpr auto **prio_low** = detail::task_priority::low
Tasks with low priority.

constexpr auto **prio_background** = detail::task_priority::background
Tasks with lowest possible priority.

Friends

friend bool **operator==**(*global_executor l*, *global_executor r*)
Equality operator.

friend bool **operator!=**(*global_executor l*, *global_executor r*)
Inequality operator.

Struct init_data

- Defined in file_include_concore_init.hpp

Struct Documentation

struct `concore::v1::`**init_data**
Configuration data for the concore library.

Store here all the parameters needed to be passed to concore when initializing. Any parameters that are left unfilled will have reasonable defaults in concore.

Public Members

int **`num_workers_`** = {0}
The number of workers we need to create in the task system; 0 = num core available.

int **`reserved_slots_`** = {10}
The number of extra slots we reserve for other threads to temporary join the tasks system.

std::function<void()> **`worker_start_fun_`**
Function to be called at the start of each thread. Use this if you want to do things like setting thread priority, affinity, etc.

Struct inline_executor

- Defined in file_include_concore_inline_executor.hpp

Struct Documentation

`struct` `concore::v1::`**`inline_executor`**
Executor type that executes the work inline.

Whenever `execute` is called with a functor, the functor is directly called. The calling party will be blocked until the functor finishes execution.

Two objects of this type will always compare equal.

Public Functions

template<typename **F**>
void **`execute`**(*F* &&*f*) **`const`**
Method called to execute work in this executor.

Friends

`friend` bool **`operator==`**(*inline_executor*, *inline_executor*)
Equality operator; always true.

`friend` bool **`operator!=`**(*inline_executor*, *inline_executor*)
Inequality operator; always false.

Struct invocable_archetype

- Defined in file_include_concore_execution.hpp

Struct Documentation

struct `concore::v1::`**`invocable_archetype`**

A type representing the archetype of an invocable object.

This essentially represents a `void()` functor.

Public Functions

void **`operator()`**`()` & **`noexcept`**

Type of operation compatible with `void()`.

Template Struct operation_state

- Defined in file_include_concore_execution.hpp

Struct Documentation

template<typename **`OpState`**>
`struct operation_state`

Concept that defines an operation state.

An object whose type satisfies *operation_state* represents the state of an asynchronous operation. It is the result of calling *concore::connect()* with a sender and a receiver.

Template Parameters

- `OpState`: The type that is being checked to see if it's a *operation_state*

A compatible type must implement the *concore::start()* CPO. In addition, any object of this type must be destructible. Only object types model operation states.

See *sender*, *receiver*, *connect()*

Struct partition_hints

- Defined in file_include_concore_partition_hints.hpp

Struct Documentation

struct `concore::v1::`**`partition_hints`**

Hints to alter the behavior of a *conc_for()* or *conc_reduce()* algorithms.

The hints in this structure can influence the behavior of the *conc_for()* and *conc_reduce()* algorithms, but the algorithms can decide to ignore these hints.

In general, the algorithms performs well on a large variety of cases, when the functions being executed per element are not extremely fast. Therefore, manually giving hints to it is not usually needed. If the operations are really fast, the user might want to play with the granularity to ensure that the work unit is sufficiently large.

See partition_method, *conc_for()*, *conc_reduce()*

Public Members

partition_method **`method_`** = *partition_method::auto_partition*

The method of partitioning the input range.

int **`granularity_`** = {-1}

The granularity of the algorithm.

When choosing how many iterations to handle in one task, this parameter can instruct the algorithm to not place less than the value here. This can be used when the iterations are really small, and the task management overhead can become significant.

Does not apply to the upfront_partition method

int **`tasks_per_worker_`** = {-1}

The (maximum) number of tasks to create per worker

Whenever this is set, we ensure that we don't break the work into too many tasks. It has a similar effect to setting the granularity.

If, for example, this is set to 10, and we have 8 workers, then the upfront partition will create maximum 80 tasks. The auto partition will not create more than 80 tasks.

Struct pipeline_end_t

- Defined in file_include_concore_pipeline.hpp

Struct Documentation

`struct pipeline_end_t`

Tag type to help end the expression of building a pipeline.

Template Struct receiver

- Defined in file_include_concore_execution.hpp

Struct Documentation

template<typename **`T`**, typename **`E`** = std::exception_ptr>
`struct receiver`

Concept that defines a bare-bone receiver.

A receiver represents the continuation of an asynchronous operation. An asynchronous operation may complete with a (possibly empty) set of values, an error, or it may be canceled. A receiver has three principal operations corresponding to the three ways an asynchronous operation may complete: *concore::set_value()*, *concore::set_error()*, and *concore::set_done()*. These are collectively known as a receiver's *completion-signal operations*.

Template Parameters

- `T`: The type being checked to see if it's a bare-bone receiver
- `E`: The type of errors that the receiver accepts; default `std::exception_ptr`

The following constraints must hold with respect to receiver's completion-signal operations:

- None of a receiver's completion-signal operations shall be invoked before *concore::start()* has been called on the operation state object that was returned by *concore::connect()* to connect that receiver to a sender.
- Once *concore::start()* has been called on the operation state object, exactly one of the receiver's completion-signal operations shall complete non-exceptionally before the receiver is destroyed.
- If *concore::set_value()* exits with an exception, it is still valid to call *concore::set_error()* or *concore::set_done()* on the receiver.

A bare-bone receiver is a receiver that only checks for the following CPOs:

- *set_done()*
- *set_error()*

The *set_value()* CPO is ignored in a bare-bone receiver, as a receiver may have many ways to be notified about the success of a sender.

In addition to these, the type should be move constructible and copy constructible.

See *receiver_of*, *set_done()*, *set_error()*

Struct receiver_invocation_error

- Defined in file_include_concore_execution.hpp

Inheritance Relationships

Base Types

- `public runtime_error`
- `public nested_exception`

Struct Documentation

`struct` `concore::v1::`**`receiver_invocation_error`** : **`public`** runtime_error, **`public`** nested_exception

Exception type representing a receiver invocation error.

Public Functions

`receiver_invocation_error``()` **`noexcept`**

Default constructor.

Template Struct receiver_of

- Defined in file_include_concore_execution.hpp

Struct Documentation

template<typename **T**, typename **E** = std::exception_ptr, typename ...**Vs**>
`struct receiver_of`

Concept that defines a receiver of a particular kind.

A receiver represents the continuation of an asynchronous operation. An asynchronous operation may complete with a (possibly empty) set of values, an error, or it may be canceled. A receiver has three principal operations corresponding to the three ways an asynchronous operation may complete: *concore::set_value()*, *concore::set_error()*, and *concore::set_done()*. These are collectively known as a receiver's *completion-signal operations*.

Template Parameters

- `T`: The type being checked to see if it's a bare-bone receiver
- `E`: The type of errors that the receiver accepts; default `std::exception_ptr`
- `Vs...`: The types of the values accepted by the receiver

The following constraints must hold with respect to receiver's completion-signal operations:

- None of a receiver's completion-signal operations shall be invoked before *concore::start()* has been called on the operation state object that was returned by *concore::connect()* to connect that receiver to a sender.
- Once *concore::start()* has been called on the operation state object, exactly one of the receiver's completion-signal operations shall complete non-exceptionally before the receiver is destroyed.
- If *concore::set_value()* exits with an exception, it is still valid to call *concore::set_error()* or *concore::set_done()* on the receiver.

This concept checks that all three CPOs are defined (as opposed to *receiver* who only checks *set_done()* and *set_error()*).

This is an extension of the *receiver* concept, but also requiring the *set_value()* CPO to be present, for a given set of value types.

See *receiver*, *set_value()*, *set_done()*, *set_error()*

Struct schedule_t

- Defined in file_include_concore_execution.hpp

Struct Documentation

`struct schedule_t`

Customization point object tag for *concore::schedule()*

To add support for *concore::schedule()* to a type `T`, one can define:

```
auto tag_invoke(schedule_t, T);
```

See *concore::schedule()*

Template Struct scheduler

- Defined in file_include_concore_execution.hpp

Struct Documentation

template<typename **S**>
`struct scheduler`

Concept that defines a scheduler.

A scheduler type allows a *schedule()* operation that creates a sender out of the scheduler. A typical scheduler contains an execution context that will pass to the sender on its creation.

Template Parameters

- `S`: The type that is being checked to see if it's a scheduler

The type that match this concept must be move and copy constructible and must also define the *schedule()* CPO.

See *sender*

Template Struct sender

- Defined in file_include_concore_execution.hpp

Struct Documentation

template<typename **S**>
`struct sender`

Concept that defines a sender.

A sender represents an asynchronous operation not yet scheduled for execution. A sender's responsibility is to fulfill the receiver contract to a connected receiver by delivering a completion signal.

Template Parameters

- `S`: The type that is being checked to see if it's a sender

A sender, once the asynchronous operation is started must successfully call exactly one of these on the associated receiver:

- *set_value()* in the case of success
- *set_done()* if the operation was canceled

- *set_error()* if an exception occurred during the operation, or while calling *set_value()*

The sender starts working when `submit(S, R)` or `start(connect(S, R))` is called passing the sender and a receiver object in. The sender should not execute any other work after calling one of the three completion signals operations. The sender should not finish its work without calling one of these.

A sender should expose a *connect()* customization-point object to connect it to a receiver. Also, typically there is *submit()* CPO for a sender, but this can be inferred from *connect()*.

A sender typically exposes the type of the values it sets, and the type of errors it can generate, but this is not mandatory.

See *receiver*, *receiver_of*, *typed_sender*, *sender_to*

Template Struct sender_to

- Defined in file_include_concore_execution.hpp

Struct Documentation

template<typename **S**, typename **R**>
`struct sender_to`

Concept that brings together a sender and a receiver.

This concept extends the *sender* concept, and ensures that it can connect to the given receiver type. It does that by checking if *concore::connect()* is valid for the given types.

Template Parameters

- `S`: The type of sender that is assessed
- `R`: The type of receiver that the sender must conform to

See *sender*, *receiver*

Struct set_done_t

- Defined in file_include_concore_execution.hpp

Struct Documentation

`struct set_done_t`

Type to use for customization point for *concore::set_done()*

To add support for *concore::set_done()* to a type `T`, one can define:

```
void tag_invoke(set_done_t, T);
```

A type models is a receiver if it has the following three customization point objects defined:

- *concore::set_value()*
- *concore::set_done()*
- *concore::set_error()*

See *concore::set_done()*, *set_value_t*, *set_error_t*

Struct set_error_t

- Defined in file_include_concore_execution.hpp

Struct Documentation

`struct set_error_t`

Type to use for customization point for *concore::set_error()*

To add support for *concore::set_error()* to a type `T`, with an error type `E`, one can define:

```
void tag_invoke(set_error_t, T, E);
```

A type models is a receiver if it has the following three customization point objects defined:

- *concore::set_value()*
- *concore::set_done()*
- *concore::set_error()*

See *concore::set_error()*, *set_value_t*, *set_done_t*

Struct set_value_t

- Defined in file_include_concore_execution.hpp

Struct Documentation

`struct set_value_t`

Type to use for customization point for *concore::set_value()*

To add support for *concore::set_value()* to a type `T`, for value types `Vs...`, one can define:

```
void tag_invoke(set_value_t, T, Vs...);
```

A type models is a receiver if it has the following three customization point objects defined:

- *concore::set_value()*
- *concore::set_done()*
- *concore::set_error()*

See *concore::set_value()*, *set_done_t*, *set_error_t*

Struct spawn_continuation_executor

- Defined in file_include_concore_spawn.hpp

Struct Documentation

struct `concore::v1::`**`spawn_continuation_executor`**

Executor that spawns tasks instead of enqueueing them, but not waking other workers. Similar to calling `spawn(task, false)` on the task.

See *spawn()*, *spawn_continuation_executor*, *global_executor*

Public Functions

template<typename **F**>
void **`execute`**(*F* &&*f*) **const**

Spawns the execution of the given functor.

This will spawn a task that will call the given functor.

Parameters

- `f`: The functor object to be executed

void **`execute`**(*task t*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Friends

friend bool **`operator==`**(*spawn_continuation_executor*, *spawn_continuation_executor*)

Equality operator; always true.

friend bool **`operator!=`**(*spawn_continuation_executor*, *spawn_continuation_executor*)

Inequality operator; always false.

Struct spawn_executor

- Defined in file_include_concore_spawn.hpp

Struct Documentation

struct `concore::v1::`**`spawn_executor`**

Executor that spawns tasks instead of enqueueing them. Similar to calling *spawn()* on the task.

See *spawn()*, *spawn_continuation_executor*, *global_executor*

Public Functions

template<typename **F**>
void **execute**(*F* &&*f*) **const**

Spawns the execution of the given functor.

This will spawn a task that will call the given functor.

Parameters

- `f`: The functor object to be executed

void **execute**(*task* *t*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Friends

friend bool **operator==**(*spawn_executor*, *spawn_executor*)

Equality operator; always true.

friend bool **operator!=**(*spawn_executor*, *spawn_executor*)

Inequality operator; always false.

Struct start_t

- Defined in file_include_concore_execution.hpp

Struct Documentation

struct start_t

Customization point object tag for *concore::start()*

To add support for *concore::start()* to a type `T`, one can define:

```
void tag_invoke(start_t, T);
```

See *concore::start()*

Struct submit_t

- Defined in file_include_concore_execution.hpp

Struct Documentation

struct submit_t

Customization point object tag for *concore::submit()*

To add support for *concore::submit()* to a type `S`, with the receiver `R`, one can define:

```
void tag_invoke(submit_t, S, R);
```

See *concore::submit()*

Struct tbb_executor

- Defined in file_include_concore_tbb_executor.hpp

Struct Documentation

struct `concore::v1::`**tbb_executor**

Executor that sends tasks to TBB.

This executors wraps the task execution from TBB.

This executor provides just basic support for executing tasks, but not other features like cancellation, waiting for tasks, etc.

The executor takes as constructor parameter the priority of the task to be used when enqueueing the task.

Two executor objects are equivalent if their priorities match.

See *global_executor*

Public Types

enum priority

The priority of the task to be used.

Values:

enumerator prio_high

High-priority tasks.

enumerator prio_normal

Tasks with normal priority.

enumerator prio_low

Tasks with low priority.

Public Functions

`tbb_executor`(*priority prio = prio_normal*)
Constructor.

template<typename **F**>
void **`execute`**(*F &&f*) **`const`**
Method called to execute work in this executor.

Friends

`friend` bool **`operator==`**(*tbb_executor l*, *tbb_executor r*)
Equality operator.

`friend` bool **`operator!=`**(*tbb_executor l*, *tbb_executor r*)
Inequality operator.

Template Struct typed_sender

- Defined in file_include_concore_execution.hpp

Struct Documentation

template<typename **S**>
`struct typed_sender`
Concept that defines a typed sender.

This is just like the *sender* concept, but it requires the type information; that is the types that expose the types of values it sets to the receiver and the type of errors it can generate.

Template Parameters

- `S`: The type that is being checked to see if it's a typed sender

See *sender*, *sender_to*

Class any_executor

- Defined in file_include_concore_any_executor.hpp

Class Documentation

`class` `concore::v1::`**`any_executor`**
A polymorphic executor wrapper.

This provides a type erasure on an executor type. It can hold any type of executor object in it and wraps its execution. The type of the wrapped object must model the *executor* concept.

If the any executor is is not initialized with a valid executor, then calling *execute()* on it is undefined behavior.

See *executor*, *inline_executor*, *global_executor*

Public Functions

`any_executor``()` **`noexcept`** = default

Constructs an invalid executor.

`any_executor``(`std::nullptr_t`)` **`noexcept`**

Constructs an invalid executor.

`any_executor``(`**`const`** *any_executor &other*`)` **`noexcept`**

Copy constructor.

`any_executor``(`*any_executor &&other*`)` **`noexcept`**

Move constructor.

template<typename **`Executor`**>
`any_executor``(`*Executor e*`)`

Constructor from a valid executor.

This will construct the object by wrapping the given executor. The given executor must be valid and must model the *executor* concept.

Parameters

- `e`: The executor to be wrapped by this object

See *executor*

any_executor &**`operator=`**`(`**`const`** *any_executor &other*`)` **`noexcept`**

Copy assignment.

any_executor &**`operator=`**`(`*any_executor &&other*`)` **`noexcept`**

Move assignment.

any_executor &**`operator=`**`(`std::nullptr_t`)` **`noexcept`**

Copy assignment from nullptr_t.

template<typename **`Executor`**>
any_executor &**`operator=`**`(`*Executor e*`)`

Assignment operator from another executor.

This will implement assignment from another executor. The given executor must be valid and must model the *executor* concept.

Return Reference to this object

Parameters

- `e`: The executor to be wrapped in this object

See *executor*

`~any_executor``()`

Destructor.

void **`swap`**`(`*any_executor &other*`)` **`noexcept`**

Swaps the content of this object with the content of the given object.

template<typename **`F`**>
void **`execute`**`(`*F &&f*`)` **`const`**

The main execution method of this executor.

This implements the `execute()` CPO for this executor object, making it conform to the executor concept. It forwards the call to the underlying executor.

Parameters

- `f`: Functor to be executed in the wrapped executor

See *executor*

void **execute**(*task* *t*) **const**

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

operator bool() **const noexcept**

Checks if this executor is wrapping another executor.

const std::type_info &**target_type**() **const noexcept**

Returns the type_info for the wrapped executor type.

template<typename **Executor**>
Executor ***target**() **noexcept**

Helper method to get the underlying executor, if its type is specified.

template<typename **Executor**>
const *Executor* ***target**() **const noexcept**

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Friends

friend bool **operator==**(*any_executor* *l*, *any_executor* *r*)

Equality operator.

friend bool **operator==**(*any_executor* *l*, std::nullptr_t)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

friend bool **operator==**(std::nullptr_t, *any_executor* *r*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

friend bool **operator!=**(*any_executor* *l*, *any_executor* *r*)

Inequality operator.

friend bool **operator!=**(*any_executor* *l*, std::nullptr_t *r*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

friend bool **operator!=**(std::nullptr_t *l*, *any_executor* *r*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Class binary_semaphore

- Defined in file_include_concore_low_level_semaphore.hpp

Class Documentation

class concore::v1::**binary_semaphore**

A semaphore that has two states: SIGNALED and WAITING.

It's assumed that the user will not call *signal()* multiple times.

It may be implemented exactly as a *semaphore*, but on some platforms it can be implemented more efficiently.

See *semaphore*

Public Functions

binary_semaphore()

Constructor. Puts the semaphore in the SIGNALED state.

~binary_semaphore()

Destructor.

binary_semaphore(**const** *binary_semaphore*&) = delete

Copy constructor is DISABLED.

void **operator=**(**const** *binary_semaphore*&) = delete

Copy assignment is DISABLED.

binary_semaphore(*binary_semaphore*&&) = delete

Move constructor is DISABLED.

void **operator=**(*binary_semaphore*&&) = delete

Move assignment is DISABLED.

void **wait**()

Wait for the semaphore to be signaled.

This will put the binary semaphore in the WAITING state, and wait for a thread to signal it. The call will block until a corresponding thread will signal it.

See signal(0)

void **signal**()

Signal the binary semaphore.

Puts the semaphore in the SIGNALED state. If there is a thread that waits on the semaphore it will wake it.

Class chained_task

- Defined in file_include_concore_task_graph.hpp

Class Documentation

`class concore::v1::chained_task`

A type of tasks that can be chained with other such tasks to create graphs of tasks.

This is a wrapper on top of a *task*, and cannot be directly interchanged with a *task*. This can directly enqueue the encapsulated *task*, and also, one can create a *task* on top of this one (as this defines the call operator, and it's also a functor).

One can create multiple *chained_task* objects, then call add_dependency() oradd_dependencies() to create dependencies between such task objects. Thus, one can create graphs of tasks from *chained_task* objects.

The built graph must be acyclic. Cyclic graphs can lead to execution stalls.

After building the graph, the user should manually start the execution of the graph by enqueueing a *chained_task* that has no predecessors. After completion, this will try to enqueue follow-up tasks, and so on, until all the graph is completely executed.

A chained task will be executed only after all the predecessors have been executed. If a task has three predecessors it will be executed only when the last predecessor completes. Looking from the opposite direction, at the end of the task, the successors are checked; the number of active predecessors is decremented, and, if one drops to zero, that successor will be enqueued.

The *chained_task* can be configured with an executor this will be used when enqueueing successor tasks.

If a task throws an exception, the handler in the associated *task_group* is called (if set) and the execution of the graph will continue. Similarly, if a task from the graph is canceled, the execution of the graph will continue as if the task wasn't supposed to do anything.

See *task*, add_dependency(), add_dependencies(), *task_group*

Public Functions

`chained_task``()` = default

Default constructor. Constructs an invalid *chained_task*. Such a task cannot be placed in a graph of tasks.

`chained_task``(`*task t*, *any_executor executor* = {}`)`

Constructor.

This will initialize a valid *chained_task*. After this constructor, add_dependency() and add_dependencies() can be called to add predecessors and successors of this task.

Parameters

- `t`: The task to be executed
- `executor`: The executor to be used for the successor tasks (optional)

If this tasks tries to start executing successor tasks it will use the given executor.

If no executor is given, the *spawn_executor* will be used.

See add_dependency(), add_dependencies(), *task*

template<typename **F**>

`chained_task`(*F f*, *any_executor executor* = {})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void **`operator()`**()

The call operator.

This will be called when executing the *chained_task*. It will execute the task received on constructor and then will check if it needs to start executing successors it will try to start executing the successors that don't have any other active predecessors.

This will use the executor given at construction to start successor tasks.

void **`set_exception_handler`**(*except_fun_t except_fun*)

Sets the exception handler for enqueueing tasks.

The exception handler set here will be called whenever an exception is thrown while enqueueing a follow-up task. It will not be called whenever the task itself throws an exception; that will be handled by the exception handler set in the group of the task.

Parameters

- `except_fun`: The function to be called whenever an exception occurs.

See *task_group::set_exception_handler*

`operator bool`() **`const noexcept`**

Bool conversion operator.

Indicates if this is a valid chained task.

void **`clear_next`**()

Clear all the dependencies that go from this task.

This is useful for constructing and destroying task graphs manually.

Template Class concurrent_dequeue

- Defined in file_include_concore_low_level_concurrent_dequeue.hpp

Class Documentation

template<typename **T**>
`class` `concore::v1::`**`concurrent_dequeue`**

Concurrent double-ended queue implementation, for a small number of elements.

This will try to preallocate a vector with enough elements to cover the most common cases. Operations on the concurrent queue when we have few elements are fast: we only make atomic operations, no memory allocation. We only use spin mutexes in this case.

Template Parameters

- `T`: The type of elements to store

If we have too many elements in the queue, we switch to a slower implementation that can grow to a very large number of elements. For this we use regular mutexes.

Note 1: when switching between fast and slow, the FIFO ordering of the queue is lost.

Note 2: for efficiency reasons, the element size should be at least as a cache line (otherwise we can have false sharing when accessing adjacent elements)

Note 3: we expect very-low contention on the front of the queue, and some contention at the end of the queue. And of course, there will be more contention when the queue is empty or close to empty.

Note 4: we expect contention over the atomic that stores the begin/end position in the fast queue

The intent of this queue is to hold tasks in the task system. There, we typically add any enqueued tasks to the end of the queue. The tasks that are spawned while working on some task are pushed to the front of the queue. The popping of the tasks is typically done on the front of the queue, but when stealing tasks, popping is done from the back of the queue trying to maximize locality for nearby tasks.

Public Types

`using value_type` = *T*

The value type stored in the concurrent dequeue.

Public Functions

`concurrent_dequeue`(size_t *expected_size*)

Constructs a new instance of the queue, with the given preallocated size.

If we ever add more elements in our queue than the given limit, our queue starts to become slower.

Parameters

- `expected_size`: How many elements to preallocate in our fast queue.

The number of reserved elements should be bigger than the expected concurrency.

void **`push_back`**(*T* &&*elem*)

Pushes one element in the back of the queue. This is considered the default pushing operation.

void **`push_front`**(*T* &&*elem*)

Push one element to the front of the queue.

bool **`try_pop_front`**(*T* &*elem*)

Try to pop one element from the front of the queue. Returns false if the queue is empty. This is considered the default popping operation.

bool **`try_pop_back`**(*T* &*elem*)

Try to pop one element from the back of the queue. Returns false if the queue is empty.

void **`unsafe_clear`**()

Clears the queue.

Template Class concurrent_queue

- Defined in file_include_concore_data_concurrent_queue.hpp

Class Documentation

template<typename **T**, *queue_type* **conc_type** = *queue_type*::*multi_prod_multi_cons*>
class `concore::v1::`**concurrent_queue**

Concurrent double-ended queue implementation.

Based on the conc_type parameter, this can be:

- single-producer, single-consumer
- single-producer, multi-consumer
- multi-producer, single-consumer
- multi-producer, multi-consumer

Template Parameters

- `T`: The type of elements to store
- `conc_type`: The expected concurrency for the queue

Note, that the implementation for some of these alternatives might coincide.

The queue, has 2 ends:

- the *back*: where new element can be added
- the *front*: from which elements can be extracted

The queue has only 2 operations corresponding to pushing new elements into the queue and popping elements out of the queue.

See *push()*, *try_pop()*

Public Types

using value_type = *T*

The value type of the concurrent queue.

Public Functions

concurrent_queue() = default

Default constructor. Creates a valid empty queue.

~concurrent_queue() = default

Destructor.

concurrent_queue(**const** *concurrent_queue*&) = delete

Copy constructor is DISABLED.

const *concurrent_queue* &**operator=**(**const** *concurrent_queue*&) = delete

Copy assignment is DISABLED.

concurrent_queue(*concurrent_queue*&&) = default

Move constructor.

concurrent_queue &**operator=**(*concurrent_queue*&&) = default

Move assignment.

void **push** (*T &&elem*)

Pushes one element in the back of the queue.

This ensures that is thread-safe with respect to the chosen queue_type concurrency policy.

Parameters

- `elem`: The element to be added to the queue

See *try_pop()*

bool **try_pop** (*T &elem*)

Try to pop one element from the front of the queue.

If the queue is empty, this will return false and not touch the given parameter. If the queue is not empty, it will extract the element from the front of the queue and store it in the given parameter.

Return True if an element was popped; false otherwise.

Parameters

- `elem`: [out] Location where to put the popped element

This ensures that is thread-safe with respect to the chosen queue_type concurrency policy.

See *push()*

Class n_serializer

- Defined in file_include_concore_n_serializer.hpp

Inheritance Relationships

Base Type

- `public std::enable_shared_from_this< n_serializer >`

Class Documentation

class `concore::v1::`**`n_serializer`** : **public** std::enable_shared_from_this<*n_serializer*>

Executor type that allows max N tasks to be executed at a given time.

If the main purpose of other executors is to define where and when tasks will be executed, the purpose of this executor is to introduce constrains between the tasks enqueued into it.

Given *M* tasks to be executed, this serializer ensures that there are no more than *N* tasks executed in parallel. It serializes the executions of this task. After *N* tasks start executing all other tasks enqueued into the serializer are put on hold. As soon as one task is completed a new task is scheduled for execution.

As this executor doesn't know to schedule tasks for executor it relies on one or two given executors to do the scheduling. If a `base_executor` is given, this will be the one used to schedule for execution of tasks whenever a new task is enqueued and the pool on on-hold tasks is empty. E.g., whenever we enqueue the first time in the serializer. If this is not given, the *global_executor* will be used.

If a `cont_executor` is given, this will be used to enqueue tasks after another task is finished; i.e., enqueue the next task. If this is not given, the serializer will use the `base_executor` if given, or *spawn_continuation_executor*.

An *n_serializer* in a concurrent system based on tasks is similar to semaphores for traditional synchronization-based concurrent systems. However, using *n_serializer* objects will not block threads, and if the application has enough other tasks, throughput doesn't decrease.

Guarantees:

- no more than *N* task is executed at once.
- if N==1, behaves like the *serializer* class.

See *serializer*, *rw_serializer*, *any_executor*, *global_executor*, *spawn_continuation_executor*

Public Functions

`n_serializer`(int *N*, *any_executor* *base_executor* = {}, *any_executor* *cont_executor* = {})

Constructor.

If `base_executor` is not given, *global_executor* will be used. If `cont_executor` is not given, it will use `base_executor` if given, otherwise it will use *spawn_continuation_executor* for enqueueing continuations.

Parameters

- `N`: The maximum number of tasks allowed to be run in parallel
- `base_executor`: Executor to be used to enqueue new tasks
- `cont_executor`: Executor that enqueues follow-up tasks

The first executor is used whenever new tasks are enqueued, and no task is in the wait list. The second executor is used whenever a task is completed and we need to continue with the enqueueing of another task. In this case, the default, *spawn_continuation_executor* tends to work better than *global_executor*, as the next task is picked up immediately by the current working thread, instead of going over the most general flow.

See *global_executor*, *spawn_continuation_executor*

template<typename **`F`**>
void **`execute`**(*F* &&*f*) **`const`**

Executes the given functor in the context of the N serializer.

If there are no more than *N* tasks in the serializer, this task will be enqueued in the `base_executor` given to the constructor (default is *global_executor*). If there are already enough other tasks in the serializer, the given task will be placed in a waiting list. When all the previous tasks are executed, this task will also be enqueued for execution.

Parameters

- `f`: The task functor to be enqueued in the serializer

void **`execute`**(*task* *t*) **`const`**

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void **`set_exception_handler`**(*except_fun_t* *except_fun*)

Sets the exception handler for enqueueing tasks.

The exception handler set here will be called whenever an exception is thrown while enqueueing a follow-up task. It will not be called whenever the task itself throws an exception; that will be handled by the exception handler set in the group of the task.

Parameters

- `except_fun`: The function to be called whenever an exception occurs.

Cannot be called in parallel with task enqueueing and execution.

See *task_group::set_exception_handler*

Friends

`friend` bool **`operator==`**`(`*n_serializer l*, *n_serializer r*`)`
Equality operator.

`friend` bool **`operator!=`**`(`*n_serializer l*, *n_serializer r*`)`
Inequality operator.

Template Class pipeline

- Defined in file_include_concore_pipeline.hpp

Class Documentation

template<typename **T**>
`class` `concore::v1::`**`pipeline`**

Implements a pipeline, used to add concurrency to processing of items that need several stages of processing with various ordering constraints.

A pipeline is a sequence of stages in a the processing of a items (lines). All items share the same stages. All the stages operate on the same type (line type).

Template Parameters

- `T`: The type of items that flow through the pipeline.

The pipeline is built with the help of the *pipeline_builder* class.

Even if the stages need to be run in sequence, we might get concurrency out of this structure as we can execute multiple lines concurrently. However, most of the pipelines also add more constraints on the execution of lines.

The first constraint that one might add is the number of lines that can be processed concurrently. This is done by a parameter passed to the constructor.

The other way to constraint the pipeline is to add various ordering constraints for the stages. We might have multiple ordering constraints for stages:

- `in_order`: at most one item can be processed in the stage, and the items are executed in the order in which they are added to the pipeline
- `out_of_order`: at most one item can be processed in the stage, but the order of processing doesn't matter
- `concurrent`: no constraints are set; the items can run concurrently on the stage

The `in_order` type adds the more constraints to a stage; `out_of_order` is a bit more flexible, but still limits the concurrency a lot. The most relaxed mode is `concurrent`.

If a pipeline has only `in_order` stages, then the concurrency of the pipeline grows to the number of stages it has; but typically the concurrency is very limited. We gain concurrency if we can make some of the stages `concurrent`. A pipeline scales well with respect to concurrency if most of its processing is happening in `concurrent` stages.

If a stage processing throws an exception, then, for that particular line, the next stages will not be run. If some of the next stages are `in_order` then the next items will not be blocked by not executing this stage; i.e., the processing in the stage is just skipped.

Example of building a pipeline:

```
auto my_pipeline = concore::pipeline_builder<int>()
    | concore::stage_ordering::concurrent
    | [&](int idx) {
        work1(idx);
    }
    | concore::stage_ordering::out_of_order
    | [&](int idx) {
        work2(idx);
    }
    | concore::pipeline_end;
for ( int i=0; i<100; i++)
    my_pipeline.push(i);
```

See *pipeline_builder*, stage_ordering, *serializer*, *n_serializer*

Public Functions

void **`push`**(*T line_data*)

Pushes a new item (line) through the pipeline.

This will start processing from the first stage and will iteratively pass through all the stages of the pipeline. The same line data is passed to the functors registered with each stage of the pipeline; i.e., all the stages of the pipeline work on the same line.

Parameters

- `line_data`: The data associated with the line

Template Class pipeline_builder

- Defined in file_include_concore_pipeline.hpp

Class Documentation

template<typename **T**>
`class` `concore::v1::`**`pipeline_builder`**

Front-end to create pipeline objects by adding stages, step by step.

This tries to extract the building of the pipeline stages from the *pipeline* class.

Template Parameters

- `T`: The type of the data corresponding to a line.

It just knows how to configure a pipeline and then to create an actual *pipeline* object.

After we get a pipeline object, this builder cannot be used anymore.

Example of building a pipeline:

```
auto my_pipeline = concore::pipeline_builder<MyT>()
    | concore::stage_ordering::concurrent
    | [&](MyT data) {
        work1(data);
    }
    | concore::stage_ordering::out_of_order
    | [&](MyT data) {
        work2(data);
    }
    | concore::pipeline_end;
```

See *pipeline*

Public Functions

`pipeline_builder`(int *max_concurrency* = 0xffff)

Constructs a pipeline object.

Parameters

- `max_concurrency`: The concurrency limit for the pipeline

`pipeline_builder`(int *max_concurrency*, *task_group* *grp*)

Constructs a pipeline object.

Parameters

- `max_concurrency`: The concurrency limit for the pipeline
- `grp`: The group in which tasks need to be executed

`pipeline_builder`(int *max_concurrency*, *task_group* *grp*, *any_executor* *exe*)

Constructs a pipeline object.

Parameters

- `max_concurrency`: The concurrency limit for the pipeline
- `grp`: The group in which tasks need to be executed
- `exe`: The executor to be used by the pipeline

`pipeline_builder`(int *max_concurrency*, *any_executor* *exe*)

Constructs a pipeline object.

Parameters

- `max_concurrency`: The concurrency limit for the pipeline
- `exe`: The executor to be used by the pipeline

template<typename **F**>
pipeline_builder &**`add_stage`**(*stage_ordering* *ord*, *F* &&*work*)

Adds a stage to the pipeline.

This takes a functor of type `void (T)` and an ordering and constructs a stage in the pipeline with them.

Parameters

- `ord`: The ordering for the stage
- `work`: The work to be done in this stage

Template Parameters

- `F`: The type of the work

See stage_ordering

`operator pipeline<T>``()`

Creates the actual pipeline object, ready to process items.

After calling this, we can no longer own any pipeline data, and we cannot add stages any longer. The returned pipeline object is ready to process items with the stages defined by this class.

pipeline<*T*> **`build`**`()`

Creates the actual pipeline object, ready to process items.

After calling this, we can no longer own any pipeline data, and we cannot add stages any longer. The returned pipeline object is ready to process items with the stages defined by this class.

Return Resulting *pipeline* object.

pipeline_builder &**`operator|`**`(`*stage_ordering ord*`)`

Pipe operator to specify the ordering for the next stages.

This allows easily constructing pipelines by using the `|` operator.

Return The same *pipeline_builder* object

Parameters

- `ord`: The ordering to be applied to next stages

template<typename **F**>
pipeline_builder &**`operator|`**`(`*F* &&*work*`)`

Pipe operator to add new stages to the pipeline.

This adds a new stage to the pipeline, using the latest specified stage ordering. If no stage ordering is specified, before adding this stage, the `in_order` is used.

Return The same *pipeline_builder* object

Parameters

- `work`: The work corresponding to the stage; needs to be of type `void(T)`

Template Parameters

- `F`: The type of the functor

pipeline<*T*> **`operator|`**`(`*pipeline_end_t*`)`

Pipe operator to a tag that tells us that we are done building the pipeline.

This will actually finalize the building process and return the corresponding *pipeline* object. After this is called, any other operations on the builder are illegal.

Return The *pipeline* object built by this *pipeline_builder* object.

Class rw_serializer

- Defined in file_include_concore_rw_serializer.hpp

Nested Relationships

Nested Types

- *Class rw_serializer::reader_type*
- *Class rw_serializer::writer_type*

Class Documentation

class `concore::v1::`**`rw_serializer`**

Similar to a serializer but allows two types of tasks: READ and WRITE tasks.

This class is not an executor. It binds together two executors: one for *READ* tasks and one for *WRITE* tasks. This class adds constraints between *READ* and *WRITE* threads.

The *READ* tasks can be run in parallel with other *READ* tasks, but not with *WRITE* tasks. The *WRITE* tasks cannot be run in parallel neither with *READ* nor with *WRITE* tasks.

This class provides two methods to access to the two executors: *reader()* and *writer()*. The *reader()* executor should be used to enqueue *READ* tasks while the *writer()* executor should be used to enqueue *WRITE* tasks.

This implementation slightly favors the *WRITEs*: if there are multiple pending *WRITEs* and multiple pending *READs*, this will execute all the *WRITEs* before executing the *READs*. The rationale is twofold:

- it's expected that the number of *WRITEs* is somehow smaller than the number of *READs* (otherwise a simple serializer would probably work too)
- it's expected that the *READs* would want to *read* the latest data published by the *WRITEs*, so they are aiming to get the latest *WRITE*

Guarantees:

- no more than 1 *WRITE* task is executed at once
- no *READ* task is executed in parallel with other *WRITE* task
- the *WRITE* tasks are executed in the order of enqueueing

See *reader_type*, *writer_type*, *serializer*, *rw_serializer*

Public Functions

`rw_serializer`(*any_executor base_executor* = {}, *any_executor cont_executor* = {})

Constructor.

If `base_executor` is not given, *global_executor* will be used. If `cont_executor` is not given, it will use `base_executor` if given, otherwise it will use *spawn_continuation_executor* for enqueueing continuations.

Parameters

- `base_executor`: Executor to be used to enqueue new tasks

- `cont_executor`: Executor that enqueues follow-up tasks

The first executor is used whenever new tasks are enqueued, and no task is in the wait list. The second executor is used whenever a task is completed and we need to continue with the enqueueing of another task. In this case, the default, *spawn_continuation_executor* tends to work better than *global_executor*, as the next task is picked up immediately by the current working thread, instead of going over the most general flow.

See *global_executor*, *spawn_continuation_executor*

reader_type **reader**() **const**

Returns an executor to enqueue *READ* tasks.

Return The executor for *READ* types

writer_type **writer**() **const**

Returns an executor to enqueue *WRITE* tasks.

Return The executor for *WRITE* types

void **set_exception_handler**(*except_fun_t except_fun*)

Sets the exception handler for enqueueing tasks.

The exception handler set here will be called whenever an exception is thrown while enqueueing a follow-up task. It will not be called whenever the task itself throws an exception; that will be handled by the exception handler set in the group of the task.

Parameters

- `except_fun`: The function to be called whenever an exception occurs.

Cannot be called in parallel with task enqueueing and execution.

See *task_group::set_exception_handler*

class reader_type

The type of the executor used for *READ* tasks.

Objects of this type will be created by *rw_serializer* to allow enqueueing *READ* tasks

Public Functions

reader_type(std::shared_ptr<*impl*> *impl*)

Constructor. Should only be called by *rw_serializer*.

template<typename **F**>
void **execute**(*F* &&*f*) **const**

Enqueue a functor as a write operation in the RW serializer.

Depending on the state of the parent *rw_serializer* object this will enqueue the task immediately (if there are no *WRITE* tasks), or it will place it in a waiting list to be executed later. The tasks on the waiting lists will be enqueued once there are no more *WRITE* tasks.

void **execute**(*task t*) **const**

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Friends

friend bool **operator==**(*reader_type l*, *reader_type r*)
Equality operator.

friend bool **operator!=**(*reader_type l*, *reader_type r*)
Inequality operator.

class writer_type
The type of the executor used for *WRITE* tasks.

Objects of this type will be created by *rw_serializer* to allow enqueueing *WRITE* tasks

Public Functions

writer_type(std::shared_ptr<*impl*> *impl*)
Constructor. Should only be called by *rw_serializer*.

template<typename **F**>
void **execute**(*F &&f*) **const**
Enqueue a functor as a write operation in the RW serializer.

Depending on the state of the parent *rw_serializer* object this will enqueue the task immediately (if there are no other tasks executing), or it will place it in a waiting list to be executed later. The tasks on the waiting lists will be enqueued, in order, one by one. No new *READ* tasks are executed while we have *WRITE* tasks in the waiting list.

void **execute**(*task t*) **const**
This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void **operator()**(*task t*) **const**
Enqueue a functor as a write operation in the RW serializer.

Depending on the state of the parent *rw_serializer* object this will enqueue the task immediately (if there are no other tasks executing), or it will place it in a waiting list to be executed later. The tasks on the waiting lists will be enqueued, in order, one by one. No new *READ* tasks are executed while we have *WRITE* tasks in the waiting list.

Friends

friend bool **operator==**(*writer_type l*, *writer_type r*)
Equality operator.

friend bool **operator!=**(*writer_type l*, *writer_type r*)
Inequality operator.

Class rw_serializer::reader_type

- Defined in file_include_concore_rw_serializer.hpp

Nested Relationships

This class is a nested type of *Class rw_serializer*.

Class Documentation

class `concore::v1::`*`rw_serializer`*`::`**`reader_type`**

The type of the executor used for *READ* tasks.

Objects of this type will be created by *rw_serializer* to allow enqueueing *READ* tasks

Public Functions

`reader_type`(std::shared_ptr<*impl*> *impl*)

Constructor. Should only be called by *rw_serializer*.

template<typename **F**>
void **`execute`**(*F* &&*f*) **`const`**

Enqueue a functor as a write operation in the RW serializer.

Depending on the state of the parent *rw_serializer* object this will enqueue the task immediately (if there are no *WRITE* tasks), or it will place it in a waiting list to be executed later. The tasks on the waiting lists will be enqueued once there are no more *WRITE* tasks.

void **`execute`**(*task* *t*) **`const`**

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Friends

`friend` bool **`operator==`**(*reader_type* *l*, *reader_type* *r*)

Equality operator.

`friend` bool **`operator!=`**(*reader_type* *l*, *reader_type* *r*)

Inequality operator.

Class rw_serializer::writer_type

- Defined in file_include_concore_rw_serializer.hpp

Nested Relationships

This class is a nested type of *Class rw_serializer*.

Class Documentation

class `concore::v1::`*`rw_serializer`*`::`**`writer_type`**

The type of the executor used for *WRITE* tasks.

Objects of this type will be created by *rw_serializer* to allow enqueueing *WRITE* tasks

Public Functions

`writer_type` (std::shared_ptr<*impl*> *impl*)

Constructor. Should only be called by *rw_serializer*.

template<typename **F**>
void **`execute`** (*F* &&*f*) **`const`**

Enqueue a functor as a write operation in the RW serializer.

Depending on the state of the parent *rw_serializer* object this will enqueue the task immediately (if there are no other tasks executing), or it will place it in a waiting list to be executed later. The tasks on the waiting lists will be enqueued, in order, one by one. No new *READ* tasks are executed while we have *WRITE* tasks in the waiting list.

void **`execute`** (*task t*) **`const`**

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void **`operator()`** (*task t*) **`const`**

Enqueue a functor as a write operation in the RW serializer.

Depending on the state of the parent *rw_serializer* object this will enqueue the task immediately (if there are no other tasks executing), or it will place it in a waiting list to be executed later. The tasks on the waiting lists will be enqueued, in order, one by one. No new *READ* tasks are executed while we have *WRITE* tasks in the waiting list.

Friends

`friend` bool **`operator==`** (*writer_type l*, *writer_type r*)

Equality operator.

`friend` bool **`operator!=`** (*writer_type l*, *writer_type r*)

Inequality operator.

Class semaphore

- Defined in file_include_concore_low_level_semaphore.hpp

Class Documentation

class `concore::v1::`**`semaphore`**

The classic "semaphore" synchronization primitive.

It atomically maintains an internal count. The count can always be increased by calling *signal()*, which is always a non-blocking call. When calling *wait()*, the count is decremented; if the count is still positive the call will be non-blocking; if the count goes below zero, the call to *wait()* will block until some other thread calls *signal()*.

See *binary_semaphore*

Public Functions

`semaphore`(int *start_count* = 0)

Constructs a new semaphore instance.

Parameters

- `start_count`: The value that the semaphore count should have at start

`~semaphore`()

Destructor.

`semaphore`(**const** *semaphore*&) = delete

Copy constructor is DISABLED.

void **`operator=`**(**const** *semaphore*&) = delete

Copy assignment is DISABLED.

`semaphore`(*semaphore*&&) = delete

Move constructor is DIABLED.

void **`operator=`**(*semaphore*&&) = delete

Move assignment is DIABLED.

void **`wait`**()

Decrement the internal count and wait on the count to be positive.

If the count of the semaphore is positive this will decrement the count and return immediately. On the other hand, if the count is 0, it wait for it to become positive before decrementing it and returning.

See *signal()*

void **`signal`**()

Increment the internal count.

If there are at least one thread that is blocked inside a *wait()* call, this will wake up a waiting thread.

See *wait()*

Class serializer

- Defined in file_include_concore_serializer.hpp

Class Documentation

class `concore::v1::`**`serializer`**

Executor type that allows only one task to be executed at a given time.

If the main purpose of other executors is to define where and when tasks will be executed, the purpose of this executor is to introduce constrains between the tasks enqueued into it.

Given *N* tasks to be executed, the serializer ensures that there are no two tasks executed in parallel. It serializes the executions of this task. If a tasks starts executing all other tasks enqueued into the serializer are put on hold. As soon as one task is completed a new task is scheduled for execution.

As this executor doesn't know to schedule tasks for executor it relies on one or two given executors to do the scheduling. If a `base_executor` is given, this will be the one used to schedule for execution of tasks whenever a new task is enqueued and the pool on on-hold tasks is empty. E.g., whenever we enqueue the first time in the serializer. If this is not given, the *global_executor* will be used.

If a `cont_executor` is given, this will be used to enqueue tasks after another task is finished; i.e., enqueue the next task. If this is not given, the serializer will use the `base_executor` if given, or *spawn_continuation_executor*.

A serializer in a concurrent system based on tasks is similar to mutexes for traditional synchronization-based concurrent systems. However, using serializers will not block threads, and if the application has enough other tasks, throughput doesn't decrease.

Guarantees:

- no more than 1 task is executed at once.
- the tasks are executed in the order in which they are enqueued.

See *any_executor*, *global_executor*, *spawn_continuation_executor*, *n_serializer*, *rw_serializer*

Public Functions

`serializer`(*any_executor* *base_executor* = {}, *any_executor* *cont_executor* = {})

Constructor.

If `base_executor` is not given, *global_executor* will be used. If `cont_executor` is not given, it will use `base_executor` if given, otherwise it will use *spawn_continuation_executor* for enqueueing continuations.

Parameters

- `base_executor`: Executor to be used to enqueue new tasks
- `cont_executor`: Executor that enqueues follow-up tasks

The first executor is used whenever new tasks are enqueued, and no task is in the wait list. The second executor is used whenever a task is completed and we need to continue with the enqueueing of another task. In this case, the default, *spawn_continuation_executor* tends to work better than *global_executor*, as the next task is picked up immediately by the current working thread, instead of going over the most general flow.

See *global_executor*, *spawn_continuation_executor*

template<typename **F**>
void **execute**(*F* &&*f*) **const**

Executes the given functor as a task in the context of the serializer.

If there are no tasks in the serializer, the given functor will be enqueued as a task in the `base_executor` given to the constructor (default is *global_executor*). If there are already other tasks in the serializer, the given functor/task will be placed in a waiting list. When all the previous tasks are executed, this task will also be enqueued for execution.

Parameters

- `f`: The functor to be executed

void **execute**(*task* *t*) **const**

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

void **set_exception_handler**(*except_fun_t* *except_fun*)

Sets the exception handler for enqueueing tasks.

The exception handler set here will be called whenever an exception is thrown while enqueueing a follow-up task. It will not be called whenever the task itself throws an exception; that will be handled by the exception handler set in the group of the task.

Parameters

- `except_fun`: The function to be called whenever an exception occurs.

Cannot be called in parallel with task enqueueing and execution.

See *task_group::set_exception_handler*

Friends

friend bool **operator==**(*serializer* *l*, *serializer* *r*)

Equality operator.

friend bool **operator!=**(*serializer* *l*, *serializer* *r*)

Inequality operator.

Class shared_spin_mutex

- Defined in file_include_concore_low_level_shared_spin_mutex.hpp

Class Documentation

class `concore::v1::`**shared_spin_mutex**

A shared (read-write) mutex class that uses CPU spinning.

For mutexes that protect very small regions of code, a *shared_spin_mutex* can be much faster than a traditional shared_mutex. Instead of taking a lock, this will spin on the CPU, trying to avoid yielding the CPU quanta.

The ownership of the mutex can fall in 3 categories:

- no ownership no thread is using the mutex

- exclusive ownership only one thread can access the mutex, exclusively (*WRITE* operations)
- shared ownership multiple threads can access the mutex in a shared way (*READ* operations)

While one threads acquires exclusive ownership, no other thread can have shared ownership. Multiple threads can have a shared ownership over the mutex.

This implementation favors exclusive ownership versus shared ownership. If a thread is waiting for exclusive ownership and one thread is waiting for the shared ownership, the thread that waits on the exclusive ownership will be granted the ownership first.

This uses an exponential backoff spinner. If after some time doing small waits it cannot enter the critical section, it will yield the CPU quanta of the current thread.

Spin shared mutexes should only be used to protect very-small regions of code; a handful of CPU instructions. For larger scopes, a traditional shared mutex may be faster; but then, think about using *rw_serializer* to avoid mutexes completely.

See *spin_mutex*, *spin_backoff*, *rw_serializer*

Public Functions

`shared_spin_mutex``()` = default

Default constructor.

Constructs a shared spin mutex that is in the *no ownership* state.

`~shared_spin_mutex``()` = default

`shared_spin_mutex``(`**`const`** *shared_spin_mutex*&`)` = delete

Copy constructor is DISABLED.

shared_spin_mutex &**`operator=`**`(`**`const`** *shared_spin_mutex*&`)` = delete

Copy assignment is DISABLED.

`shared_spin_mutex``(`*shared_spin_mutex*&&`)` = delete

Move constructor is DIABLED.

shared_spin_mutex &**`operator=`**`(`*shared_spin_mutex*&&`)` = delete

Move assignment is DIABLED.

void **`lock`**`()`

Acquires exclusive ownership of the mutex.

This will put the mutex in the *exclusive ownership* case. If other threads have exclusive or shared ownership, this will wait until those threads are done

Uses a *spin_backoff* to spin while waiting for the ownership to be free. When exiting this function the mutex will be exclusively owned by the current thread.

An *unlock()* call must be made for each call to *lock()*.

See *try_lock()*, *unlock()*, *lock_shared()*

bool **`try_lock`**`()`

Tries to acquire exclusive ownership; returns false it fails the acquisition.

This is similar to *lock()* but does not wait for the mutex to be free again. If the mutex is acquired by a different thread, or if the mutex has shared ownership this will return false.

Return True if the mutex exclusive ownership was acquired; false if the mutex is busy

An *unlock()* call must be made for each call to this method that returns true.

See *lock()*, *unlock()*

void **`unlock`**`()`

Releases the exclusive ownership on the mutex.

This needs to be called for every *lock()* and for every *try_lock()* that returns true. It should not be called without a matching *lock()* or *try_lock()*.

See *lock()*, *try_lock()*

void **`lock_shared`**`()`

Acquires shared ownership of the mutex.

This will put the mutex in the *shared ownership* case. If other threads have exclusive ownership, this will wait until those threads are done.

Uses a *spin_backoff* to spin while waiting for the ownership to be free. When exiting this function the mutex will be exclusively owned by the current thread.

An *unlock_shared()* call must be made for each call to *lock()*.

See *try_lock_shared()*, *unlock_shared()*, *lock()*

bool **`try_lock_shared`**`()`

Tries to acquire shared ownership; returns false it fails the acquisition.

This is similar to *lock_shared()* but does not wait for the mutex to be free again. If the mutex is exclusively acquired by a different thread this will return false.

Return True if the mutex shared ownership was acquired; false if the mutex is busy

An *unlock_shared()* call must be made for each call to this method that returns true.

See *lock_shared()*, *unlock_shared()*

void **`unlock_shared`**`()`

Releases the sjared ownership on the mutex.

This needs to be called for every *lock_shared()* and for every *try_lock_shared()* that returns true. It should not be called without a matching *lock_shared()* or *try_lock_shared()*.

See *lock_shared()*, *try_lock_shared()*

Class spin_backoff

- Defined in file_include_concore_low_level_spin_backoff.hpp

Class Documentation

class `concore::v1::`**`spin_backoff`**

Class that can spin with exponential backoff.

This is intended to be used for implement spin-wait algorithms. It is assumed that the thread that is calling this will wait on some resource from another thread, and the other thread should release that resource shortly. Instead of giving up the CPU quanta, we prefer to spin a bit until we can get the resource

This will spin with an exponential long pause; after a given threshold this will just yield the CPU quanta of the current thread.

See concore::spin_mutex

Public Functions

void **`pause`**`()`

Pauses a short while.

Calling this multiple times will pause more and more. In the beginning the pauses are short, without yielding the CPU quanta of the current thread. But, after a threshold this attempts to give up the CPU quanta for the current executing thread.

Class spin_mutex

- Defined in file_include_concore_low_level_spin_mutex.hpp

Class Documentation

class `concore::v1::`**`spin_mutex`**

Mutex class that uses CPU spinning while attempting to take the lock.

For mutexes that protect very small regions of code, a *spin_mutex* can be much faster than a traditional mutex. Instead of taking a lock, this will spin on the CPU, trying to avoid yielding the CPU quanta.

This uses an exponential backoff spinner. If after some time doing small waits it cannot enter the critical section, it will yield the CPU quanta of the current thread.

Spin mutexes should only be used to protect very-small regions of code; a handful of CPU instructions. For larger scopes, a traditional mutex may be faster; but then, think about using *serializer* to avoid mutexes completely.

See *spin_backoff*

Public Functions

spin_mutex() = default

Default constructor.

Constructs a spin mutex that is not acquired by any thread.

~spin_mutex() = default

Destructor.

spin_mutex(**const** *spin_mutex*&) = delete

Copy constructor is DISABLED.

spin_mutex &**operator=**(**const** *spin_mutex*&) = delete

Copy assignment is DISABLED.

spin_mutex(*spin_mutex*&&) = delete

Move constructor is DISABLED.

spin_mutex &**operator=**(*spin_mutex*&&) = delete

Move assignment is DIABLED.

void **lock**()

Acquires ownership of the mutex.

Uses a *spin_backoff* to spin while waiting for the ownership to be free. When exiting this function the mutex will be owned by the current thread.

An *unlock()* call must be made for each call to *lock()*.

See *try_lock()*, *unlock()*

bool **try_lock**()

Tries to lock the mutex; returns false if the mutex is not available.

This is similar to *lock()* but does not wait for the mutex to be free again. If the mutex is acquired by a different thread, this will return false.

Return True if the mutex ownership was acquired; false if the mutex is busy

An *unlock()* call must be made for each call to this method that returns true.

See *lock()*, *unlock()*

void **unlock**()

Releases the ownership on the mutex.

This needs to be called for every *lock()* and for every *try_lock()* that returns true. It should not be called without a matching *lock()* or *try_lock()*.

See *lock()*, *try_lock()*

Class static_thread_pool

- Defined in file_include_concore_thread_pool.hpp

Class Documentation

class `concore::v1::`**`static_thread_pool`**

A pool of threads that can execute work.

This is constructed with the number of threads that are needed in the pool. Once constructed, these threads cannot be detached from the pool. The user is allowed to attach other threads to the pool, but without any possibility of detaching them earlier than the destruction of the pool. There is no automatic resizing of the pool.

The user can manually signal the thread pool to stop processing items, and/or wait for the existing work items to drain out.

When destructing this object, the implementation ensures to wait on all the in-progress items.

Any scheduler or executor objects that are created cannot exceed the lifetime of this object.

Properties of the *static_thread_pool* executor:

- blocking.always
- relationship.fork
- outstanding_work.untracked
- bulk_guarantee.parallel
- mapping.thread

See *executor*, *scheduler*, *sender*

Public Types

`using scheduler_type` = detail::thread_pool_scheduler

The type of scheduler that this object exposes.

`using executor_type` = detail::thread_pool_executor

The type of executor that this object exposes.

Public Functions

`static_thread_pool` (std::size_t *num_threads*)

Constructs a thread pool.

This thread pool will create the given number of "internal" threads. This number of threads cannot be changed later on. In addition to these threads, the user might manually add other threads in the pool by calling the *attach()* method.

Parameters

- `num_threads`: The number of threads to statically create in the thread pool

See *attach()*

static_thread_pool(**const** *static_thread_pool*&) = delete

Copy constructor is DISABLED.

static_thread_pool &**operator=**(**const** *static_thread_pool*&) = delete

Copy assignment is DISABLED.

static_thread_pool(*static_thread_pool*&&) = default

Move constructor.

static_thread_pool &**operator=**(*static_thread_pool*&&) = default

Move assignment.

~static_thread_pool()

Destructor for the static pool.

Ensures that all the tasks already in the pool are drained out before destructing the pool. New tasks will not be executed anymore in the pool.

This is equivalent to calling *stop()* and then *wait()*.

void **attach**()

Attach the current thread to the thread pool.

The thread that is calling this will temporary join this thread pool. The thread will behave as if it was created during the constructor of this class. The thread will be released from the pool (and return to the caller) whenever the *stop()* and *wait()* are releasing the threads from this pool.

If the thread pool is stopped, this will exit immediately without attaching the current thread to the thread pool.

See *stop()*, *wait()*

void **join**()

Alternative name for *attach()*

void **stop**()

Signal all work to complete.

This will signal the thread pool to stop working as soon as possible. This will return immediately without waiting on the worker threads to complete.

After calling this, no new work will be taken by the thread pool.

This will cause the threads attached to this pool to detach (after completing ongoing work).

This is not thread-safe. Ensure that this is called from a single thread.

void **wait**()

Wait for all the threads attached to the thread pool to complete.

If not already stopped, it will signal the thread pool for completion. Calling just *wait()* is similar to calling *stop()* and then *wait()*.

If there are ongoing tasks in the pool that are still executing, this will block until all these tasks are completed.

After the call to this function, all threads are either terminated or detached from the thread pool.

See *stop()*, *attach()*

scheduler_type **scheduler**() **noexcept**

Returns a scheduler that can be used to schedule work here.

The returned scheduler object can be used to create sender objects that may be used to submit receiver objects to this thread pool.

The returned object has the following properties:

- execution::allocator
- execution::allocator(std::allocator<void>())

See *executor()*

executor_type `executor() noexcept`

Returns an executor object that can add work to this thread pool.

This returns an executor object that can be used to submit function objects to be executed by this thread pool.

The returned object has the following properties:

- execution::blocking.possibly
- execution::relationship.fork
- execution::outstanding_work.untracked
- execution::allocator
- execution::allocator(std::allocator<void>())

See *scheduler()*

Class task

- Defined in file_include_concore_task.hpp

Class Documentation

`class concore::v1::task`

A task. Core abstraction for representing an independent unit of work.

A task can be enqueued into an *executor* and executed at a later time. That is, this represents work that can be scheduled.

Tasks have move-only semantics, and disable copy semantics. Also, the library prefers to move tasks around instead of using shared references to the task. That means that, after construction and initialization, once passed to an executor, the task cannot be modified.

It is assumed that a task can only be executed once.

Ensuring correctness when working with tasks

Within the *independent unit of work* definition, the word **independent** is crucial. It is the one that guarantees thread-safety of the applications relying on tasks to represent concurrency.

A task needs to be independent, meaning that **it must not be run in parallel with other tasks that touch the same data** (and one of them is writing). In other words, it enforces no data races, and no corruptions (in this context data races have the negative effect and they represent undefined behavior).

Please note that this does not say that there can't be two tasks that touch the same data (and one of them is writing). It says that if we have such case we need to ensure that these tasks are not running in parallel. In other words, one need to apply *constraints* on these tasks to ensure that they re not run in parallel.

If constraints are added on the tasks ensuring that there are no two conflicting tasks that run in parallel, then we can achieve concurrency without data races.

At the level of *task* object, there is no explicit way of adding constraints on the tasks. The constraints can be added on top of tasks. See *chained_task* and *serializer*.

task_function and *task_group*

A task is essentially a pair of a `task_function` an a *task_group*. The `task_function` part offers storage for the work associated with the task. The *task_group* part offers a way of controlling the task execution.

One or most tasks can belong to a *task_group*. To add a task to an existing *task_group* pass the *task_group* object to the constructor of the task object. By using a *task_group* the user can tell the system to cancel the execution of all the tasks that belong to the *task_group*. It can also implement logic that depends on the the *task_group* having no tasks attached to it.

See task_function, *task_group*, *chained_task*, *serializer*

Public Functions

`task``()` = default

Default constructor.

Brings the task into a valid state. The task has no action to be executed, and does not belong to any task group.

template<typename **`T`**>
`task``(`*T ftor*`)`

Constructs a new task given a functor.

When the task will be executed, the given functor will be called. This typically happens on a different thread than this constructor is called.

Parameters

- `ftor`: The functor to be called when executing task.

Template Parameters

- `T`: The type of the functor. Must be compatible with task_function.

To be assumed that the functor will be called at a later time. All the resources needed by the functor must be valid at that time.

template<typename **`T`**>
`task``(`*T ftor*, *task_group grp*`)`

Constructs a new task given a functor and a task group.

When the task will be executed, the given functor will be called. This typically happens on a different thread than this constructor is called.

Parameters

- `ftor`: The functor to be called when executing task.
- `grp`: The task group to place the task in the group.

Template Parameters

- `T`: The type of the functor. Must be compatible with task_function.

To be assumed that the functor will be called at a later time. All the resources needed by the functor must be valid at that time.

Through the given group one can cancel the execution of the task, and check (indirectly) when the task is complete.

After a call to this constructor, the group becomes "active". Calling *task_group::is_active()* will return true.

See *get_task_group()*

template<typename **T**>
task &**operator=**(*T ftor*)

Assignment operator from a functor.

This can be used to change the task function inside the task.

Return The result of the assignment

Parameters

- `ftor`: The functor to be called when executing task.

Template Parameters

- `T`: The type of the functor. Must be compatible with task_function.

~task()

Destructor.

If the task belongs to the group, the group will contain one less active task. If this was the last task registered in the group, after this call, calling *task_group::is_active()* will yield false.

task(*task*&&) = default

Move constructor.

task &**operator=**(*task*&&) = default

Move operator.

task(**const** *task*&) = delete

Copy constructor is DISABLED.

task &**operator=**(**const** *task*&) = delete

Copy assignment operator is DISABLED.

void **swap**(*task &other*)

Swap the content of the task with another task.

Parameters

- `other`: The other task

operator bool() **const noexcept**

Bool conversion operator.

Indicates if a valid functor is set into the tasks, i.e., if there is anything to be executed.

void **operator()**()

Function call operator; performs the action stored in the task.

This is called by the execution engine whenever the task is ready to be executed. It will call the functor stored in the task.

The functor can throw, and the execution system is responsible for catching the exception and ensuring its properly propagated to the user.

This is typically called after some time has passed since task creation. The user must ensure that the functor stored in the task is safe to be executed at that point.

`const` *task_group* `&`**`get_task_group`**`() const`

Gets the task group.

This allows the users to consult the task group associated with the task and change it.

Return The group associated with this task.

Class task_group

- Defined in file_include_concore_task_group.hpp

Class Documentation

`class` `concore::v1::`**`task_group`**

Used to control a group of tasks (cancellation, waiting, exceptions).

Tasks can point to one *task_group* object. A *task_group* object can point to a parent *task_group* object, thus creating hierarchies of *task_group* objects.

task_group implements shared-copy semantics. If one makes a copy of a *task_group* object, the actual value of the *task_group* will be shared. For example, if we cancel one *task_group*, the second *task_group* is also canceled. concore takes advantage of this type of semantics and takes all *task_group* objects by copy, while the content is shared.

Scenario 1: cancellation User can tell a *task_group* object to cancel the tasks. After this, any tasks that use this *task_group* object will not be executed anymore. Tasks that are in progress at the moment of cancellation are not by default canceled. Instead, they can check from time to time whether the task is canceled.

If a parent *task_group* is canceled, all the tasks belonging to the children *task_group* objects are canceled.

Scenario 2: waiting for tasks A *task_group* object can be queried to check if all the tasks in a *task_group* are done executing. Actually, we are checking whether they are still active (the distinction is small, but can be important in some cases). Whenever all the tasks are done executing (they don't reference the *task_group* anymore) then the *task_group* object can tell that. One can easily query this property by calling *is_active()*

Also, one can spawn a certain number of tasks, associating a *task_group* object with them and wait for all these tasks to be completed by waiting on the *task_group* object. This is an active wait: the thread tries to execute tasks while waiting (with the idea that it will try to speed up the completion of the tasks) the waiting algorithm can vary based on other factors.

Scenario 3: Exception handling One can set an exception handler to the *task_group*. If a task throws an exception, and the associated *task_group* has a handler set, then the handler will be called. This can be useful to keep track of exceptions thrown by tasks. For example, one might want to add logging for thrown exceptions.

See *task*

Public Functions

task_group()

Default constructor.

Creates an empty, invalid *task_group*. No operations can be called on it. This is used to mark the absence of a real *task_group*.

See *create()*

~task_group()

Destructor.

task_group(**const** *task_group*&) = default

Copy constructor.

Creates a shared-copy of this object. The new object and the old one will share the same implementation data.

task_group(*task_group*&&) = default

Move constructor; rarely used.

task_group &**operator=**(**const** *task_group*&) = default

Assignment operator.

Creates a shared-copy of this object. The new object and the old one will share the same implementation data.

Return The result of the assignment

task_group &**operator=**(*task_group*&&) = default

Move assignment; rarely used.

operator bool() **const**

Checks if this is a valid task group object.

Returns `true` if this object was created by *create()* or if it's a copy of an object created by calling *create()*.

Such an object is *valid*, and operations can be made on it. Tasks will register into it and they can be influenced by the *task_group* object.

An object for which this returns `false` is considered invalid. It indicates the absence of a real *task_group* object.

See *create()*, *task_group()*

void **set_exception_handler**(*except_fun_t except_fun*)

Set the function to be called whenever an exception is thrown by a task.

On execution, tasks can throw exceptions. If tasks have an associated *task_group*, one can use this function to register an exception handler that will be called for exceptions.

Parameters

- `except_fun`: The function to be called on exceptions

The given exception function will be called each time a new exception is thrown by a task belonging to this *task_group* object.

void **cancel**()

Cancels the execution tasks in the group.

All tasks from this task group scheduled for execution that are not yet started are canceled they won't be executed anymore. If there are tasks of this group that are in execution, they can continue execution until the end. However, they have ways to check if the task group is canceled, so that they can stop prematurely.

Tasks that are added to the group after the group was canceled will be not executed.

To get rid of the cancellation, one can cal *clear_cancel()*.

See *clear_cancel()*, *is_cancelled()*

void **`clear_cancel`**`()`

Clears the cancel flag; new tasks can be executed again.

This reverts the effect of calling *cancel()*. Tasks belonging to this group can be executed once more after clear_clance() is called.

Note, once individual tasks were decided that are canceled and not executed, this *clear_cancel()* cannot revert that. Those tasks will be forever not-executed.

See *cancel()*, *is_cancelled()*

bool **`is_cancelled`**`()` **`const`**

Checks if the tasks overseen by this object are canceled.

This will return `true` after *cancel()* is called, and `false` if *clear_cancel()* is called. If this return `true` it means that tasks belonging to this group will not be executed.

Return True if the task group is canceled, False otherwise.

cancel(), *clear_cancel()*

bool **`is_active`**`()` **`const`**

Checks whether there are active tasks in this group.

Creating one task into the task group will make the task group active, regardless of whether the task is executed or not. The group will become non-active whenever all the tasks created in the group are destroyed.

Return True if active, False otherwise.

One main assumption of task_groups is that, if a task is created in a *task_group*, then it is somehow on the path to execution (i.e., enqueued in some sort of executor).

This can be used to check when all tasks from a group are completed.

If a group has sub-groups which have active tasks, this will return true.

See *task*

Public Static Functions

task_group **`create`**`(`**`const`** *task_group* *&parent* = {}`)`

Creates a *task_group* object, with an optional parent.

As opposed to a default constructor, this creates a valid *task_group* object. Operations (canceling, waiting) can be performed on objects created by this function.

Return The task group created.

Parameters

- `parent`: The parent of the *task_group* object (optional)

The optional `parent` parameter allows one to create hierarchies of *task_group* objects. A hierarchy can be useful, for example, for canceling multiple groups of tasks all at once.

See *task_group()*

task_group **`current_task_group`**`()`

Returns the *task_group* object for the current running task.

If there is no task running, this will return an empty (i.e., default-constructed) object. If there is a running task on this thread, it will return the *task_group* object for the currently running task.

Return The task group associated with the current running task

The intent of this function is to be called from within running tasks.

This uses thread-local-storage to store the *task_group* of the current running task.

See *is_current_task_cancelled()*, *task_group()*

bool **`is_current_task_cancelled`**`()`

Determines if current task cancelled.

This should be called from within tasks to check if the *task_group* associated with the current running task was cancelled.

Return True if current task cancelled, False otherwise.

The intent of this function is to be called from within running tasks.

See *current_task_group()*

task_group **`set_current_task_group`**`(`**`const`** *task_group* &*grp*`)`

Sets the task group for the current worker.

This is used by implementation of certain algorithms to speed up the use of task groups. Not intended to be heavily used. To be used with care.

Return The previous set task group (if any).

Parameters

- `grp`: The new group to be set for the current worker.

In general, after setting a task group, one may want to restore the old task group. This is why the function returns the previous *task_group* object.

Enums

Enum partition_method

- Defined in file_include_concore_partition_hints.hpp

Enum Documentation

enum concore::v1::**partition_method**

The method of dividing the work for concurrent algorithms on ranges.

Using this would provide a hint the *conc_for()*, *conc_reduce()* or conc_scan() algorithms on how to partition the input data.

The implementation of the algorithms may choose not to follow the specified method. Typically the default method (auto_partition) works good enough, so most users don't need to change this.

See *partition_hints*, *conc_for()*, *conc_reduce()*, conc_scan()

Values:

enumerator auto_partition

Automatically partitions the data, trying to maximize locality.

This method tries to create as many tasks as needed to fill the available workers, but tries not to split the work too much to reduce locality. If only one worker is free to do work, this method tries to put all the iterations in the task without splitting the work.

Whenever new workers can take tasks, this method tries to ensure that the furthest away elements are taken from the current processing.

This method tries as much as possible to keep all the available workers busy, hopefully to finish this faster. It works well if different iterations take different amounts of time (work is not balanced).

It uses spawning to divide the work. Can be influenced by the granularity level.

This is the default method for random-access iterators.

enumerator upfront_partition

Partitions the data upfront.

Instead of partitioning the data on the fly lie the auto_partiion method, this will partition the data upfront, creating as many tasks as needed to cover all the workers. This can minimize the task management, but it doesn't necessarily to ensure that all the workers have tasks to do, especially in unbalanced workloads.

Locality is preserved when splitting upfront.

This method only works for random-access iterators.

enumerator iterative_partition

Partitions the iterations as it advances through them.

This partition tries to create a task for each iteration (or, if granularity is > 1, for a number of iterations), and the tasks are created as the algorithm progresses. Locality is not preserved, as nearby elements typically end up on different threads. This method tries to always have tasks to be executed. When a task finished, a new task is spawned.

This method works for forward iterators.

This is the default method for non-random-access iterators.

enumerator naive_partition

Naive partition.

This creates a task for each iteration (or, depending of granularity, on each group of iterations). If there are too many elements in the given range, this can spawn too many tasks.

Does not preserve locality, but is ensures that the tasks are filling up the worker threads in the best possible way.

Enum queue_type

- Defined in file_include_concore_data_concurrent_queue_type.hpp

Enum Documentation

enum `concore::v1::`**`queue_type`**

Queue type, based o the desired level of concurrency for producers and consumers.

Please note that this express only the desired type. It doesn't mean that implementation will be strictly obey the policy. The implementation can be more conservative and fall-back to less optimal implementation. For example, the implementation can always use the multi_prod_multi_cons type, as it includes all the constraints for all the other types.

Values:

enumerator single_prod_single_cons

Single-producer, single-consumer concurrent queue.

enumerator single_prod_multi_cons

Single-producer, multiple-consumer concurrent queue.

enumerator multi_prod_single_cons

Multiple-producer, single-consumer concurrent queue.

enumerator multi_prod_multi_cons

Multiple-producer, multiple-consumer concurrent queue.

enumerator default_type

The default queue type. Multiple-producer, multiple-consumer concurrent queue.

Enum stage_ordering

- Defined in file_include_concore_pipeline.hpp

Enum Documentation

enum `concore::v1::`**`stage_ordering`**

The possible ordering constraints for a stage in a pipeline.

Values:

enumerator in_order

All the items are processed in the order they were started, one at a time.

enumerator out_of_order

Items are processed one at a time, but not necessarily in a specific order.

enumerator concurrent

No constraints; all items can be processed concurrently.

Functions

Template Function concore::conc_sort(It, It, const Comp&, task_group)

- Defined in file_include_concore_conc_sort.hpp

Function Documentation

template<typename **It**, typename **Comp**>
void `concore::`**`conc_sort`**(*It begin*, *It end*, **const** *Comp* &*comp*, task_group *grp*)

Concurrently sort a collection of elements.

Sorts the given collection of elements concurrently. The comparison function must be able to be called in parallel without causing any data races.

Parameters

- `begin`: Iterator pointing to the first element in the collection
- `end`: Iterator pointing to one past the last element in the collection
- `comp`: The comparison functor
- `grp`: Group in which to execute the tasks

Template Parameters

- `It`: The type of the iterator over the collection of elements
- `Comp`: The type of the comparison functor

Template Function concore::conc_sort(It, It, const Comp&)

- Defined in file_include_concore_conc_sort.hpp

Function Documentation

template<typename **It**, typename **Comp**>
void `concore::`**`conc_sort`**(*It begin*, *It end*, **const** *Comp* &*comp*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Template Function concore::conc_sort(It, It)

- Defined in file_include_concore_conc_sort.hpp

Function Documentation

template<typename **It**>
void `concore::`**`conc_sort`**(*It begin*, *It end*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Template Function concore::conc_sort(It, It, task_group)

- Defined in file_include_concore_conc_sort.hpp

Function Documentation

template<typename **It**>
void `concore::`**`conc_sort`**(*It begin*, *It end*, task_group *grp*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Function concore::v1::add_dependencies(chained_task, std::initializer_list<chained_task>)

- Defined in file_include_concore_task_graph.hpp

Function Documentation

void `concore::v1::`**`add_dependencies`**(*chained_task prev*, std::initializer_list<*chained_task*> *nexts*)

Add a dependency from a task to a list of tasks.

This creates dependencies between `prev` and all the tasks in `nexts`. It's like calling *add_dependency()* multiple times.

Parameters

- `prev`: The task dependent on
- `nexts`: A set of tasks that all depend on `prev`

All the tasks in the `nexts` lists will not be started until `prev` is completed.

See *chained_task*, *add_dependency()*

Function concore::v1::add_dependencies(std::initializer_list<chained_task>, chained_task)

- Defined in file_include_concore_task_graph.hpp

Function Documentation

void `concore::v1::`**`add_dependencies`**(std::initializer_list<*chained_task*> *prevs*, *chained_task* *next*)

Add a dependency from list of tasks to a tasks.

This creates dependencies between all the tasks from `prevs` to the `next` task. It's like calling *add_dependency()* multiple times.

Parameters

- `prevs`: The list of tasks that `next` is dependent on
- `next`: The task that depends on all the `prevs` tasks

The `next` tasks will not start until all the tasks from the `prevs` list are complete.

See *chained_task*, *add_dependency()*

Function concore::v1::add_dependency

- Defined in file_include_concore_task_graph.hpp

Function Documentation

void `concore::v1::`**`add_dependency`**(*chained_task* *prev*, *chained_task* *next*)

Add a dependency between two tasks.

This creates a dependency between the given tasks. It means that `next` will only be executed only after `prev` is completed.

Parameters

- `prev`: The task dependent on
- `next`: The task that depends on `prev`

See *chained_task*, *add_dependencies()*

Template Function concore::v1::bulk_execute

- Defined in file_include_concore_execution.hpp

Function Documentation

template<typename **`Executor`**, typename **`Ftor`**, typename **`Num`**>
void `concore::v1::`**`bulk_execute`**(*Executor* &&*e*, *Ftor* &&*f*, *Num* *n*)

Customization point object that can be used to bulk_execute work on executors.

This will tell the executor object to invoke the given functor, according to the rules defined in the executor.

Parameters

- `e`: The executor object we are using for our execution
- `f`: The functor to be invoked
- `n`: The number of times we have to invoke the functor

See *concore::execute()*

Template Function concore::v1::conc_for(It, It, const UnaryFunction&, const task_group&, partition_hints)

- Defined in file_include_concore_conc_for.hpp

Function Documentation

template<typename **It**, typename **UnaryFunction**>
void `concore::v1::`**`conc_for`**(*It first*, *It last*, **const** *UnaryFunction &f*, **const** *task_group &grp*, *partition_hints hints*)

A concurrent `for` algorithm.

If there are no dependencies between the iterations of a for loop, then those iterations can be run in parallel. This function attempts to parallelize these iterations. On a machine that has a very large number of cores, this can execute each iteration on a different core.

Parameters

- `first`: Iterator pointing to the first element in a collection
- `last`: Iterator pointing to the last element in a collection (1 past the end)
- `f`: Functor to apply to each element of the collection
- `grp`: Group in which to execute the tasks
- `hints`: Hints that may be passed to the
- `work`: The work to be applied to be executed for the elements

Template Parameters

- `It`: The type of the iterator to use
- `UnaryFunction`: The type of function to be applied for each element
- `WorkType`: The type of a work object to be used (when not using `f`)

This ensure that the given work/functor is called exactly once for each element from the given sequence. But the call may happen on different threads.

The function does not return until all the iterations are executed. (It may execute other non-related tasks while waiting for the conc_for tasks to complete).

This generates internal tasks by spawning and waiting for those tasks to complete. If the user spawns other tasks during the execution of an iteration, those tasks would also be waited on. This can be a method of generating more work in the concurrent `for` loop.

One can cancel the execution of the tasks by passing a *task_group* in, and canceling that *task_group*.

One can also provide hints to the implementation to fine-tune the algorithms to better fit the data it operates on. Please note however that the implementation may completely ignore all the hints it was provided.

There are two forms of this function: one that uses a functor, and one that takes a work as parameter. The version with the `work` given as argument may be faster in certain cases in which, between iterators, we can store temporary data.

The work structure given to the function must have the following structure:

```
struct GenericWorkType {
    using iterator = my_iterator_type;
    void exec(my_iterator_type first, my_iterator_type last) { ... }
};
```

This work will be called for various chunks from the input. The `iterator` type defined in the given work must support basic random-iterator operations, but without dereference. That is: difference, incrementing, and addition with an integer. The work objects must be copyable.

In the case that no work is given, the algorithm expects either input iterators, or integral types.

Warning If the iterations are not completely independent, this results in undefined behavior.

See *partition_hints*, partition_method, *task_group*

Template Function concore::v1::conc_for(It, It, const UnaryFunction&, const task_group&)

- Defined in file_include_concore_conc_for.hpp

Function Documentation

template<typename **`It`**, typename **`UnaryFunction`**>
void `concore::v1::`**`conc_for`**`(`*It first*, *It last*, **`const`** *UnaryFunction &f*, **`const`** *task_group &grp*`)`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Template Function concore::v1::conc_for(It, It, const UnaryFunction&, partition_hints)

- Defined in file_include_concore_conc_for.hpp

Function Documentation

template<typename **`It`**, typename **`UnaryFunction`**>
void `concore::v1::`**`conc_for`**`(`*It first*, *It last*, **`const`** *UnaryFunction &f*, *partition_hints hints*`)`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Template Function concore::v1::conc_for(It, It, const UnaryFunction&)

- Defined in file_include_concore_conc_for.hpp

Function Documentation

template<typename **`It`**, typename **`UnaryFunction`**>
void `concore::v1::`**`conc_for`**(*It first*, *It last*, **`const`** *UnaryFunction &f*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Template Function concore::v1::conc_for(typename WorkType::iterator, typename WorkType::iterator, WorkType&, const task_group&, partition_hints)

- Defined in file_include_concore_conc_for.hpp

Function Documentation

template<typename **`WorkType`**>
void `concore::v1::`**`conc_for`**(**`typename`** *WorkType*::iterator *first*, **`typename`** *WorkType*::iterator *last*, *WorkType &work*, **`const`** *task_group &grp*, *partition_hints hints*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Template Function concore::v1::conc_for(typename WorkType::iterator, typename WorkType::iterator, WorkType&, const task_group&)

- Defined in file_include_concore_conc_for.hpp

Function Documentation

template<typename **`WorkType`**>
void `concore::v1::`**`conc_for`**(**`typename`** *WorkType*::iterator *first*, **`typename`** *WorkType*::iterator *last*, *WorkType &work*, **`const`** *task_group &grp*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Template Function concore::v1::conc_for(typename WorkType::iterator, typename WorkType::iterator, WorkType&, partition_hints)

- Defined in file_include_concore_conc_for.hpp

Function Documentation

template<typename **`WorkType`**>
void `concore::v1::`**`conc_for`**(**`typename`** *WorkType*::iterator *first*, **`typename`** *WorkType*::iterator *last*, *WorkType &work*, *partition_hints hints*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Template Function concore::v1::conc_for(typename WorkType::iterator, typename WorkType::iterator, WorkType&)

- Defined in file_include_concore_conc_for.hpp

Function Documentation

template<typename **WorkType**>
void `concore::v1::`**`conc_for`**(**`typename`** *WorkType*::iterator *first*, **`typename`** *WorkType*::iterator *last*, *WorkType* &*work*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Template Function concore::v1::conc_reduce(It, It, Value, const BinaryOp&, const ReductionOp&, task_group, partition_hints)

- Defined in file_include_concore_conc_reduce.hpp

Function Documentation

template<typename **It**, typename **Value**, typename **BinaryOp**, typename **ReductionOp**>
Value `concore::v1::`**`conc_reduce`**(*It first*, *It last*, *Value identity*, **`const`** *BinaryOp* &*op*, **`const`** *ReductionOp* &*reduction*, *task_group grp*, *partition_hints hints*)

A concurrent reduce algorithm.

This implements a reduce operation (similar to `std::accumulate`) but does it by exposing concurrent computations. In order for this to be well defined, the given operation and reduction functors must be able to properly run in parallel on different parts of the input collection.

Parameters

- `first`: Iterator pointing to the first element in a collection
- `last`: Iterator pointing to the last element in a collection (1 past the end)
- `identity`: The identity value; the starting point of the reduce
- `op`: Binary functor that is applied to the elements
- `reduction`: Binary functor that reduces the values
- `grp`: Group in which to execute the tasks
- `hints`: Hints that may be passed to the
- `work`: The work to be applied to be executed for the elements (when not using `f`)

Template Parameters

- `It`: The type of the iterator to use
- `Value`: The type of the value used for computation
- `BinaryOp`: The type of function used by the operation that applies to elements
- `ReductionOp`: The type of function used by the reduction operation
- `WorkType`: The type of a work object to be used (when not using `op` and `reduction`)

The operation is called exactly once for each element. The reduction is called each time we need to combine the results of two computations.

This generates internal tasks by spawning and waiting for those tasks to complete. If the user spawns other tasks during the execution of an iteration, those tasks would also be waited on.

One can cancel the execution of the tasks by passing a *task_group* object in, and canceling that *task_group*.

One can also provide hints to the implementation to fine-tune the algorithms to better fit the data it operates on. Please note however that the implementation may completely ignore all the hints it was provided.

Warning If the iterations are not completely independent, this results in undefined behavior.

See *partition_hints*, partition_method, *conc_for()*, *task_group*

Template Function concore::v1::conc_reduce(It, It, Value, const BinaryOp&, const ReductionOp&, task_group)

- Defined in file_include_concore_conc_reduce.hpp

Function Documentation

template<typename **`It`**, typename **`Value`**, typename **`BinaryOp`**, typename **`ReductionOp`**>
Value `concore::v1::`**`conc_reduce`**(*It first*, *It last*, *Value identity*, **`const`** *BinaryOp &op*, **`const`** *ReductionOp &reduction*, *task_group grp*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Template Function concore::v1::conc_reduce(It, It, Value, const BinaryOp&, const ReductionOp&, partition_hints)

- Defined in file_include_concore_conc_reduce.hpp

Function Documentation

template<typename **`It`**, typename **`Value`**, typename **`BinaryOp`**, typename **`ReductionOp`**>
Value `concore::v1::`**`conc_reduce`**(*It first*, *It last*, *Value identity*, **`const`** *BinaryOp &op*, **`const`** *ReductionOp &reduction*, *partition_hints hints*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Template Function concore::v1::conc_reduce(It, It, Value, const BinaryOp&, const ReductionOp&)

- Defined in file_include_concore_conc_reduce.hpp

Function Documentation

template<typename **It**, typename **Value**, typename **BinaryOp**, typename **ReductionOp**>
Value `concore::v1::`**`conc_reduce`**(*It first*, *It last*, *Value identity*, **`const`** *BinaryOp &op*, **`const`** *ReductionOp &reduction*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Template Function concore::v1::conc_reduce(typename WorkType::iterator, typename WorkType::iterator, WorkType&, const task_group&, partition_hints)

- Defined in file_include_concore_conc_reduce.hpp

Function Documentation

template<typename **WorkType**>
void `concore::v1::`**`conc_reduce`**(**`typename`** *WorkType*::iterator *first*, **`typename`** *WorkType*::iterator *last*, *WorkType &work*, **`const`** *task_group &grp*, *partition_hints hints*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Template Function concore::v1::conc_reduce(typename WorkType::iterator, typename WorkType::iterator, WorkType&, const task_group&)

- Defined in file_include_concore_conc_reduce.hpp

Function Documentation

template<typename **WorkType**>
void `concore::v1::`**`conc_reduce`**(**`typename`** *WorkType*::iterator *first*, **`typename`** *WorkType*::iterator *last*, *WorkType &work*, **`const`** *task_group &grp*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Template Function concore::v1::conc_reduce(typename WorkType::iterator, typename WorkType::iterator, WorkType&, partition_hints)

- Defined in file_include_concore_conc_reduce.hpp

Function Documentation

template<typename **WorkType**>
void `concore::v1::`**conc_reduce**(**typename** *WorkType*::iterator *first*, **typename** *WorkType*::iterator *last*, *WorkType* &*work*, *partition_hints* *hints*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Template Function concore::v1::conc_reduce(typename WorkType::iterator, typename WorkType::iterator, WorkType&)

- Defined in file_include_concore_conc_reduce.hpp

Function Documentation

template<typename **WorkType**>
void `concore::v1::`**conc_reduce**(**typename** *WorkType*::iterator *first*, **typename** *WorkType*::iterator *last*, *WorkType* &*work*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Template Function concore::v1::conc_scan(It, It, It2, Value, const BinaryOp&, task_group, partition_hints)

- Defined in file_include_concore_conc_scan.hpp

Function Documentation

template<typename **It**, typename **It2**, typename **Value**, typename **BinaryOp**>
Value `concore::v1::`**conc_scan**(*It* *first*, *It* *last*, *It2* *d_first*, *Value* *identity*, **const** *BinaryOp* &*op*, *task_group* *grp*, *partition_hints* *hints*)

A concurrent scan algorithm.

This implements the prefix sum algorithm. Assuming the given operation is summation, this will write in the destination corresponding to each element, the sum of the previous elements, including itself. Similar to std::inclusive_sum.

Return The result value after applying the operation to the input collection

Parameters

- `first`: Iterator pointing to the first element in the collection
- `last`: Iterator pointing to the last element in the collection (1 past the end)
- `d_first`: Iterator pointing to the first element in the destination collection
- `identity`: The identity element (i.e., 0)
- `op`: The operation to be applied (i.e., summation)
- `grp`: Group in which to execute the tasks
- `hints`: Hints that may be passed to the algorithm

Template Parameters

- `It`: The type of the iterator in the input collection
- `It2`: The type of the output iterator
- `Value`: The type of the values we are operating on
- `BinaryOp`: The type of the binary operation (i.e., summation)

This will try to parallelize the prefix sum algorithm. It will try to create enough task to make all the sums in parallel. In the process of parallelizing, this will create twice as much work as the serial algorithm

One can also provide hints to the implementation to fine-tune the algorithms to better fit the data it operates on. Please note however that the implementation may completely ignore all the hints it was provided.

The operation needs to be able to be called in parallel.

See *conc_for()*, *conc_reduce()*, *task_group*

Template Function concore::v1::conc_scan(It, It, It2, Value, const BinaryOp&, task_group)

- Defined in file_include_concore_conc_scan.hpp

Function Documentation

template<typename **It**, typename **It2**, typename **Value**, typename **BinaryOp**>
Value `concore::v1::`**`conc_scan`**(*It first*, *It last*, *It2 d_first*, *Value identity*, **const** *BinaryOp &op*, *task_group grp*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Template Function concore::v1::conc_scan(It, It, It2, Value, const BinaryOp&, partition_hints)

- Defined in file_include_concore_conc_scan.hpp

Function Documentation

template<typename **It**, typename **It2**, typename **Value**, typename **BinaryOp**>
Value `concore::v1::`**`conc_scan`**(*It first*, *It last*, *It2 d_first*, *Value identity*, **const** *BinaryOp &op*, *partition_hints hints*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Template Function concore::v1::conc_scan(It, It, It2, Value, const BinaryOp&)

- Defined in file_include_concore_conc_scan.hpp

Function Documentation

template<typename **It**, typename **It2**, typename **Value**, typename **BinaryOp**>
Value `concore::v1::`**`conc_scan`**(*It first*, *It last*, *It2 d_first*, *Value identity*, **const** *BinaryOp &op*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Template Function concore::v1::connect

- Defined in file_include_concore_execution.hpp

Function Documentation

template<typename **Sender**, *receiver* **Receiver**>
auto `concore::v1::`**`connect`**(*Sender &&snd*, *Receiver &&rcv*)

Connect a sender with a receiver, returning an async operation object.

The type of the `rcv` parameter must model the *receiver* concept. Usually, the `snd` parameter will model the *sender* and *`sender_to`*`<Receiver>` concepts.

Parameters

- `snd`: The sender object, that triggers the work
- `rcv`: The receiver object that receives the results of the work

The resulting type should model the *operation_state* concept.

Usage example:

```
auto op = concore::connect(snd, rcv);
// later
concore::start(op);
```

See *receiver*, *sender*, *sender_to*, *operation_state*

Template Function concore::v1::execute

- Defined in file_include_concore_execution.hpp

Function Documentation

template<typename **Executor**, typename **Ftor**>
void `concore::v1::`**`execute`**(*Executor &&e*, *Ftor &&f*)

Customization point object that can be used to execute work on executors.

This will tell the executor object to invoke the given functor, according to the rules defined in the executor.

Parameters

- `e`: The executor object we are using for our execution
- `f`: The functor to be invoked

The `Executor` type should model concept *`executor_of`*`<Ftor>`.

See *executor_of*

Function concore::v1::init

- Defined in file_include_concore_init.hpp

Function Documentation

void `concore::v1::`**`init`**`(`**`const`** *init_data &config* = {}`)`

Initializes the concore library.

This will initialize the library, with the given parameters. If the library is already initialized this will throw an *already_initialized* exception.

Parameters

- `config`: The configuration to be passed to the library; optional.

If this is not explicitly called the library will be initialized with default settings the first time that a global task needs to be executed.

See *shutdown()*, *is_initialized()*, *already_initialized*, *init_data*

Function concore::v1::is_initialized

- Defined in file_include_concore_init.hpp

Function Documentation

bool `concore::v1::`**`is_initialized`**`()`

Determines if the library is initialized.

Return True if initialized, False otherwise.

Template Function concore::v1::schedule

- Defined in file_include_concore_execution.hpp

Function Documentation

template<typename **`Scheduler`**>
auto `concore::v1::`**`schedule`**`(`*Scheduler &&sched*`)`

Transforms a scheduler (an execution context) into a single-shot sender.

The return type of the operation must model the *sender* concept.

Parameters

- `sched`: The scheduler object

The availability of this operation for a given type makes the type model the *scheduler* concept.

Usage example:

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

See *scheduler*, *sender*

Template Function concore::v1::set_done

- Defined in file_include_concore_execution.hpp

Function Documentation

template<typename **Receiver**>
void `concore::v1::`**`set_done`**`(`*Receiver* &&*r*`)`

Customization point object that can be used to signal stop to receivers.

This is called by a sender whenever the sender is stopped, and the execution of the task cannot continue. When this is called, *set_value()* is not called anymore.

Parameters

- `r`: The receiver object that is signaled about sender's stop signal

The `Receiver` type should model concept *receiver*.

See *set_value()*, *set_error()*, *receiver*

Template Function concore::v1::set_error

- Defined in file_include_concore_execution.hpp

Function Documentation

template<typename **Receiver**, typename **Err**>
void `concore::v1::`**`set_error`**`(`*Receiver* &&*r*, *Err* &&*e*`)`

Customization point object that can be used to notify receivers of errors.

This is called by a sender whenever the sender has an error to report to the sender. Sending an error means that the sender is done processing; it will not call *set_value()* and *set_done()*.

Parameters

- `r`: The receiver object that is signaled about sender's error
- `e`: The error to be to the receiver

The `Receiver` type should model concept `receiver<E>`.

See *set_value()*, *set_done()*, *receiver*

Template Function concore::v1::set_value

- Defined in file_include_concore_execution.hpp

Function Documentation

template<typename **Receiver**, typename ...**Vs**>
void `concore::v1::`**`set_value`**`(`*Receiver* &&*r*, *Vs*&&... *vs*`)`

Customization point object that can be used to set values to receivers.

This is called by a sender whenever the sender has finished work and produces some values. This can be called even if the sender doesn't have any values to send to the receiver.

Parameters

- `r`: The receiver object that is signaled about sender's success
- `vs`: The values sent by the sender

The `Receiver` type should model concepts *receiver* and `receiver_of<Vs...>`.

See *set_done()*, *set_error()*, *receiver*, *receiver_of*

Function concore::v1::shutdown

- Defined in file_include_concore_init.hpp

Function Documentation

void `concore::v1::`**`shutdown`**`()`

Shuts down the concore library.

This can be called at the end of the program to stop the worker threads and cleanup the enqueued tasks. In general, the library is destroyed automatically at the end, so this is not necessarily needed. However, we might want to call this in unit tests to ensure that the library is in a clean state for the next test.

Warning It is forbidden to shutdown the library while it's still in use. All tasks need to be finished executing, and no other objects/functions are accessed anymore.

Warning After shutdown, any call to the library except an initialization-specific call leads to undefined behavior.

Function concore::v1::spawn(task&&, bool)

- Defined in file_include_concore_spawn.hpp

Function Documentation

void `concore::v1::`**`spawn`**(*task* &&*t*, bool *wake_workers* = true)

Spawns a task, hopefully in the current working thread.

This is intended to be called from within a task. In this case, the task will be added to the list of tasks for the current worker thread. The tasks will be added in the front of the list, so it will be executed in front of others.

Parameters

- `t`: The task to be spawned
- `wake_workers`: True if we should wake other workers for this task

The add-to-front strategy aims as improving locality of execution. We assume that this task is closer to the current task than other tasks in the system.

If the current running task does not finish execution after spawning this new task, it's advised for the `wake_workers` parameter to be set to `true`. If, on the other hand, the current task finishes execution after this, it may be best to not set `wake_workers` to `false` and thus try to wake other threads. Waking up other threads can be an efficiency loss that we don't need if we know that this thread is finishing soon executing the current task.

Note that the given task ca take a *task_group* at construction. This way, the users can control the groups of the spawned tasks.

Template Function concore::v1::spawn(F&&, bool)

- Defined in file_include_concore_spawn.hpp

Function Documentation

template<typename **`F`**>
void `concore::v1::`**`spawn`**(*F* &&*ftor*, bool *wake_workers* = true)

Spawn one task, given a functor to be executed.

This is similar to the spawn(task&&, bool) function, but it takes directly a functor instead of a task.

Parameters

- `ftor`: The ftor to be executed
- `wake_workers`: True if we should wake other workers for this task

Template Parameters

- `F`: The type of the functor

If the current task has a group associated, the new task will inherit that group.

See spawn(task&& t, bool wake_workers)

Template Function concore::v1::spawn(F&&, task_group, bool)

- Defined in file_include_concore_spawn.hpp

Function Documentation

template<typename **F**>
void `concore::v1::`**`spawn`**(*F* &&*ftor*, *task_group* *grp*, bool *wake_workers* = true)

Spawn one task, given a functor to be executed.

This is similar to the spawn(task&&, bool) function, but it takes directly a functor and a task group instead of a task.

Parameters

- `ftor`: The ftor to be executed
- `grp`: The group in which the task should be executed.
- `wake_workers`: True if we should wake other workers for this task

Template Parameters

- `F`: The type of the functor

If the current task has a group associated, the new task will inherit that group.

See spawn(task&&, bool)

Function concore::v1::spawn(std::initializer_list<task_function>&&, bool)

- Defined in file_include_concore_spawn.hpp

Function Documentation

void `concore::v1::`**`spawn`**(std::initializer_list<*task_function*> &&*ftors*, bool *wake_workers* = true)

Spawn multiple tasks, given the functors to be executed.

This is similar to the other two *spawn()* functions, but it takes a series of functions to be executed. Tasks will be created for all these functions and spawn accordingly.

Parameters

- `ftors`: A list of functors to be executed
- `wake_workers`: True if we should wake other workers for the last task

The `wake_workers` will control whether to wake threads for the last task or not. For the others tasks, it is assumed that we always want to wake other workers to attempt to get as many tasks as possible from the current worker task list.

If the current task has a task group associated, all the newly created tasks will inherit that group.

spawn(task&&, bool), *spawn_and_wait()*

Function concore::v1::spawn(std::initializer_list<task_function>&&, task_group, bool)

- Defined in file_include_concore_spawn.hpp

Function Documentation

void `concore::v1::`**`spawn`**(std::initializer_list<*task_function*> &&*ftors*, *task_group* *grp*, bool *wake_workers* = true)

Spawn multiple tasks, given the functors to be executed.

This is similar to the other two *spawn()* functions, but it takes a series of functions to be executed. Tasks will be created for all these functions and spawn accordingly.

Parameters

- `ftors`: A list of functors to be executed
- `grp`: The group in which the functors are to be executed
- `wake_workers`: True if we should wake other workers for the last task

The `wake_workers` will control whether to wake threads for the last task or not. For the others tasks, it is assumed that we always want to wake other workers to attempt to get as many tasks as possible from the current worker task list.

spawn(task&&, bool), *spawn_and_wait()*

Template Function concore::v1::spawn_and_wait(F&&)

- Defined in file_include_concore_spawn.hpp

Function Documentation

template<typename **F**>
void `concore::v1::`**`spawn_and_wait`**(*F* &&*ftor*)

Spawn a task and wait for it.

This function is similar to the *spawn()* functions, but, after spawning, also waits for the spawned task to complete. This wait is an active-wait, as it tries to execute other tasks. In principle, the current thread executes the spawn task.

Parameters

- `ftor`: The functor of the tasks to be spawned

Template Parameters

- `F`: The type of the functor.

This will create a new task group, inheriting from the task group of the currently executing task and add the new task in this new group. The waiting is done on this new group.

See *spawn()*

Function concore::v1::spawn_and_wait(std::initializer_list<task_function>&&, bool)

- Defined in file_include_concore_spawn.hpp

Function Documentation

void `concore::v1::`**`spawn_and_wait`**(std::initializer_list<*task_function*> &&*ftors*, bool *wake_workers* = true)

Spawn multiple tasks and wait for them to complete.

This is used when a task needs multiple things done in parallel.

Parameters

- `ftors`: A list of functors to be executed
- `wake_workers`: True if we should wake other workers for the last task

This function is similar to the *spawn()* functions, but, after spawning, also waits for the spawned tasks to complete. This wait is an active-wait, as it tries to execute other tasks. In principle, the current thread executes the last of the spawned tasks.

This will create a new task group, inheriting from the task group of the currently executing task and add the new tasks in this new group. The waiting is done on this new group.

Template Function concore::v1::start

- Defined in file_include_concore_execution.hpp

Function Documentation

template<*operation_state* **Oper**>
void `concore::v1::`**`start`**(*Oper* &&*o*)

Customization point object that can be used to start asynchronous operations.

This is called whenever one needs to start an asynchronous operation.

Parameters

- `o`: The operation that should be started

The `Oper` type must model concept *operation_state*.

See *operation_state*

Template Function concore::v1::submit

- Defined in file_include_concore_execution.hpp

Function Documentation

template<*sender* **Sender**, *receiver* **Receiver**>
void `concore::v1::`**`submit`**(*Sender* &&*snd*, *Receiver* &&*rcv*)

Submit work from a sender, by combining it with a receiver.

The *`sender_to`*`<Sender, Receiver>` concept must hold.

Parameters

- `snd`: The sender object, that triggers the work
- `rcv`: The receiver object that receives the results of the work

If there is no *submit()* customization point defined for the given `Sender` object (taking a `Receiver` object), then this will fall back to calling *concore::connect()*.

Usage example:

```
submit(snd, rcv);
```

See *connect()*

Function concore::v1::wait

- Defined in file_include_concore_spawn.hpp

Function Documentation

void `concore::v1::`**`wait`**(*task_group* &*grp*)

Wait on all the tasks in the given group to finish executing.

The wait here is an active-wait. This will execute tasks from the task system in the hope that the tasks in the group are executed faster.

Parameters

- `grp`: The task group to wait on

Using this inside active tasks is not going to block the worker thread and thus not degrade performance.

Warning If one adds task in a group and never executes them, this function will block indefinitely.

See *spawn()*, *spawn_and_wait()*

Variables

Variable concore::v1::pipeline_end

- Defined in file_include_concore_pipeline.hpp

Variable Documentation

constexpr auto `concore::v1::`**`pipeline_end`** = *pipeline_end_t*{}

Tag value to help end the expression of building a pipeline.

Defines

Define CONCORE_LOW_LEVEL_SHORT_PAUSE

- Defined in file_include_concore_low_level_spin_backoff.hpp

Define Documentation

`CONCORE_LOW_LEVEL_SHORT_PAUSE`()

Pauses the CPU for a short while.

The intent of this macro is to pause the CPU, without consuming energy, while waiting for some other condition to happen. The pause should be sufficiently small so that the current thread will not give up its work quanta.

This pause should be smaller than the pause caused by *CONCORE_LOW_LEVEL_YIELD_PAUSE*.

This is used in *spin* implementations that are waiting for certain conditions to happen, and it is expected that these condition will become true in a very short amount of time.

The implementation of this uses platform-specific instructions.

See *CONCORE_LOW_LEVEL_YIELD_PAUSE*, *concore::v1::spin_backoff*

Define CONCORE_LOW_LEVEL_YIELD_PAUSE

- Defined in file_include_concore_low_level_spin_backoff.hpp

Define Documentation

`CONCORE_LOW_LEVEL_YIELD_PAUSE`()

Pause that will make the current thread yield its CPU quanta.

This is intended to be a longer pause than *CONCORE_LOW_LEVEL_SHORT_PAUSE()*. It is used in *spin* algorithms that wait for some condition to become true, but apparently that condition does not become true soon enough. Instead of blocking the CPU waiting on this condition, we give up the CPU quanta to be used by other threads; hopefully, by running other threads, that condition can become true.

See *CONCORE_LOW_LEVEL_SHORT_PAUSE*, *concore::v1::spin_backoff*

Typedefs

Typedef concore::except_fun_t

- Defined in file_include_concore_except_fun_type.hpp

Typedef Documentation

`using` `concore::`**`except_fun_t`** = std::function<void (std::exception_ptr) >

Type of function to be called for handling exceptions.

This defines the type of exception handler function used across concore. A handler of this type will be called whenever an exception is thrown.

Typedef concore::v1::task_function

- Defined in file_include_concore_task.hpp

Typedef Documentation

`using` `concore::v1::`**`task_function`** = std::function<void () >

A function type that is compatible with a task.

This function takes no arguments and returns nothing. It represents generic *work*.

A concore *task* is essentially a wrapper over a `task_function`.

See *task*

INDEX

C

`concore::conc_sort` (*C++ function*), 82, 83
`concore::except_fun_t` (*C++ type*), 103
`concore::v1::add_dependencies` (*C++ function*), 83, 84
`concore::v1::add_dependency` (*C++ function*), 84
`concore::v1::already_initialized` (*C++ struct*), 21
`concore::v1::already_initialized::already_initialized` (*C++ function*), 22
`concore::v1::any_executor` (*C++ class*), 45
`concore::v1::any_executor::~any_executor` (*C++ function*), 46
`concore::v1::any_executor::any_executor` (*C++ function*), 46
`concore::v1::any_executor::execute` (*C++ function*), 46, 47
`concore::v1::any_executor::operator bool` (*C++ function*), 47
`concore::v1::any_executor::operator!=` (*C++ function*), 47
`concore::v1::any_executor::operator=` (*C++ function*), 46
`concore::v1::any_executor::operator==` (*C++ function*), 47
`concore::v1::any_executor::swap` (*C++ function*), 46
`concore::v1::any_executor::target` (*C++ function*), 47
`concore::v1::any_executor::target_type` (*C++ function*), 47
`concore::v1::as_invocable` (*C++ struct*), 22
`concore::v1::as_invocable::~as_invocable` (*C++ function*), 22
`concore::v1::as_invocable::as_invocable` (*C++ function*), 22
`concore::v1::as_invocable::operator()` (*C++ function*), 22
`concore::v1::as_invocable::operator=` (*C++ function*), 22
`concore::v1::as_operation` (*C++ struct*), 23
`concore::v1::as_operation::as_operation` (*C++ function*), 23
`concore::v1::as_operation::executor_type` (*C++ type*), 23
`concore::v1::as_operation::receiver_type` (*C++ type*), 23
`concore::v1::as_operation::start` (*C++ function*), 23
`concore::v1::as_receiver` (*C++ struct*), 23
`concore::v1::as_receiver::as_receiver` (*C++ function*), 24
`concore::v1::as_receiver::set_done` (*C++ function*), 24
`concore::v1::as_receiver::set_error` (*C++ function*), 24
`concore::v1::as_receiver::set_value` (*C++ function*), 24
`concore::v1::as_sender` (*C++ struct*), 24
`concore::v1::as_sender::as_sender` (*C++ function*), 25
`concore::v1::as_sender::connect` (*C++ function*), 25
`concore::v1::as_sender::error_types` (*C++ type*), 25
`concore::v1::as_sender::sends_done` (*C++ member*), 25
`concore::v1::as_sender::value_types` (*C++ type*), 25
`concore::v1::binary_semaphore` (*C++ class*), 48
`concore::v1::binary_semaphore::~binary_semaphore` (*C++ function*), 48
`concore::v1::binary_semaphore::binary_semaphore` (*C++ function*), 48
`concore::v1::binary_semaphore::operator=` (*C++ function*), 48
`concore::v1::binary_semaphore::signal` (*C++ function*), 48
`concore::v1::binary_semaphore::wait` (*C++ function*), 48
`concore::v1::bulk_execute` (*C++ function*), 84
`concore::v1::bulk_execute_t` (*C++ struct*),

25
`concore::v1::chained_task` (*C++ class*), 49
`concore::v1::chained_task::chained_task` (*C++ function*), 49
`concore::v1::chained_task::clear_next` (*C++ function*), 50
`concore::v1::chained_task::operator bool` (*C++ function*), 50
`concore::v1::chained_task::operator()` (*C++ function*), 50
`concore::v1::chained_task::set_exception_handler` (*C++ function*), 50
`concore::v1::conc_for` (*C++ function*), 85–88
`concore::v1::conc_reduce` (*C++ function*), 88–91
`concore::v1::conc_scan` (*C++ function*), 91–93
`concore::v1::concurrent_dequeue` (*C++ class*), 50
`concore::v1::concurrent_dequeue::concurrent_dequeue` (*C++ function*), 51
`concore::v1::concurrent_dequeue::push_back` (*C++ function*), 51
`concore::v1::concurrent_dequeue::push_front` (*C++ function*), 51
`concore::v1::concurrent_dequeue::try_pop_back` (*C++ function*), 51
`concore::v1::concurrent_dequeue::try_pop_front` (*C++ function*), 51
`concore::v1::concurrent_dequeue::unsafe_clear` (*C++ function*), 51
`concore::v1::concurrent_dequeue::value_type` (*C++ type*), 51
`concore::v1::concurrent_queue` (*C++ class*), 52
`concore::v1::concurrent_queue::~concurrent_queue` (*C++ function*), 52
`concore::v1::concurrent_queue::concurrent_queue` (*C++ function*), 52
`concore::v1::concurrent_queue::operator=` (*C++ function*), 52
`concore::v1::concurrent_queue::push` (*C++ function*), 52
`concore::v1::concurrent_queue::try_pop` (*C++ function*), 53
`concore::v1::concurrent_queue::value_type` (*C++ type*), 52
`concore::v1::connect` (*C++ function*), 93
`concore::v1::connect_t` (*C++ struct*), 26
`concore::v1::delegating_executor` (*C++ struct*), 26
`concore::v1::delegating_executor::delegating_executor` (*C++ function*), 26
`concore::v1::delegating_executor::execute` (*C++ function*), 26
`concore::v1::delegating_executor::fun_type` (*C++ type*), 26
`concore::v1::delegating_executor::operator!=` (*C++ function*), 27
`concore::v1::delegating_executor::operator==` (*C++ function*), 27
`concore::v1::dispatch_executor` (*C++ struct*), 27
`concore::v1::dispatch_executor::dispatch_executor` (*C++ function*), 27
`concore::v1::dispatch_executor::execute` (*C++ function*), 27
`concore::v1::dispatch_executor::operator!=` (*C++ function*), 28
`concore::v1::dispatch_executor::operator==` (*C++ function*), 28
`concore::v1::dispatch_executor::priority` (*C++ enum*), 27
`concore::v1::dispatch_executor::priority::prio_high` (*C++ enumerator*), 27
`concore::v1::dispatch_executor::priority::prio_low` (*C++ enumerator*), 27
`concore::v1::dispatch_executor::priority::prio_norm` (*C++ enumerator*), 27
`concore::v1::execute` (*C++ function*), 93
`concore::v1::execute_t` (*C++ struct*), 28
`concore::v1::executor` (*C++ struct*), 28
`concore::v1::executor_of` (*C++ struct*), 29
`concore::v1::finish_event` (*C++ struct*), 29
`concore::v1::finish_event::notify_done` (*C++ function*), 30
`concore::v1::finish_task` (*C++ struct*), 30
`concore::v1::finish_task::event` (*C++ function*), 31
`concore::v1::finish_task::finish_task` (*C++ function*), 31
`concore::v1::finish_wait` (*C++ struct*), 31
`concore::v1::finish_wait::event` (*C++ function*), 32
`concore::v1::finish_wait::finish_wait` (*C++ function*), 32
`concore::v1::finish_wait::wait` (*C++ function*), 32
`concore::v1::global_executor` (*C++ struct*), 32
`concore::v1::global_executor::execute` (*C++ function*), 33
`concore::v1::global_executor::global_executor` (*C++ function*), 33
`concore::v1::global_executor::operator!=` (*C++ function*), 33
`concore::v1::global_executor::operator==` (*C++ function*), 33
`concore::v1::global_executor::prio_background`

(*C++ member*), 33
concore::v1::global_executor::prio_critical (*C++ member*), 33
concore::v1::global_executor::prio_high (*C++ member*), 33
concore::v1::global_executor::prio_low (*C++ member*), 33
concore::v1::global_executor::prio_normal (*C++ member*), 33
concore::v1::global_executor::priority (*C++ type*), 32
concore::v1::init (*C++ function*), 94
concore::v1::init_data (*C++ struct*), 33
concore::v1::init_data::num_workers_ (*C++ member*), 34
concore::v1::init_data::reserved_slots_ (*C++ member*), 34
concore::v1::init_data::worker_start_fun (*C++ member*), 34
concore::v1::inline_executor (*C++ struct*), 34
concore::v1::inline_executor::execute (*C++ function*), 34
concore::v1::inline_executor::operator!= (*C++ function*), 34
concore::v1::inline_executor::operator== (*C++ function*), 34
concore::v1::invocable_archetype (*C++ struct*), 35
concore::v1::invocable_archetype::operator() (*C++ function*), 35
concore::v1::is_initialized (*C++ function*), 94
concore::v1::n_serializer (*C++ class*), 53
concore::v1::n_serializer::execute (*C++ function*), 54
concore::v1::n_serializer::n_serializer (*C++ function*), 54
concore::v1::n_serializer::operator!= (*C++ function*), 55
concore::v1::n_serializer::operator== (*C++ function*), 55
concore::v1::n_serializer::set_exception_handler (*C++ function*), 54
concore::v1::operation_state (*C++ struct*), 35
concore::v1::partition_hints (*C++ struct*), 35
concore::v1::partition_hints::granularity_ (*C++ member*), 36
concore::v1::partition_hints::method_ (*C++ member*), 36
concore::v1::partition_hints::tasks_per_worker_ (*C++ member*), 36
concore::v1::partition_method (*C++ enum*), 80
concore::v1::partition_method::auto_partition (*C++ enumerator*), 80
concore::v1::partition_method::iterative_partition (*C++ enumerator*), 80
concore::v1::partition_method::naive_partition (*C++ enumerator*), 80
concore::v1::partition_method::upfront_partition (*C++ enumerator*), 80
concore::v1::pipeline (*C++ class*), 55
concore::v1::pipeline::push (*C++ function*), 56
concore::v1::pipeline_builder (*C++ class*), 56
concore::v1::pipeline_builder::add_stage (*C++ function*), 57
concore::v1::pipeline_builder::build (*C++ function*), 58
concore::v1::pipeline_builder::operator pipeline<T> (*C++ function*), 58
concore::v1::pipeline_builder::operator| (*C++ function*), 58
concore::v1::pipeline_builder::pipeline_builder (*C++ function*), 57
concore::v1::pipeline_end (*C++ member*), 102
concore::v1::pipeline_end_t (*C++ struct*), 36
concore::v1::queue_type (*C++ enum*), 81
concore::v1::queue_type::default_type (*C++ enumerator*), 81
concore::v1::queue_type::multi_prod_multi_cons (*C++ enumerator*), 81
concore::v1::queue_type::multi_prod_single_cons (*C++ enumerator*), 81
concore::v1::queue_type::single_prod_multi_cons (*C++ enumerator*), 81
concore::v1::queue_type::single_prod_single_cons (*C++ enumerator*), 81
concore::v1::receiver (*C++ struct*), 36
concore::v1::receiver_invocation_error (*C++ struct*), 37
concore::v1::receiver_invocation_error::receiver_in (*C++ function*), 37
concore::v1::receiver_of (*C++ struct*), 38
concore::v1::rw_serializer (*C++ class*), 59
concore::v1::rw_serializer::reader (*C++ function*), 60
concore::v1::rw_serializer::reader_type (*C++ class*), 60, 62
concore::v1::rw_serializer::reader_type::execute (*C++ function*), 60, 62
concore::v1::rw_serializer::reader_type::operator!=

(*C++ function*), 61, 62
concore::v1::rw_serializer::reader_type::operator==
(*C++ function*), 61, 62
concore::v1::rw_serializer::reader_type::reader_type
(*C++ function*), 60, 62
concore::v1::rw_serializer::rw_serializer
(*C++ function*), 59
concore::v1::rw_serializer::set_exception_handler
(*C++ function*), 60
concore::v1::rw_serializer::writer (*C++ function*), 60
concore::v1::rw_serializer::writer_type
(*C++ class*), 61, 63
concore::v1::rw_serializer::writer_type::execute
(*C++ function*), 61, 63
concore::v1::rw_serializer::writer_type::operator!=
(*C++ function*), 61, 63
concore::v1::rw_serializer::writer_type::operator()
(*C++ function*), 61, 63
concore::v1::rw_serializer::writer_type::operator==
(*C++ function*), 61, 63
concore::v1::rw_serializer::writer_type::writer_type
(*C++ function*), 61, 63
concore::v1::schedule (*C++ function*), 94
concore::v1::schedule_t (*C++ struct*), 39
concore::v1::scheduler (*C++ struct*), 39
concore::v1::semaphore (*C++ class*), 64
concore::v1::semaphore::~semaphore (*C++ function*), 64
concore::v1::semaphore::operator= (*C++ function*), 64
concore::v1::semaphore::semaphore (*C++ function*), 64
concore::v1::semaphore::signal (*C++ function*), 64
concore::v1::semaphore::wait (*C++ function*), 64
concore::v1::sender (*C++ struct*), 39
concore::v1::sender_to (*C++ struct*), 40
concore::v1::serializer (*C++ class*), 65
concore::v1::serializer::execute (*C++ function*), 66
concore::v1::serializer::operator!=
(*C++ function*), 66
concore::v1::serializer::operator==
(*C++ function*), 66
concore::v1::serializer::serializer
(*C++ function*), 65
concore::v1::serializer::set_exception_handler
(*C++ function*), 66
concore::v1::set_done (*C++ function*), 95
concore::v1::set_done_t (*C++ struct*), 40
concore::v1::set_error (*C++ function*), 95
concore::v1::set_error_t (*C++ struct*), 41
concore::v1::set_value (*C++ function*), 96
concore::v1::set_value_t (*C++ struct*), 41
concore::v1::shared_spin_mutex (*C++ class*), 66
concore::v1::shared_spin_mutex::~shared_spin_mutex
(*C++ function*), 67
concore::v1::shared_spin_mutex::lock (*C++ function*), 67
concore::v1::shared_spin_mutex::lock_shared
(*C++ function*), 68
concore::v1::shared_spin_mutex::operator=
(*C++ function*), 67
concore::v1::shared_spin_mutex::shared_spin_mutex (*C++ function*), 67
concore::v1::shared_spin_mutex::try_lock (*C++ function*), 67
concore::v1::shared_spin_mutex::try_lock_shared (*C++ function*), 68
concore::v1::shared_spin_mutex::unlock (*C++ function*), 68
concore::v1::shared_spin_mutex::unlock_shared (*C++ function*), 68
concore::v1::shutdown (*C++ function*), 96
concore::v1::spawn (*C++ function*), 97–99
concore::v1::spawn_and_wait (*C++ function*), 99, 100
concore::v1::spawn_continuation_executor
(*C++ struct*), 42
concore::v1::spawn_continuation_executor::execute
(*C++ function*), 42
concore::v1::spawn_continuation_executor::operator
(*C++ function*), 42
concore::v1::spawn_continuation_executor::operator=
(*C++ function*), 42
concore::v1::spawn_executor (*C++ struct*), 42
concore::v1::spawn_executor::execute
(*C++ function*), 43
concore::v1::spawn_executor::operator!=
(*C++ function*), 43
concore::v1::spawn_executor::operator==
(*C++ function*), 43
concore::v1::spin_backoff (*C++ class*), 69
concore::v1::spin_backoff::pause (*C++ function*), 69
concore::v1::spin_mutex (*C++ class*), 69
concore::v1::spin_mutex::~spin_mutex
(*C++ function*), 70
concore::v1::spin_mutex::lock (*C++ function*), 70
concore::v1::spin_mutex::operator= (*C++ function*), 70
concore::v1::spin_mutex::spin_mutex
(*C++ function*), 70

concore::v1::spin_mutex::try_lock (*C++ function*), 70
concore::v1::spin_mutex::unlock (*C++ function*), 70
concore::v1::stage_ordering (*C++ enum*), 81
concore::v1::stage_ordering::concurrent (*C++ enumerator*), 81
concore::v1::stage_ordering::in_order (*C++ enumerator*), 81
concore::v1::stage_ordering::out_of_order (*C++ enumerator*), 81
concore::v1::start (*C++ function*), 100
concore::v1::start_t (*C++ struct*), 43
concore::v1::static_thread_pool (*C++ class*), 71
concore::v1::static_thread_pool::~static_thread_pool (*C++ function*), 72
concore::v1::static_thread_pool::attach (*C++ function*), 72
concore::v1::static_thread_pool::executor (*C++ function*), 73
concore::v1::static_thread_pool::executor_type (*C++ type*), 71
concore::v1::static_thread_pool::join (*C++ function*), 72
concore::v1::static_thread_pool::operator= (*C++ function*), 72
concore::v1::static_thread_pool::scheduler (*C++ function*), 72
concore::v1::static_thread_pool::scheduler_type (*C++ type*), 71
concore::v1::static_thread_pool::static_thread_pool (*C++ function*), 71, 72
concore::v1::static_thread_pool::stop (*C++ function*), 72
concore::v1::static_thread_pool::wait (*C++ function*), 72
concore::v1::submit (*C++ function*), 101
concore::v1::submit_t (*C++ struct*), 44
concore::v1::task (*C++ class*), 73
concore::v1::task::~task (*C++ function*), 75
concore::v1::task::get_task_group (*C++ function*), 76
concore::v1::task::operator bool (*C++ function*), 75
concore::v1::task::operator() (*C++ function*), 75
concore::v1::task::operator= (*C++ function*), 75
concore::v1::task::swap (*C++ function*), 75
concore::v1::task::task (*C++ function*), 74, 75
concore::v1::task_function (*C++ type*), 103
concore::v1::task_group (*C++ class*), 76
concore::v1::task_group::~task_group (*C++ function*), 77
concore::v1::task_group::cancel (*C++ function*), 77
concore::v1::task_group::clear_cancel (*C++ function*), 78
concore::v1::task_group::create (*C++ function*), 78
concore::v1::task_group::current_task_group (*C++ function*), 79
concore::v1::task_group::is_active (*C++ function*), 78
concore::v1::task_group::is_cancelled (*C++ function*), 78
concore::v1::task_group::is_current_task_cancelled (*C++ function*), 79
concore::v1::task_group::operator bool (*C++ function*), 77
concore::v1::task_group::operator= (*C++ function*), 77
concore::v1::task_group::set_current_task_group (*C++ function*), 79
concore::v1::task_group::set_exception_handler (*C++ function*), 77
concore::v1::task_group::task_group (*C++ function*), 77
concore::v1::tbb_executor (*C++ struct*), 44
concore::v1::tbb_executor::execute (*C++ function*), 45
concore::v1::tbb_executor::operator!= (*C++ function*), 45
concore::v1::tbb_executor::operator== (*C++ function*), 45
concore::v1::tbb_executor::priority (*C++ enum*), 44
concore::v1::tbb_executor::priority::prio_high (*C++ enumerator*), 44
concore::v1::tbb_executor::priority::prio_low (*C++ enumerator*), 44
concore::v1::tbb_executor::priority::prio_normal (*C++ enumerator*), 44
concore::v1::tbb_executor::tbb_executor (*C++ function*), 45
concore::v1::typed_sender (*C++ struct*), 45
concore::v1::wait (*C++ function*), 101
CONCORE_LOW_LEVEL_SHORT_PAUSE (*C macro*), 102
CONCORE_LOW_LEVEL_YIELD_PAUSE (*C macro*), 102