concore

Lucian Radu Teodorescu

Apr 19, 2020

CONTENTS

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

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

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

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

concore concurrency core

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

CHAPTER

ONE

TABLE OF CONTENT

1.1 Quick-start

1.1.1 Building the library

The following tools are needed:

- conan
- CMake

Perform the following actions:

```
mkdir -p build
pushd build

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

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

popd build
```

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

1.1.2 Tutorial

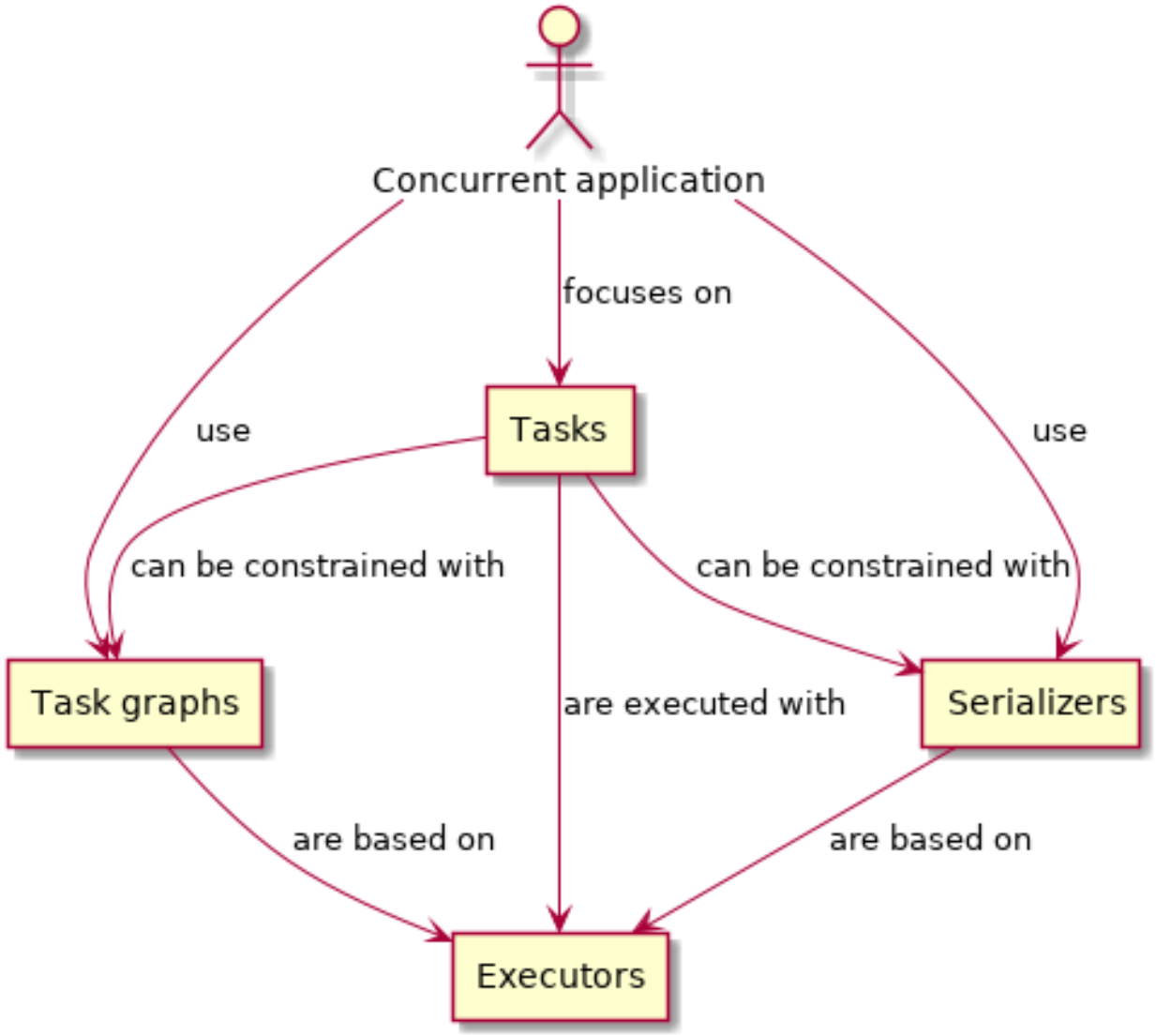
TODO

1.1.3 Examples

TODO

1.2 Concepts

Overview of main concepts:



1.2.1 Building concurrent applications with concore

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

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

concore aims at alleviating these issues by implementing a *Task-Oriented Design* model for expressing concurrent applications. Instead of focusing on manual creation of threads and solving synchronization issues, the user should

focus on decomposing the application into smaller units of work that can be executed in parallel. If the decomposition is done correctly, the synchronization problems will disappear. Also, assuming there is enough work, the performance of the application can be close-to-optimal (considering throughput). Understandability is also improved as the concurrency is directly visible at the design level.

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

Proper design should have two main aspects in mind:

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

concore have tools to help with both of these aspects.

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

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

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

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

1.2.2 Tasks

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

task An independent unit of work

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

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

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

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

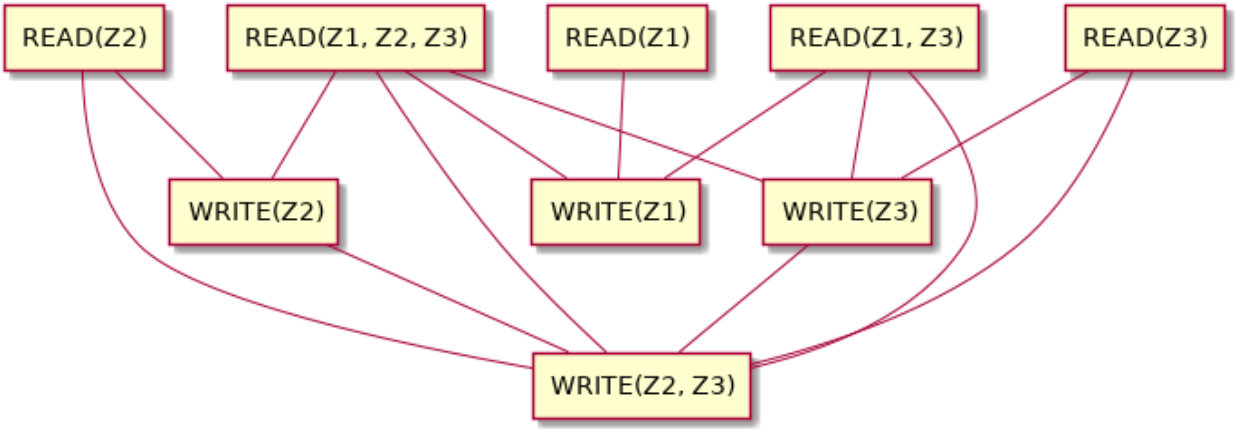
Let us take an example. Let's assume that we have an application with a central data storage. This central data storage has 3 zones with information that can be read or written: *Z1*, *Z2* and *Z3*. One can have tasks that read data, tasks that write data, and tasks that both read and write data. These operations can be specific to a zone or multiple zones in the

central data storage. We want to create a task for each of these operations. Then, at design time, we want to impose the following constraints:

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

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

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



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

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

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

1.2.3 Executors

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

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

Concore has defined the following executors:

- *global_executor*
- *global_executor_critical_prio*
- *global_executor_high_prio*
- *global_executor_normal_prio*
- *global_executor_low_prio*
- *global_executor_background_prio*

- *spawn_executor*
- *spawn_continuation_executor*
- *immediate_executor*
- *dispatch_executor*
- *tbb_executor*

An executor can always be stored into a *executor_t* (which is an alias for `std::function<void(task)>`). In other words, an executor can be thought of as objects that consume tasks.

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.

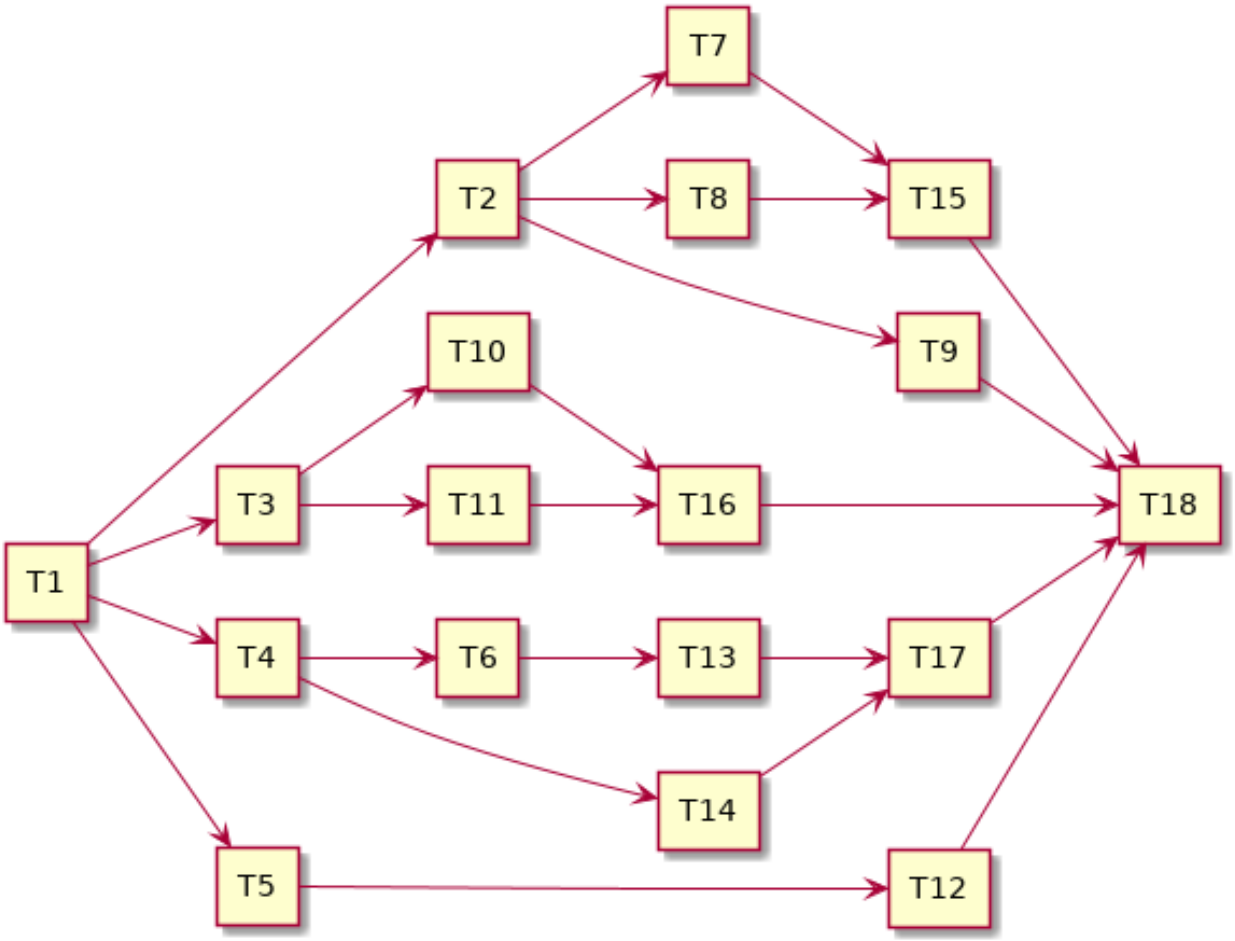
Another popular alternative is to use the *spawn* functionality (either as a free function *spawn()*, or through *spawn_executor*). This should be called from within the execution of a task and will add the given task to the local queue of the current worker thread; the thread will try to pick up the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

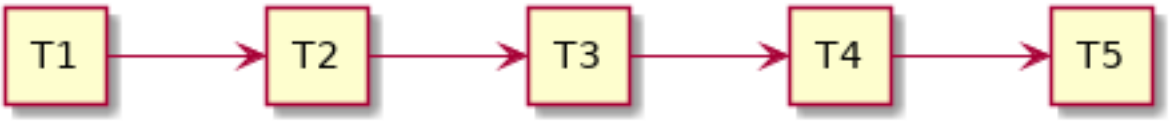
Another way of building task graph is to use concore's abstractions. The nodes in the graph can be modeled with *chained_task* objects. Dependencies can be calling *add_dependency()* of *add_dependencies()* functions.

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.

1.3 API reference

1.3.1 Tasks

task.hpp

namespace concore

namespace v1

Typedefs

using 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*

class task
#include <task.hpp> 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 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.

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(*task*&&)
Move constructor.

task &**operator=**(*task*&&)
Move operator.

task(**const** *task*&)
Copy constructor is DISABLED.

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

task_group &**get_task_group**()
Gets the task group.

This allows the users to consult the task group associated with the task and change it.
Return The task group, as a reference.

Private Members

task_function **fun_**
The function to be called.

This can be associated with a task through construction and by using the special assignment operator.

Please note that, as the tasks cannot be copied and shared, and as the task system prefers moving tasks, after the task is enqueued this is constant.

task_group **task_group_**
The group that this tasks belongs to.

This can be set by the constructor, or can be set by calling *get_task_group()*. As the library prefers passing tasks around by moving them, after the task was enqueued, the task group cannot be changed.

task_group.hpp

namespace concore

namespace v1

class task_group
#include <task_group.hpp> 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.

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*&)
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*&&)
Move constructor; rarely used.

task_group &**operator=**(**const** *task_group*&)
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*&&)
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**(std::function<void) std::exception_ptr
> *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 tasks or other *task_group* objects in this group.

This can be used to check when all tasks from a group are completed. Completed tasks are not stored by the task system, nor by the executors, so they release the reference to this object. This function returns `true` if no tasks refer to this group.
Return True if active, False otherwise.
For a newly created *task_group*, this will return `false`, as there are no tasks in it.

- TODO: Don't count subgroups
- TODO: make it work hierarchically.

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()*

Private Members

std::shared_ptr<detail::task_group_impl> **impl_**
Implementation detail of a task group object. Note that the implementation details can be shared between multiple *task_group* objects.

spawn.hpp

namespace concore

namespace v1

Functions

void **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<typename **F**>
void **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&&, bool)

void **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()

template<typename **F**>
void **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()

void **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.

void **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

constexpr auto **spawn_executor** = detail::spawn_executor{}
Executor that spawns tasks instead of enqueueing them. Similar to calling spawn() on the task.

See spawn(), spawn_continuation_executor, global_executor

constexpr auto **spawn_continuation_executor** = detail::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_executor, global_executor

task_graph.hpp

namespace concore

namespace v1

Functions

void **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()

void **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()

void **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_dependenc() 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()

class chained_task
#include <task_graph.hpp> 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()* or*add_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 constructor. Constructs an invalid *chained_task*. Such a task cannot be placed in a graph of tasks.

chained_task(*task* *t*, *executor_t* *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

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

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

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.

Private Members

std::shared_ptr<detail::chained_task_impl> **`impl_`**

Friends

void **`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()*

void **`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()*

void **`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_dependenc() 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()*

1.3.2 Executors

executor_type.hpp

namespace concore

Typedefs

using executor_t = std::function<void (task) >
Generic executor type.

This is a type-erasure, allowing all executor types to be stored and manipulated.

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

It is assumed that multiple tasks/threads can call the executor at the same time to enqueue tasks with it.

See global_executor, immediate_executor, serializer

global_executor.hpp

namespace concore

namespace v1

Variables

constexpr auto **global_executor** = detail::executor_with_prio<detail::task_priority::normal>{}
The default global executor.

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

Is is the default executor.

The tasks enqueued here are considered to have the priority "normal".

See global_executor_critical_prio, global_executor_high_prio, global_executor_normal_prio, global_executor_low_prio, global_executor_background_prio

constexpr auto **global_executor_critical_prio** = detail::executor_with_prio<detail::task_priority::critical>{
Task executor that enqueues tasks with *critical* priority.

Similar to global_executor, but the task is considered to have the *critical* priority. Tasks with critical priority take precedence over all other types of tasks in the task system.

See global_executor, global_executor_high_prio, global_executor_normal_prio, global_executor_low_prio, global_executor_background_prio

constexpr auto **global_executor_high_prio** = detail::executor_with_prio<detail::task_priority::high>{}
Task executor that enqueues tasks with *high* priority.

Similar to global_executor, but the task is considered to have the *high* priority. Tasks with high priority take precedence over normal priority tasks.

See global_executor, global_executor_critical_prio, global_executor_normal_prio, global_executor_low_prio, global_executor_background_prio

constexpr auto **global_executor_normal_prio** = detail::executor_with_prio<detail::task_priority::normal>{}
Task executor that enqueues tasks with *normal* priority.

Same as global_executor.

See global_executor, global_executor_critical_prio, global_executor_high_prio, global_executor_low_prio, global_executor_background_prio

constexpr auto **global_executor_low_prio** = detail::executor_with_prio<detail::task_priority::low>{}
Task executor that enqueues tasks with *low* priority.

Similar to global_executor, but the task is considered to have the *low* priority. Tasks with low priority are executed after tasks of normal priority.

See global_executor, global_executor_critical_prio, global_executor_high_prio, global_executor_normal_prio, global_executor_background_prio

constexpr auto **global_executor_background_prio** = detail::executor_with_prio<detail::task_priority::backgr
Task executor that enqueues tasks with *background* priority.

Similar to global_executor, but the task is considered to have the *background* priority. Tasks with background are executed after all the other tasks in the system

See global_executor, global_executor_critical_prio, global_executor_high_prio, global_executor_normal_prio, global_executor_low_prio

immediate_executor.hpp

namespace concore

namespace v1

Variables

constexpr auto **immediate_executor** = detail::immediate_executor_type{}
Executor that executes all the tasks immediately.

Most executors are storing the tasks for later execution and the enqueueing finishes very fast. This one executes the task during enqueueing, without scheduling it for a later time.

dispatch_executor.hpp

namespace concore

namespace v1

Variables

constexpr auto **dispatch_executor** = detail::disp::executor_with_prio<detail::disp::task_priority::normal>{}
Executor that enqueues task in libdispatch.

The tasks that are enqueued by this executor will have *normal* priority inside libdispatch.

This can be used as a bridge between concore and libdispatch.

See global_executor

constexpr auto **dispatch_executor_high_prio** = detail::disp::executor_with_prio<detail::disp::task_priority::h
Task executor that enqueues tasks in libdispatch with *high* priority.

This can be used as a bridge between concore and libdispatch.

constexpr auto **dispatch_executor_normal_prio** = detail::disp::executor_with_prio<detail::disp::task_priority
Task executor that enqueues tasks in libdispatch with *normal* priority.

Same as dispatch_executor.

This can be used as a bridge between concore and libdispatch.

constexpr auto **dispatch_executor_low_prio** = detail::disp::executor_with_prio<detail::disp::task_priority::low
Task executor that enqueues tasks in libdispatch with *low* priority.

This can be used as a bridge between concore and libdispatch.

tbb_executor.hpp

namespace concore

namespace v1

Variables

constexpr auto **tbb_executor** = detail::tbb_d::executor_with_prio<detail::tbb_d::task_priority::normal>{}
Executor that enqueues task in TBB.

The tasks that are enqueued by this executor will have *normal* priority inside TBB.

This can be used as a bridge between concore and Intel TBB.

See global_executor

constexpr auto **tbb_executor_high_prio** = detail::tbb_d::executor_with_prio<detail::tbb_d::task_priority::high>
Task executor that enqueues tasks in TBB with *high* priority.

This can be used as a bridge between concore and Intel TBB.

constexpr auto **tbb_executor_normal_prio** = detail::tbb_d::executor_with_prio<detail::tbb_d::task_priority::no
Task executor that enqueues tasks in TBB with *normal* priority.

Same as tbb_executor.

This can be used as a bridge between concore and Intel TBB.

constexpr auto **tbb_executor_low_prio** = detail::tbb_d::executor_with_prio<detail::tbb_d::task_priority::low>{}
Task executor that enqueues tasks in TBB with *low* priority.

This can be used as a bridge between concore and Intel TBB.

1.3.3 Serializers

serializer.hpp

namespace concore

namespace v1

class serializer

#include <serializer.hpp> 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 executor_t, global_executor, spawn_continuation_executor, *n_serializer*, *rw_serializer*

Public Functions

serializer(*executor_t base_executor* = {}, *executor_t 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

void **operator()** (*task t*)
Function call operator.

If there are no 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 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
- `t`: The tasks to be enqueued in the serializer

Private Members

std::shared_ptr<impl> **impl_**
The implementation object of this serializer. We need this to be shared pointer for lifetime issue, but also to be able to copy the serializer easily.

n_serializer.hpp

namespace concore

namespace v1

class n_serializer : **public** std::enable_shared_from_this<*n_serializer*>
#include <n_serializer.hpp> 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*, executor_t, global_executor, spawn_continuation_executor

Public Functions

n_serializer (int *N*, *executor_t* *base_executor* = {}, *executor_t* *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

void **operator()** (*task* *t*)
Function call operator.

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

- `t`: The tasks to be enqueued in the serializer

Private Members

std::shared_ptr<impl> **impl_**
The implementation object of this *n_serializer*. We need this to be shared pointer for lifetime issue, but also to be able to copy the serializer easily.

rw_serializer.hpp

namespace concore

namespace v1

class rw_serializer
#include <rw_serializer.hpp> 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: read() and write(). The read() executor should be used to enqueue *READ* tasks while the write() 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(*executor_t base_executor* = {}, *executor_t 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

Private Members

std::shared_ptr<impl> **impl_**
Implementation detail shared by both reader and writer types.

class reader_type
#include <rw_serializer.hpp> 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*.

void **operator()**(*task* *t*)
Function call operator.

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.
Parameters
- `t`: The *READ* task to be enqueued

Private Members

std::shared_ptr<impl> **impl_**

class writer_type
#include <rw_serializer.hpp> 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*.

void **operator()**(*task* *t*)
Function call operator.

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.
Parameters
- `t`: The *WRITE* task to be enqueued

Private Members

std::shared_ptr<impl> **impl_**

1.3.4 Data

data/concurrent_queue.hpp

namespace concore

namespace v1

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

#include <concurrent_queue.hpp> 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 queue_type, *push()*, pop()

Public Types

template<>
using value_type = T
The value type of the concurrent queue.

Public Functions

concurrent_queue()
Default constructor. Creates a valid empty queue.

concurrent_queue(**const** *concurrent_queue*&)
Copy constructor is DISABLED.

const concurrent_queue &**operator=**(**const** concurrent_queue&)
Copy assignment is DISABLED.

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.

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. 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()*

Private Types

template<>
using node_ptr = detail::node_ptr

Private Members

detail::concurrent_queue_data **queue_**
The data holding the actual queue.

detail::node_factory<T> **factory_**
Object that creates nodes, and keeps track of the freed nodes.

data/concurrent_queue_type.hpp

namespace concore

namespace v1

Enums

enum 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:

single_prod_single_cons
Single-producer, single-consumer concurrent queue.

single_prod_multi_cons
Single-producer, multiple-consumer concurrent queue.

multi_prod_single_cons
Multiple-producer, single-consumer concurrent queue.

multi_prod_multi_cons
Multiple-producer, multiple-consumer concurrent queue.

default_type = *multi_prod_multi_cons*
The default queue type. Multiple-producer, multiple-consumer concurrent queue.

1.3.5 Low level

low_level/spin_backoff.hpp

Defines

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*

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*

namespace concore

namespace v1

class spin_backoff
#include <spin_backoff.hpp> 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.

Private Members

int **count_** = {1}
The count of 'pause' instructions we should make.

low_level/spin_mutex.hpp

namespace concore

namespace v1

class spin_mutex
#include <spin_mutex.hpp> 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 constructor.

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

spin_mutex(**const** *spin_mutex*&)
Copy constructor is DISABLED.

spin_mutex &**operator=**(**const** *spin_mutex*&)
Copy assignment is DISABLED.

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()*

Private Members

std::atomic_flag **busy_** = ATOMIC_FLAG_INIT
True if the spin mutex is taken.

low_level/shared_spin_mutex.hpp

namespace concore

namespace v1

class shared_spin_mutex
#include <shared_spin_mutex.hpp> 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 constructor.

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

shared_spin_mutex(**const** *shared_spin_mutex*&)
Copy constructor is DISABLED.

shared_spin_mutex &**operator=**(**const** *shared_spin_mutex*&)
Copy assignment is DISABLED.

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()*

Private Members

std::atomic<uintptr_t> **lock_state_** = {0}
The state of the shared spin mutex. The first 2 LSB will indicate whether we have a writer or we are having a pending writer. The rest of the bits indicates the count of the readers.

Private Static Attributes

constexpr uintptr_t **has_writer_** = 1
Bitmask to check if we have a writer acquiring the mutex.

constexpr uintptr_t **has_writer_pending_** = 2
Bitmask to check if somebody tries to acquire the mutex as writer.

constexpr uintptr_t **has_writer_or_pending_** = *has_writer_* | *has_writer_pending_*
Bitmask indicating that we have a writer, or a pending writer.

constexpr uintptr_t **readers_** = ~(*has_writer_or_pending_*)
Bitmask with that capture all the readers that we have.

constexpr uintptr_t **is_busy_** = (*has_writer_* | *readers_*)
Bitmask indicating whether we have a writer or readers.

constexpr uintptr_t **reader_increment_** = 4
The increment that we need to use for each reader.

low_level/semaphore.hpp

namespace concore

namespace v1

class binary_semaphore
#include <*semaphore.hpp*> 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()

~binary_semaphore()
Destructor.

binary_semaphore(**const** *binary_semaphore*&)
Copy constructor is DISABLED.

void **operator=**(**const** *binary_semaphore*&)
Copy 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 semaphore
#include <*semaphore.hpp*> 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*&)
Copy constructor is DISABLED.

void **operator=**(**const** *semaphore*&)
Copy assignment is DISABLED.

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()*

low_level/concurrent_dequeue.hpp

namespace concore

namespace v1

template<typename **T**>
class concurrent_dequeue

#include <concurrent_dequeue.hpp> 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

template<>
using value_type = T

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

- `[in] 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.

Private Members

detail::bounded_dequeue<T> **fast_deque_**
The fast dequeue implementation; uses a fixed number of elements.

std::deque<T> **slow_access_elems_**
Deque of elements that have slow access; we use this if we go beyond our threshold.

std::mutex **bottleneck_**
Protects the access to slow_access_elems_.

std::atomic<int> **num_elements_slow_** = {0}
The number of elements stored in slow_access_elems_; used it before trying to take the lock.

INDEX