# Task Blocks

This lesson gives an overview of task-blocks, predicted to be introduced in C++20.

WE'LL COVER THE FOLLOWING

- Fork & Join
  - `define_task_block` versus `define_task_block_restore_thread`
  - The Interface
  - The Scheduler

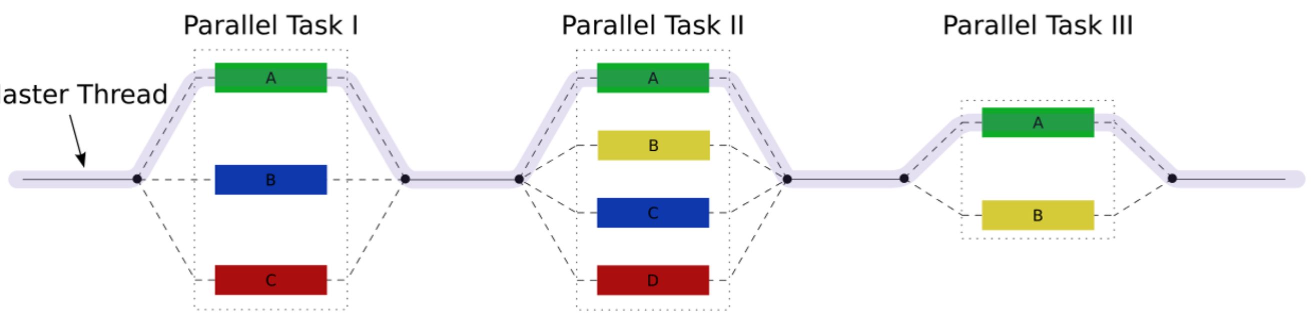
Task blocks use the well-known fork-join paradigm for the parallel execution of tasks.

Who invented it in C++? Both Microsoft with its Parallel Patterns Library (PPL) and Intel with its Threading Building Blocks (TBB) were involved in the proposal N4441. Additionally, Intel used its experience with its Cilk Plus library.

The name fork-join is quite easy to explain.

## Fork & Join

The simplest approach to explain the fork-join paradigm is through a graph.



### How does it work?

The creator invokes `define_task_block` or `define_task_block_restore_thread`. This call creates a task block that can create tasks or it can wait for their completion. The synchronization is at the end of the task block. The creation of a new task is the fork phase; the synchronization of the task block is the join phase of the workflow. Admittedly, that was the simplistic description. Let's have a look at a piece of code.

```
template <typename Func>
int traverse(node& n, Func && f){
    int left = 0, right = 0;
    define_task_block(
        [&](task_block& tb){
            if (n.left) tb.run([&]{ left = traverse(*n.left, f); });
            if (n.right) tb.run([&]{ right = traverse(*n.right, f); });
         }
    );
    return f(n) + left + right;
}
```

`traverse` is a function template that invokes function `f` on each node of its tree. The keyword `define_task_block` defines the task block. The task block `tb` can start a new task in this block; that's exactly what happens at the left and right branches of the tree in lines 6 and 7. Line 9 is the end of the task block and, hence, the synchronization point.

#### HPX (High Performance ParalleX)

The above example is from the documentation for the HPX (High Performance ParalleX) framework, which is a general purpose C++ runtime system for parallel and distributed applications of any scale. HPX has already implemented many features of the upcoming C++20 standard.

You can define a task block by using either the function `define_task_block` or the function `define_task_block_restore_thread`.

## `define_task_block` versus

## `define_task_block` versus `define_task_block_restore_thread`

The subtle difference is that `define_task_block_restore_thread` guarantees that the creator thread of the task block is the same thread that will run after the task block.

```
...
define_task_block([&](auto& tb){
  tb.run([&]{[] func(); });
  define_task_block_restore_thread([&](auto& tb){
    tb.run([&]([]{ func2(); });
    define_task_block([&](auto& tb){
       tb.run([&]{ func3(); }
    });
    ...
    ...
  });
  ...
  ...
});
...
...
```

Task blocks ensure that the creator thread of the outermost task block (lines 2 - 14) is exactly the same thread that will run the statements after finishing the task block. This means that the thread that executes line 2 is the same thread that executes lines 15 and 16. However, this guarantee will not hold for nested task blocks; therefore, the creator thread of the task block in lines 6 - 8 will not automatically execute lines 9 and 10. If you need that guarantee, you should use the function `define_task_block_restore_thread` (line 4). Now it holds that the creator thread executing line 4 is the same thread executing lines 12 and 13.

## The Interface

A task_block has a very limited interface. You can not explicitly define it; you have to use either function `define_task_block` or `define_task_block_restore_thread`. The task_block `tb` is in the scope of the defined task block active and can, therefore, start new tasks (`tb.run`) or wait (`tb.wait`) until the task is done.

```
define_task_block([&](auto& tb){
  tb.run([&]{ process(x1, x2) });
  if (x2 == x3) tb.wait();
  process(x3, x4);
});
```

What is the code snippet doing? In line 2 a new task is started. This task needs the data `x1` and `x2`. Line 4 uses the data `x3` and `x4`. If `x2 == x3` is true, the variables have to be protected from shared access. This is the reason why the task block `tb` waits until the task in line 2 is done.

## The Scheduler

The scheduler manages which thread is running. This means that it is no longer the responsibility of the programmer to decide who executes the task. In this case, threads are just an implementation detail.

There are two strategies for executing the newly created task. The parent represents the creator thread and the child the new task.

**Child stealing**: The scheduler steals the task and executes it.

**Parent stealing**: The task block `tb` itself executes the task. Now the scheduler steals the parent.

Proposal N4441 supports both strategies.