Bug Report

#402 Deadlock when running subflows in a pipeline

1. Problem Statement

The problem is stated in https://github.com/taskflow/taskflow/issues/402. Listing 1 shows the code. When we spawn Subflow tasks in a pipe and call *Runtime::run* to schedule the Subflow tasks, we run into a data race and trap in an infinite loop. We print out the join counter of the scheduled runtime task in *Executor::_invoke* and find out the value is the maximum value of size_t. Since a spawned Subflow task has the runtime task as its parent, when a worker finishes a Subflow task and begins to decrement the join counter of the parent of the finished Subflow task (runtime task), the join counter is zero already. Decrementing one from zero results in the maximum value of size_t. Now the parent node (runtime task) has a join counter of an extreme big value but no spawned task in its queue. There is no way to decrement its join counter to zero. The runtime task then traps in an infinite loop

```
tf::Pipeline pl(
  num_lines,
  tf::Pipe{
    tf::PipeType::SERIAL, [max_tokens](tf::Pipeflow& pf){
      if (pf.token() == max_tokens) {
        pf.stop();
      }
    }
  },

  tf::Pipe{
    tf::PipeType::PARALLEL, [&](tf::Pipeflow&, tf::Runtime& rt) mutable {
      rt.run([&](tf::Subflow& sf) mutable {
        for (size_t i = 0; i < 2; ++i) {
          sf.emplace([&](){
            ++sums;
          });
        }
      });
    }
  }
);
```

Listing 1. Code of the problem.

2. Why The Problem Exists?

In our pipeline algorithm, we guarantee that only one worker runs the *on_pipe* of each parallel line. Sometimes there are two workers running on the same parallel line. For example, in Figure 1 `worker j` resolves the horizontal dependency, decrements the join counter of `C` to one. Then `worker i` resolves the vertical dependency, decrements the join counter of `C` to zero and calls *Runtime::schedule* to schedule `C`. At this moment, `woker j` has not yet returned and another worker may already start to run *C*. Then we have two workers running on the same parallel line.

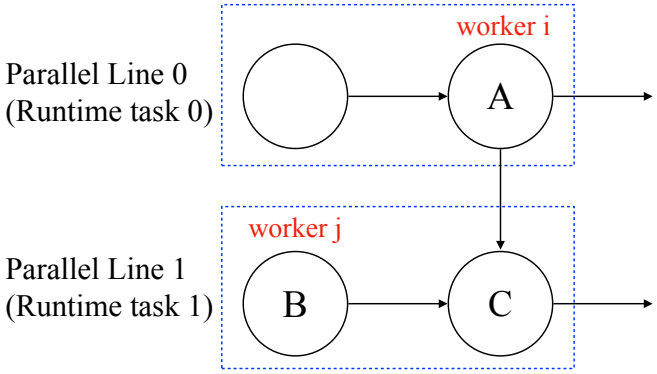

Figure 1. A pipeline of two parallel lines. Blue boxes depict parallel lines. `worker i` is running `A` and `worker j` is running `B`.

Remember in our pipeline algorithm every task has two works to do in sequence. The first part is to execute *on_pipe* and the other part is to resolve dependency. Although we may have two workers running on the same parallel line, we guarantee that one worker is running the *on_pipe* and the other worker is dealing with the scheduling.

The problem arises when a worker starts to execute *on_pipe* of `C` and updates the join counter of runtime task 1 to the number of spawned Subflow tasks of zero dependent. Right after the update of the join counter by a worker, `work j` returns and starts to reset the join counter of runtime task 1 to its initial value. Since there are two workers running runtime task 1, the reset and update of the join counter interleaves. That is when the problem starts.

Figure 2 illustrates the cause of the problem. The red rectangular is supposed to run before the blue rectangular because red rectangular represents the dependency resolving work of `B` and the blue represents the `on_pipe` work of `C`. Since there are two workers running runtime task 1, the execution order of the red and the blue boxes can not be guaranteed. When the blue boxes runs before the red boxes, the join counter of runtime task 1 are updated to two by `worker k` first. Then the red box runs and `worker j` resets the join counter of runtime task 1 to zero and returns to the thread pool. Next, in the orange rectangular `worker k` finishes one Subflow task, decrements the zero join counter of runtime task 1 by one and gets the maximum value. An infinite loop happens.

3. Solution

To solve the problem, in *Executor::_invoke(Worker& worker, Node* node)*, we do not directly assign the join counter to node's initial value but *ADD* the value. In this way, the update of join counter in the blue rectangular in Figure 2 is not revoked in the reset in the red rectangular when the blue box runs before the red box. As shown in Listing 2, we *ADD* rather than assign.

```
if((node->_state.load(std::memory_order_relaxed) & Node::CONDITIONED)) {
  //node->_join_counter = node->num_strong_dependents();
  node->_join_counter.fetch_add(node->num_strong_dependents());
}
else {
  //node->_join_counter = node->num_dependents();
  node->_join_counter.fetch_add(node->num_dependents());
```

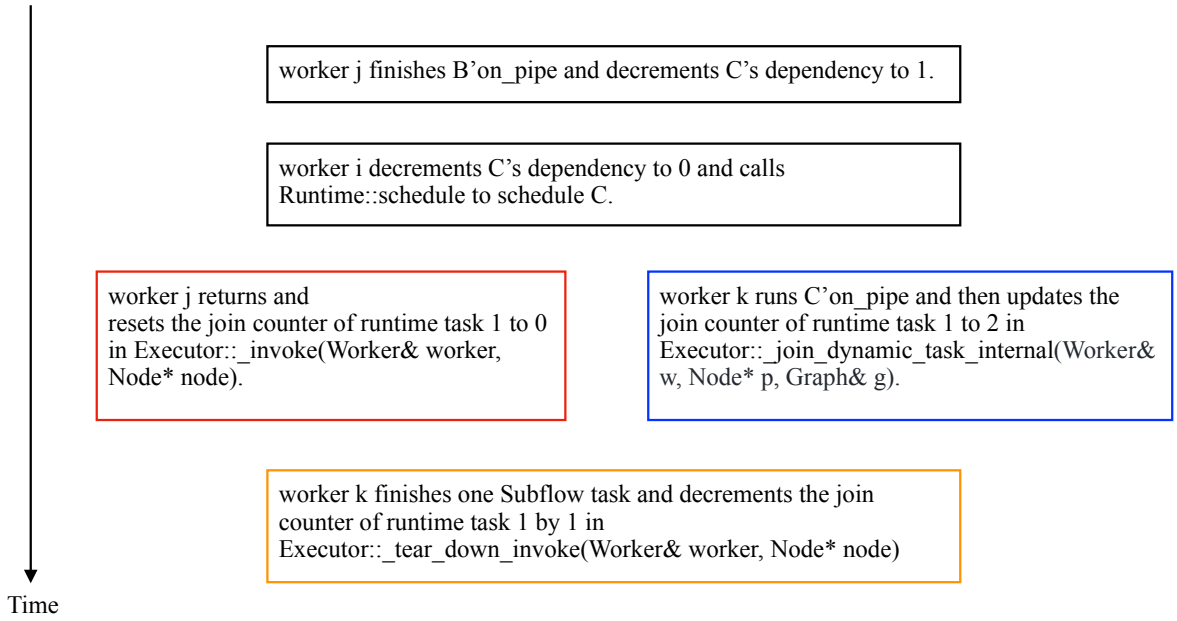

Figure 2. Illustration of the problem when running the pipeline of Listing 1. The symbols align with those in Figure 1.

```
}
```

Listing 2. Solution to the bug.