Parallel Programming on Top of SystemC

Matthieu Moy

Laboratoire de l'Informatique du Parallélisme
CASH Team

April 2022

Copyright Permission

Problems and solutions for parallel execution of SystemC/TLM

(1) Which process can be run in parallel?

(2) How to ensure co-routine semantics?

Problems and solutions for parallel execution of SystemC/TLM

(1) Which process can be run in parallel?
$\leadsto$ Same simulated time $\Rightarrow$ parallel?

(2) How to ensure co-routine semantics?
$\leadsto$ run-time monitoring (SCale), static analysis (RISC)

Problems and solutions for parallel execution of SystemC/TLM

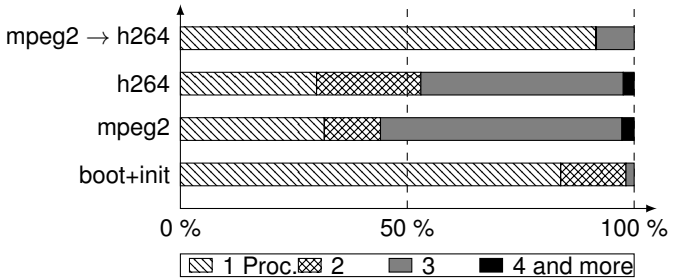
(1) Which process can be run in parallel? ← focus of this talk
   ⇝ Same simulated time ⇒ parallel?

(2) How to ensure co-routine semantics?
   ⇝ run-time monitoring (SCale), static analysis (RISC)

Our proposal = additional constructs:
Desynchronization (1) / Synchronization (2) (somehow)

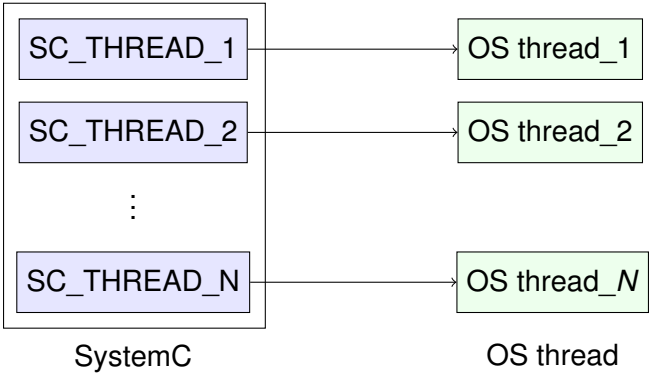
Bad news ...

Number of process (`SC_THREAD` + `SC_METHOD`) per δ-cycle:



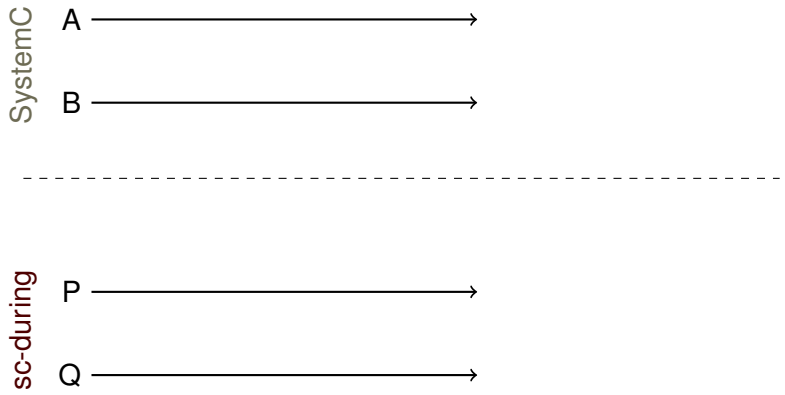
(Platform from STMicroelectronics, more data in “Parallel Simulation of Loosely Timed SystemC/TLM Programs: Challenges Raised by an Industrial Case Study”, MDPI 2015)

SC-DURING: The Idea

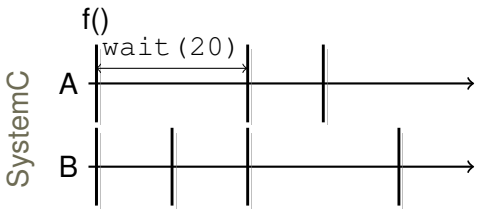


- Unmodified SystemC
- Some computation delegated to other threads
- Weak synchronization between SystemC and threads thanks to tasks with duration

Simulated Time in SystemC and SC-DURING

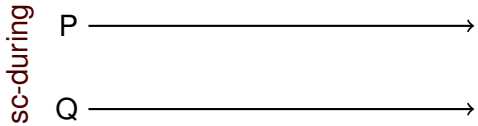


Simulated Time in SystemC and SC-DURING

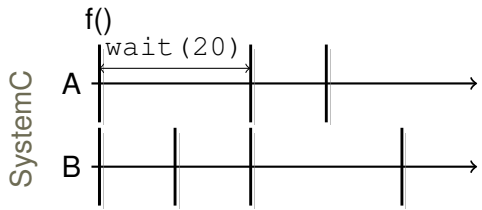


Process A:

```
//Computation
f();
//Time taken by f
wait(20);
```

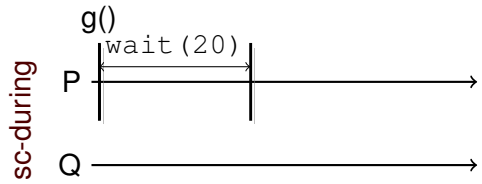


Simulated Time in SystemC and SC-DURING



Process A:

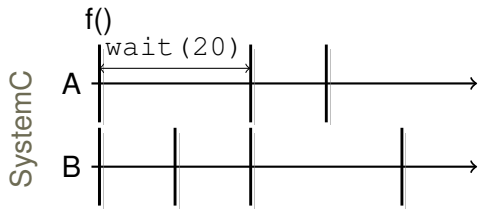
```
//Computation
f();
//Time taken by f
wait(20);
```



Process P:

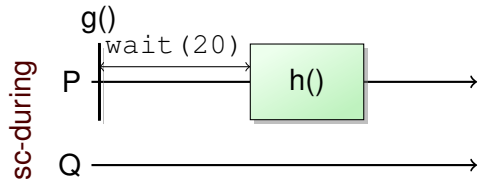
```
g();
wait(20);
```

Simulated Time in SystemC and SC-DURING



Process A:

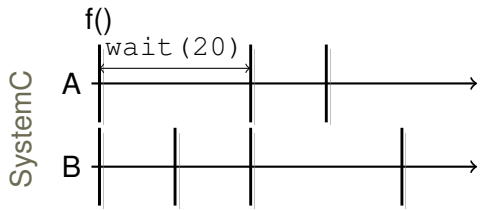
```
//Computation
f();
//Time taken by f
wait(20);
```



Process P:

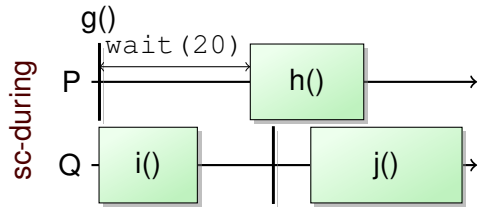
```
g();
wait(20);
during(15, h);
```

Simulated Time in SystemC and SC-DURING



Process A:

```
//Computation
f();
//Time taken by f
wait(20);
```

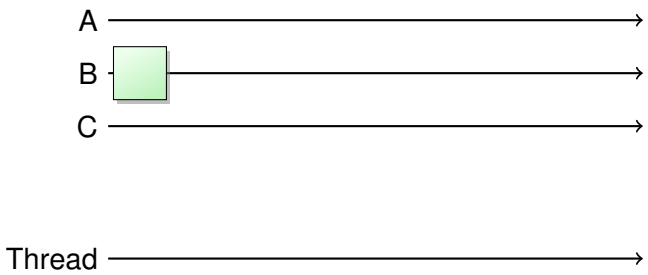


Process P:

```
g();
wait(20);
during(15, h);
```

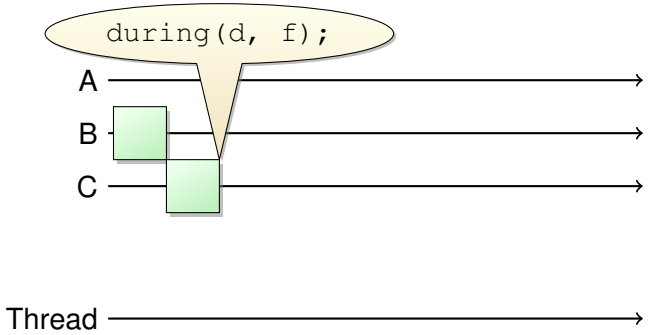
SC-DURING: First (Naive) Implementation

```
void during(sc_core::sc_time d,
            std::function<void()> f) {
  1  std::thread t(f); // Thread creation
  2  sc_core::wait(d); // SystemC executes
  3  t.join(); // Wait for completion
}
```



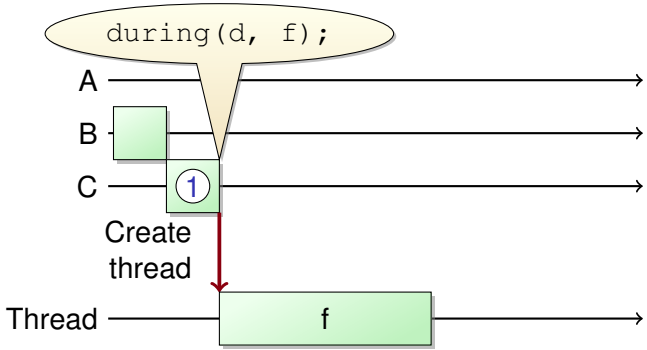
SC-DURING: First (Naive) Implementation

```
void during(sc_core::sc_time d,
            std::function<void()> f) {
  std::thread t(f); // Thread creation
  sc_core::wait(d); // SystemC executes
  t.join(); // Wait for completion
}
```



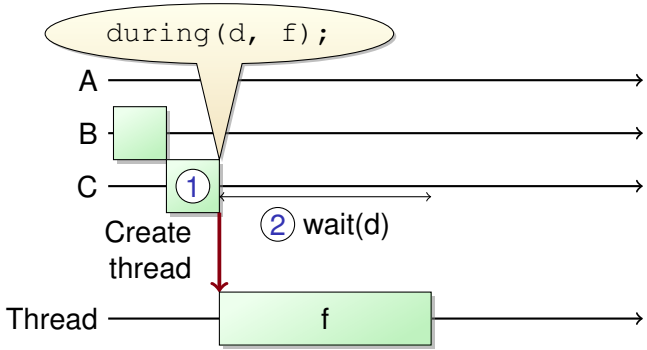
SC-DURING: First (Naive) Implementation

```
void during(sc_core::sc_time d,
            std::function<void()> f) {
  (1) std::thread t(f); // Thread creation
  (2) sc_core::wait(d); // SystemC executes
  (3) t.join(); // Wait for completion
}
```



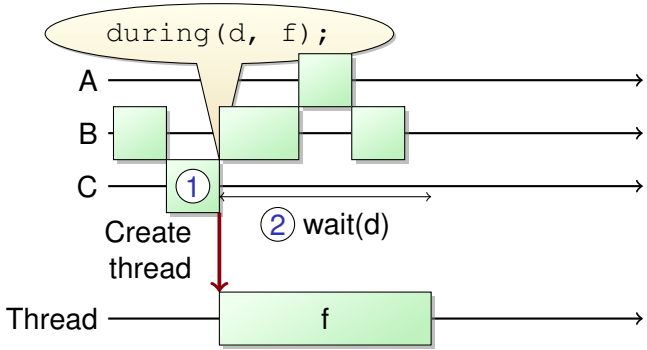
SC-DURING: First (Naive) Implementation

```
void during(sc_core::sc_time d,
            std::function<void()> f) {
  (1) std::thread t(f); // Thread creation
  (2) sc_core::wait(d); // SystemC executes
  (3) t.join(); // Wait for completion
}
```



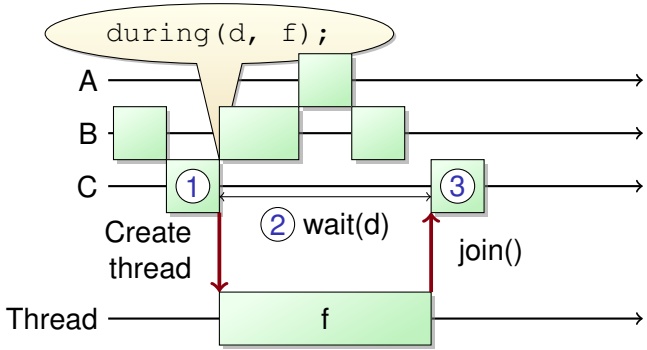
SC-DURING: First (Naive) Implementation

```
void during(sc_core::sc_time d,
            std::function<void()> f) {
  ① std::thread t(f); // Thread creation
  ② sc_core::wait(d); // SystemC executes
  ③ t.join(); // Wait for completion
}
```



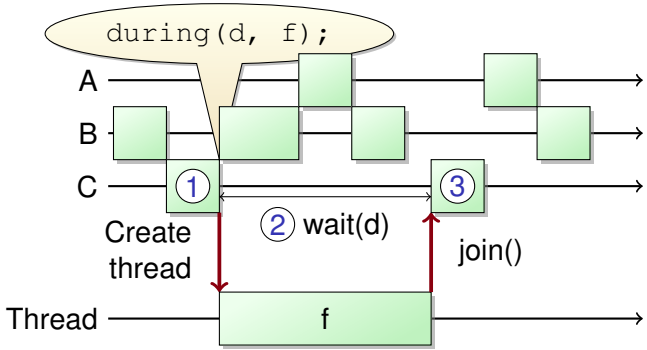
SC-DURING: First (Naive) Implementation

```
void during(sc_core::sc_time d,
            std::function<void()> f) {
  (1) std::thread t(f); // Thread creation
  (2) sc_core::wait(d); // SystemC executes
  (3) t.join(); // Wait for completion
}
```



SC-DURING: First (Naive) Implementation

```
void during(sc_core::sc_time d,
            std::function<void()> f) {
1   std::thread t(f); // Thread creation
2   sc_core::wait(d); // SystemC executes
3   t.join(); // Wait for completion
}
```



Wait ... are you saying that parallelization is just about fork/join?

Wait ... are you saying that parallelization is just about fork/join?

Well, sometimes it is ...

When Things are Easy: Pure Function

Before

```
compute_in_systemc();

// my profiler says it's
// performance critical.
// does not communicate
// with other processes.
big_computation();
wait(10, SC_MS);

next_computation();
```

After

```
compute_in_systemc();



// Won't be a performance
// bottleneck anymore
during(10, SC_MS,
       big_computation);

next_computation();
```

Wait ... are you saying that parallelization is just about fork/join?

Well, sometimes it is ...

Wait ... are you saying that parallelization is just about fork/join?

Well, sometimes it is ...

... and sometimes it isn't

Wait ... are you saying that parallelization is just about fork/join?

Well, sometimes it is ...

... and sometimes it isn't:

Time synchronization: make sure things are executed at the right simulated time

Data/scheduler synchronization: avoid data-race between tasks, processes and the SystemC scheduler.

SC-DURING: Synchronization

extra_time(t): increase current task duration

SC-DURING: Synchronization

extra_time(t): increase current task duration

catch_up(t): block task until SystemC's time reaches the end of the current task

```
while (!c) {
    extra_time(10, SC_NS);
    catch_up(); // ensures fairness
}
```

`extra_time()`: Sketch of Implementation

- SystemC side:

```
void during(duration, routine) {
    end = now() + duration;
    std::thread t(routine);
    // used to be just sc_core::wait(duration)
    while (now() != end)
        sc_core::wait(end - now());
    t.join();
}
```

- SC-DURING task side:

```
void extra_time(duration) {
    end += duration;
}
```

```
void catch_up() {
    while (now() != end)
        // avoid busy-waiting
        condition.wait();
}
```

Temporal decoupling and SC-DURING

Plain SystemC

```
f();
// instead of wait(42)
t_local += 42;
g();
t_local += 12;

// Re-synchronize with
// SystemC time
wait(t_local);
t_local = 0;

i();
```

Inside SC-DURING tasks

```
f();
// instead of wait(42)
extra_time(42);
g();
extra_time(12);

// Re-synchronize with
// SystemC time
catch_up();


i();
```

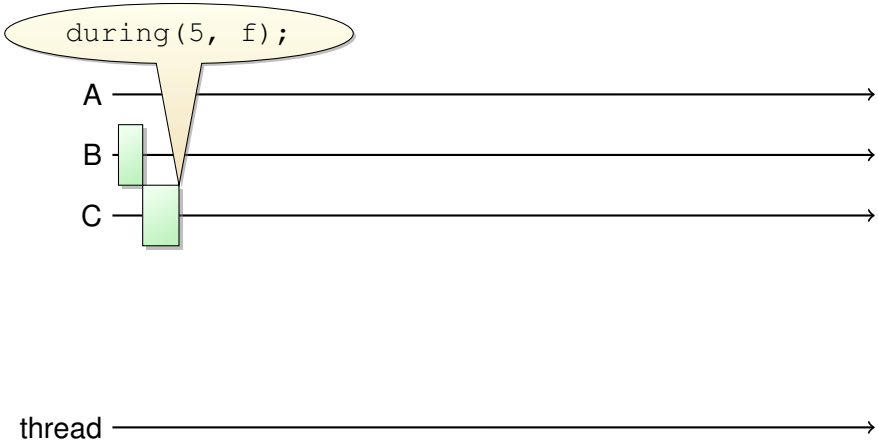
`sc_call()`: be cooperative for a while

sc_call(f): call function `f` in the context of SystemC

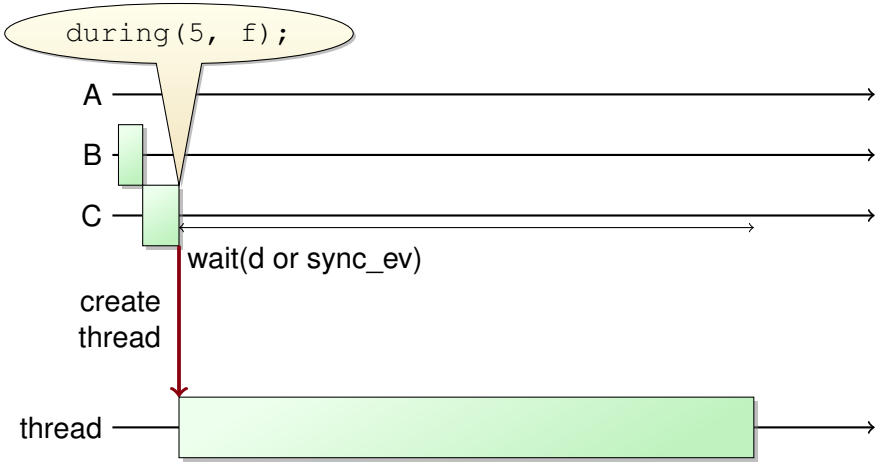
```
e.notify(); // Forbidden in during tasks

sc_call([]{e.notify()});
sc_call([]{i++"});
```

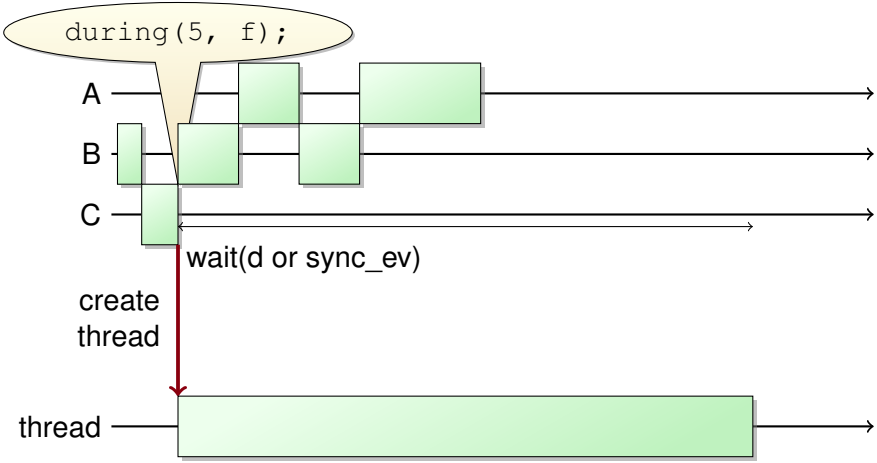
`sc_call()`: Underlying Mechanism



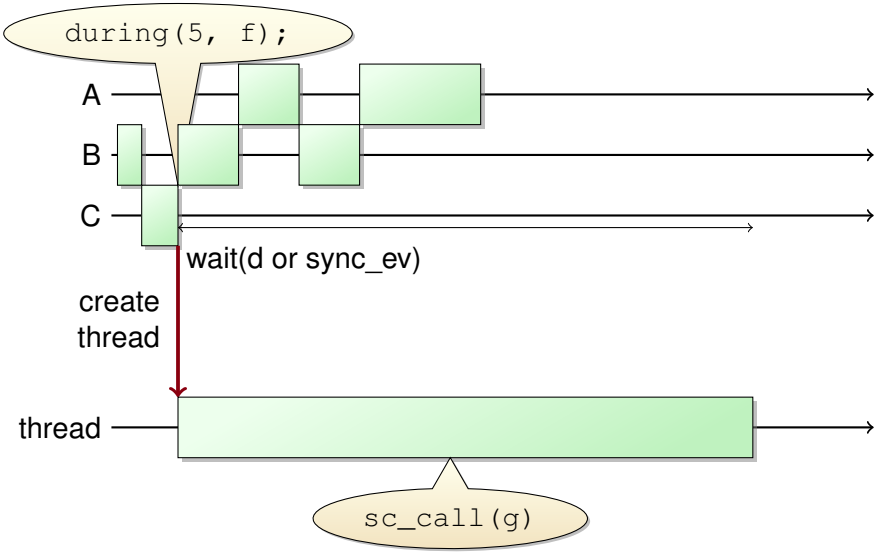
`sc_call()`: Underlying Mechanism



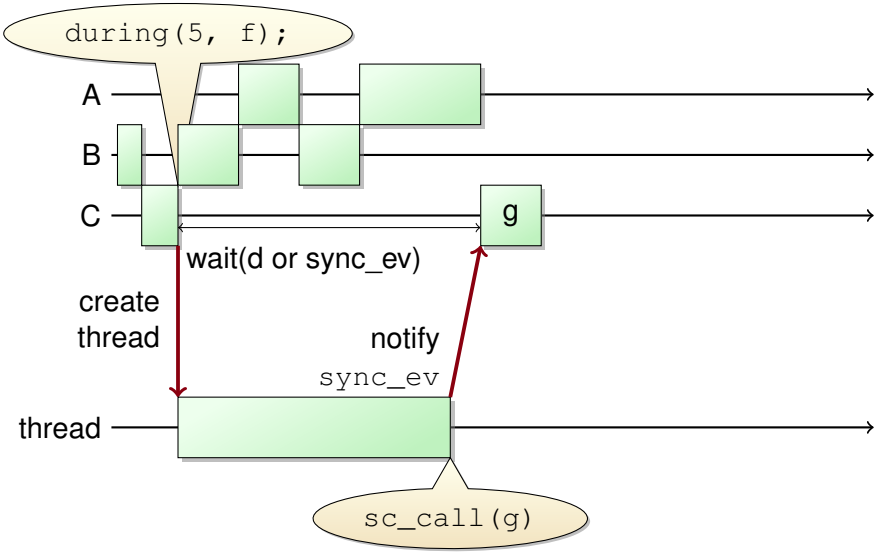
`sc_call()`: Underlying Mechanism



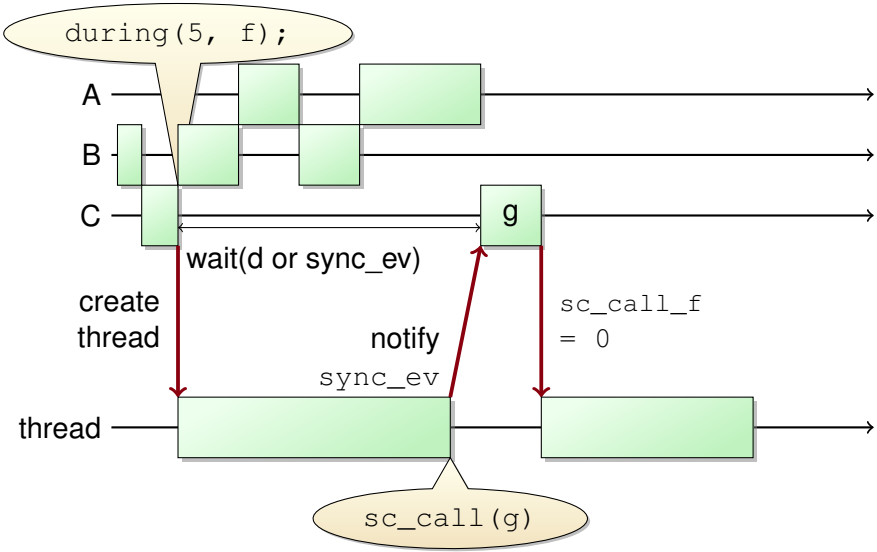
`sc_call()`: Underlying Mechanism



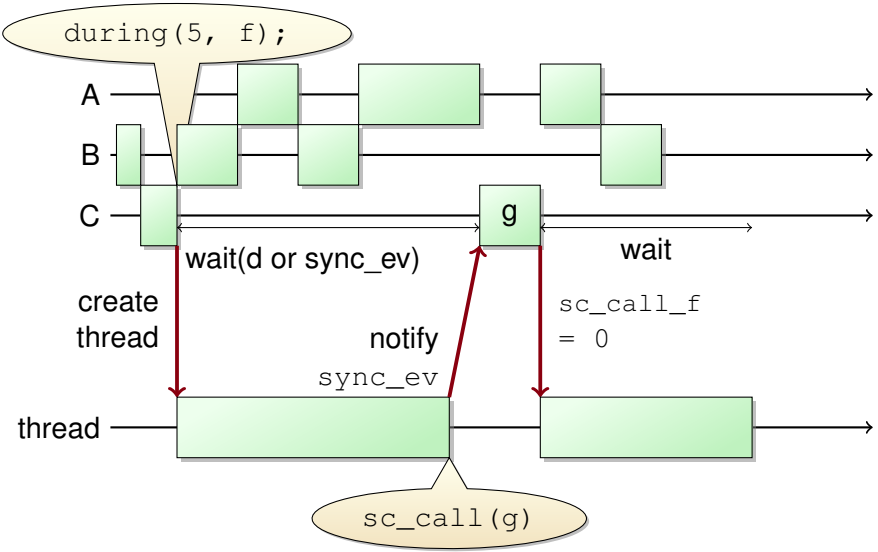
`sc_call()`: Underlying Mechanism



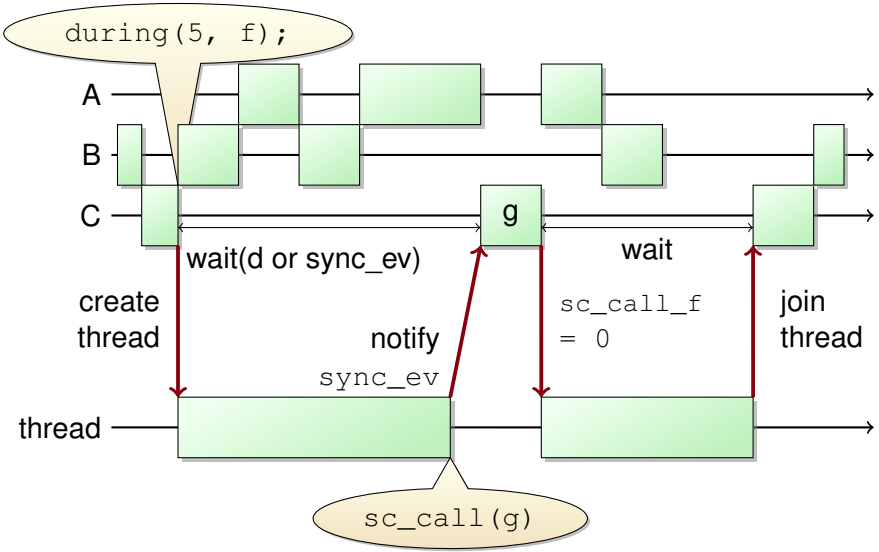
`sc_call()`: Underlying Mechanism



`sc_call()`: Underlying Mechanism



`sc_call()`: Underlying Mechanism

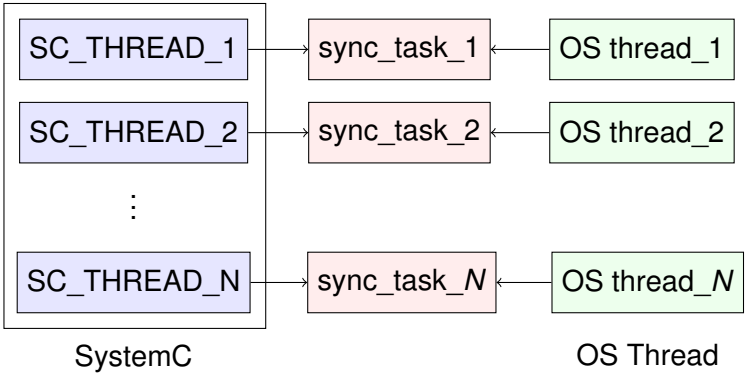


sc_call: Sketch of Implementation

```
void during(duration, f) {
   end = now() + duration;
   std::thread t(f);
   while (now() != end) {
      // wait sync_ev
      // with timeout:
      sc_core::wait
         (sync_ev,// <--
          end - now());
      if (sc_call_f) {
         sc_call_f();// <--
         sc_call_f = 0;
         condition.notify();
      }
   }
   t.join();
```

```
void sc_call(f) {
   sc_call_f = f;
   // Implemented w/
   // async_request_update()
   async_notify_event
       (sync_ev);
   while(sc_call_f != 0) {
      condition.wait();
   }
}
```

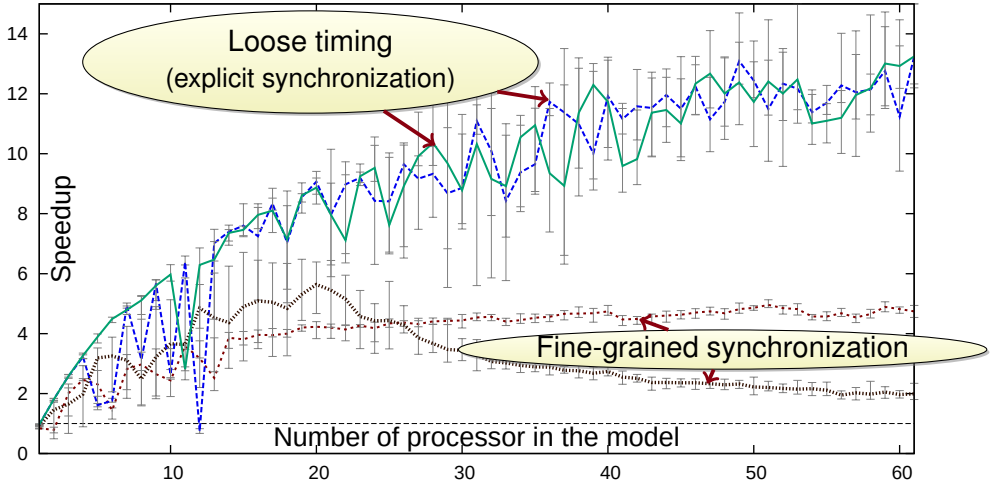
SC-DURING: Implementations



Strategies:

SEQ	Sequential (= reference)
THREAD	Thread creation + destruction for each task
POOL	Pre-allocated set of threads
ONDEMAND	Thread created on demand and reused

SC-DURING: Results



Test

machine : $4 \times 12 = 48$ cores

SC-DURING: Conclusion

- New way to express concurrency in the platform
- Allows parallel execution of loosely-timed (clockless) systems
- No modification of SystemC $\Rightarrow$ could work with a parallel SystemC kernel
- Possible improvement: performance optimizations (e.g. atomic operations + polling instead of system calls)

Try it:

`https://moy.gitlab.io/sc-during/`

Whishlist for Standard

- No change needed, sc-during already works ;-)
- Implementation detail: better way to implement `sc_call` would be nice
- More general: letting the user express loose timing directly in SystemC?

Whishlist for Standard

- No change needed, sc-during already works ;-)
- Implementation detail: better way to implement `sc_call` would be nice
- More general: letting the user express loose timing directly in SystemC?

Questions?

Whishlist for Standard

- No change needed, sc-during already works ;-)
- Implementation detail: better way to implement `sc_call` would be nice
- More general: letting the user express loose timing directly in SystemC?

Questions?

Thank You!

`https://moy.gitlab.io/sc-during/`