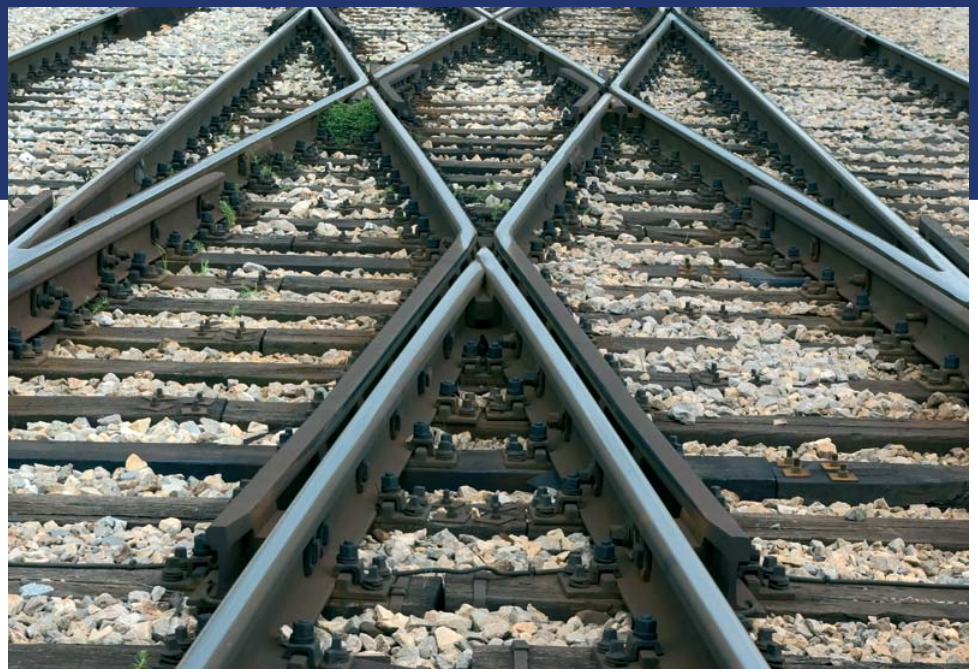
Parallel programming
C++11 threads
Part 2



FAKULTA
ELEKTROTECHNICKÁ
ČVUT V PRAZE

Lab topics

- **Future**, **promise** – synchronized access to values
  - e.g., returning values from threads
- Executing tasks by **async** object.
- **Atomic types** in C++11
- Exercise – barrier and odd-event sort

Promise object

- **promise** is used to store a value that is subsequently obtained by using the associated future object (synchronization point) in another thread.
- promise API:
  - https://en.cppreference.com/w/cpp/thread/promise
  - **promise**<T> prom; // creation
  - **future**<T> fut = prom.get_future(); // get related obj
  - prom.set_value (T());   // set promised value

Future object

- **future** object is used to obtain a from a thread
- if value is not yet available:
  - blocks until the value is computed (wait)
  - waits some time (wait_for, wait_until)
- future API:
  - https://en.cppreference.com/w/cpp/thread/future
  - T val = fut.get(); // get the returned value

Promise and future example

`lab_codes/PromiseAndFuture.cpp`

Async

- High-level API for running a thread that may return a value
- **async** executes a function asynchronously, i.e., without waiting for its completion and possibly with a delayed start
- async policy:
  - **launch::async** – creates a new thread
  - **launch::deferred** – function is started after its return value is requested (by using future object). Possible that does not create a new thread!
    - If the value of future is not requested, the function won't start
- Async API:
  - https://en.cppreference.com/w/cpp/thread/async
  - Execute the function asynchronously.
    - **future**<T> ret = **async**(function, params…);
  - Notice that the following blocks (async() waits for destructor of future)
    - **async**(lauch::async, function, params…);

Async example

`lab_codes/Async.cpp`

Atomicity in C++11

- Atomic operations are **indivisible**, i.e. they behave like one instruction.
- Useful for a non-blocking synchronization between threads.
- Often lock-free for **integer types**.
- Atomic operation:
  - **load** value
  - **modify** value
  - **write** value

```
int x = 0;
x += 5;
```

```
atomic<int> x(0);
x.fetch_add(5);
```

+= operation must be indivisible!

Atomicity in C++11

- https://en.cppreference.com/w/cpp/atomic/atomic
- Basic operations with atomic class:
  - load, store
  - operator++, operator--
  - fetch_add, fetch_sub, fetch_and, fetch_or, fetch_xor...
  - bool compare_exchange_strong (T& expected, T desired)
    - Sets the contained value to be desired if the contained value equals the expected value
    - true if expected is the same as the contained value

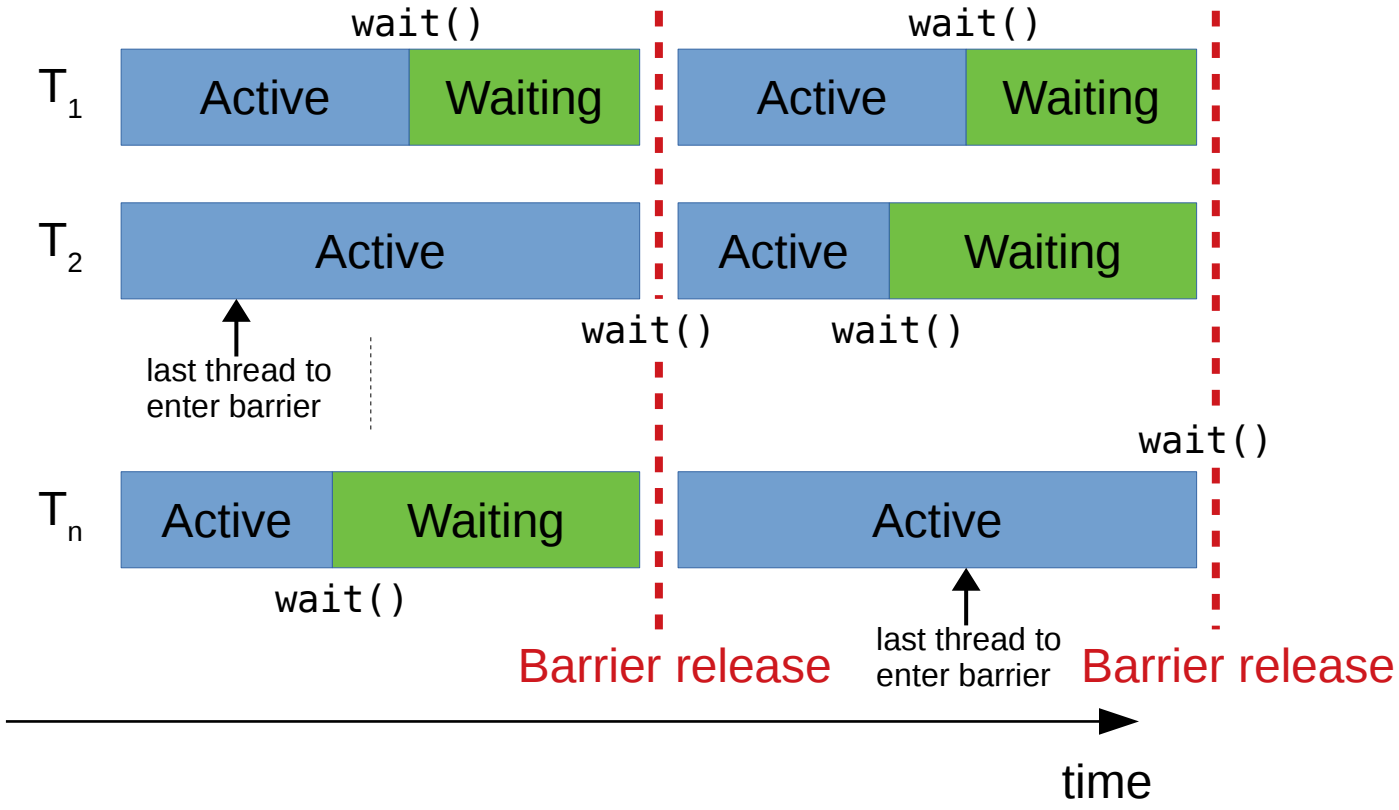
Atomic example

`lab_codes/AtomicCounter.cpp`

Main exercise – barrier

- API
  - Barrier(int numThreads);
  - Barrier.wait();
- synchronization of n threads
- threads wait on barrier until the last thread calls wait, which releases the barrier
- The barrier must be reusable, i.e., it can be released multiple times

Main exercise – barrier



Main exercise – barrier

- **Hints:**
  - Use two atomic variables and busy waiting
  - One atomic variable counts the number of waiting threads
  - Second atomic variable counts the barrier releases (*phase counter*)
    - Last thread use this variable to signal the release of barrier to other threads
- **Advanced:** replace busy waiting with waiting on a conditional variable

Additional exercise - sorting

- Write a parallel program for odd-even sort
  - Split the input array into numThreads * 2 buckets

6,3,9,1,9,7,2,6,2,1,6,5,7,6,4,4,2,3,9,6,7,9,2,6

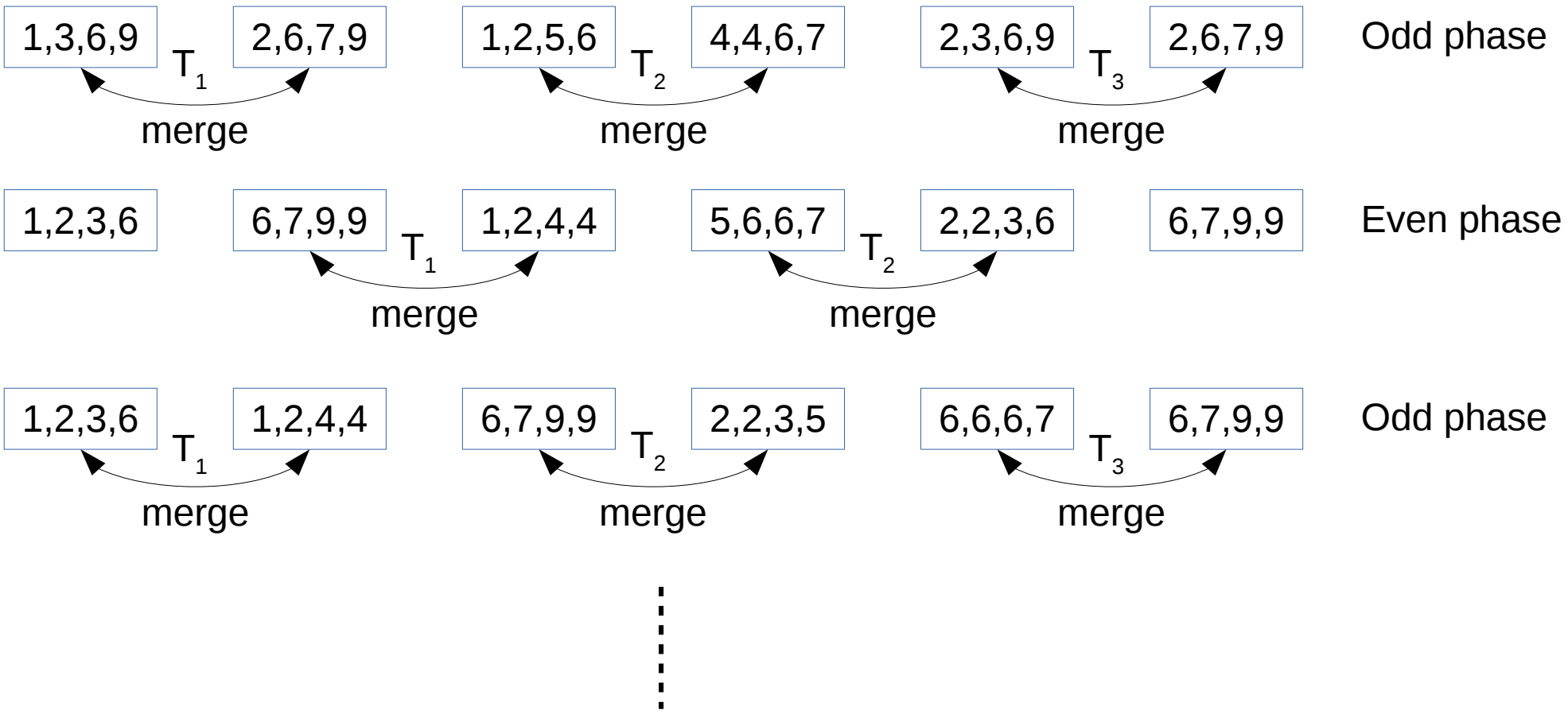
6,3,9,1	9,7,2,6	2,1,6,5	7,6,4,4	2,3,9,6	7,9,2,6

  - Initially, each thread sorts two buckets

1,3,6,9	2,6,7,9	1,2,5,6	4,4,6,7	2,3,6,9	2,6,7,9
T_1	T_1	T_2	T_2	T_3	T_3

  - Iteratively and in parallel merge adjacent buckets

Additional exercise - sorting



- Use barrier to synchronize threads between phases