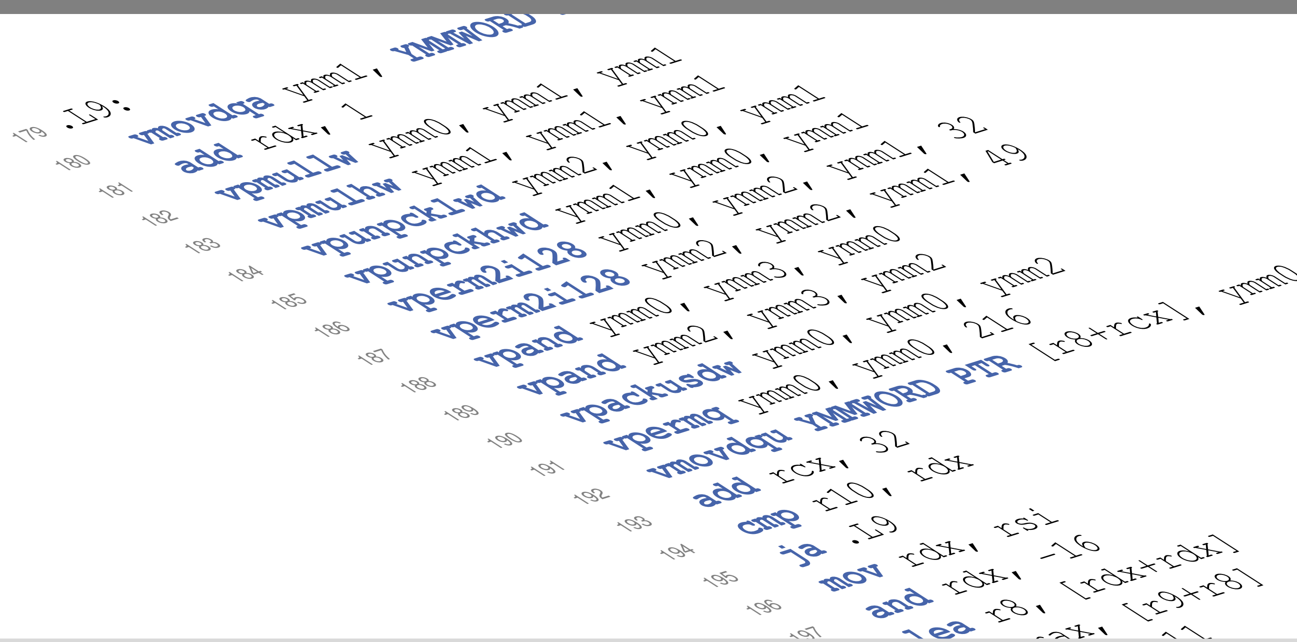
Karlsruhe Institute of Technology

# Efficient Parallel C++
# A practical course

**Tobias Maier, Sascha Witt**
Winter 2021/22

Institute of Theoretical Informatics · Algorithmics Group



# OpenMP

## Advantages

- Widely supported standard
- Portable
- Easy to use

## Disadvantages

- Fixed model of execution (fork-join)
- Some loss of control

# OpenMP Pragmas

Karlsruhe Institute of Technology

## Parallel region

```
#pragma omp parallel
{
    // parallel code
}
```

## Parallel loop

```
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    // parallel loop
}
```

## Shared and private variables

```
#pragma omp parallel private(x, y) shared(z)
{
    // parallel code
}
```

# `std::thread`

**Advantages**

- Standard since C++11
- Portable
- Control over thread lifetimes

**Disadvantages**

- Less out-of-the-box functionality
- Easier to get wrong

# Creating Threads

Karlsruhe Institute of Technology

## Single thread

```
std::thread t{[] { std::cout << "Hello,␣world!\n"; }};
// do other stuff
t.join();
```

## Multiple threads

```
std::vector<std::thread> threads;
for (int i = 0; i < num_threads; ++i)
    threads.emplace_back([i] {
        std::cout << "Hello␣from␣thread␣" << i << "!\n";
    });
// do other stuff
for (auto& t : threads)
    t.join();
```

## `std::async`

- Forget about it until you *really* understand what it does.

# Thread Pools

## No standard support yet, need to roll your own

```
class Worker {
 public:
    Worker() : thread_([this] {
        while (!stopped()) {
            // wait for a new task to execute and run it
        }
    }) {}

    // ...

 private:
    std::thread thread_;
    std::function<void()> task_;
    // ...
};
```


Institute of Theoretical Informatics
Algorithmics Group

# Concurrent Access

**Never simply access data that may be concurrently modified.**

- All data is shared by all threads by default.
- You can use `thread_local` to make a private copy for each thread.

**Use synchronization when accessing data that may be modified.**

- Prefer atomics if possible.
- Use locks when you need to modify larger chunks of data.
- You also need to use synchronization when *reading*.

# Atomics

## Required to avoid data races under concurrent writes

```
std::atomic<int> x(0);
x += 2;
// _almost_ the same as
x.fetch_add(2);
```

## Compare-and-swap operations

```
std::atomic<int> x(0);

int old_value = x;
do {
    int new_value = do_something_with(old_value);
} while (!x.compare_exchange_weak(old_value, new_value));
```

# Mutexes

## Heavy-duty locking

```
std::mutex mtx;

// ...

{    // acquire exclusive access
     std::lock_guard lock(mtx);

     // do stuff

}    // auto-release at end of scope
```

## Use `std::lock` to avoid deadlocks

```
std::lock(mtx3, mtx1, mtx2);

// do stuff

// don't forget about unlocking!
// or use std::lock_guard with std::adopt_lock
```

# Waiting for other threads

## Spinlocks: Quick-and-dirty busy waiting

```
std::atomic_flag flag = ATOMIC_FLAG_INIT;
// ...

while (flag.test_and_set(std::memory_order_acquire)) {}
// do stuff
flag.clear(std::memory_order_release);
```

## Condition Variables

```
std::condition_variable cv;

std::unique_lock lk(mtx);
cv.wait(lk, [&] { return ready; });
// ...

{   std::unique_lock lk(mtx);
    ready = true;
}
cv.notify_one();
```

# Cache Effects

**Some caches are shared among cores and sockets**

- L1 and L2 is usually private to each core (but shared by hyper-cores!).
- L3 is usually shared by all cores on a socket.
- RAM is obviously shared by all sockets.

**Shared caches can be good**

- If multiple cores work with the same data, only one has to pay the cost of loading it into the cache.
- Only really works if you don’t modify anything.

# False Sharing

Karlsruhe Institute of Technology

### Cache misses are expensive

- If you change a cache line, everybody else has to throw it away.
- That means they have to reload it before they can access it again.

### Put unrelated, frequently-updated data on different cache lines

```
struct alignas(64) SharedData {
    // ...
};
std::vector<SharedData> vec;
```

### . . . or even on different pairs of cache lines

```
struct alignas(128) SharedData {
    // ...
};
```

# NUMA and HT

### Non-Uniform Memory Access

- Only appears in multi-socket machines.
- Each socket hast fast access to some part of the main memory.
- When accessing other parts, latency and/or bandwidth is worse.
- There are libraries that help manipulating where memory is located.
- You probably don’t need to worry too much about it for this course.

### Hyper-Threading

- Commonly supported by all modern processors.
- One core may “simultaneously” execute two threads.
- Execution is interleaved to fill dead time (due to, e.g., memory latency).
- Usually leads to some speedup, but sometimes hurts overall performance.

# Thread Pinning

**Limiting which cores can execute certain threads**

- Usually, the operating system can move your threads around as it wishes.
- Sometimes, this can lead to bad performance.
- Especially on NUMA systems, pinning threads to sockets or even specific cores can be a good idea.
- Can also lead to worse performance if not done carefully.

# Guidelines

**When writing parallel code:**

- **Always** be thread-safe: Use atomics, or locks if necessary.
- Use spinlocks if the expected waiting time is low, otherwise use mutexes (even better, but more complicated: spin for a while, then switch to a mutex).
- Think about cache effects (especially false sharing).
- Multi-socket scenarios are complicated.