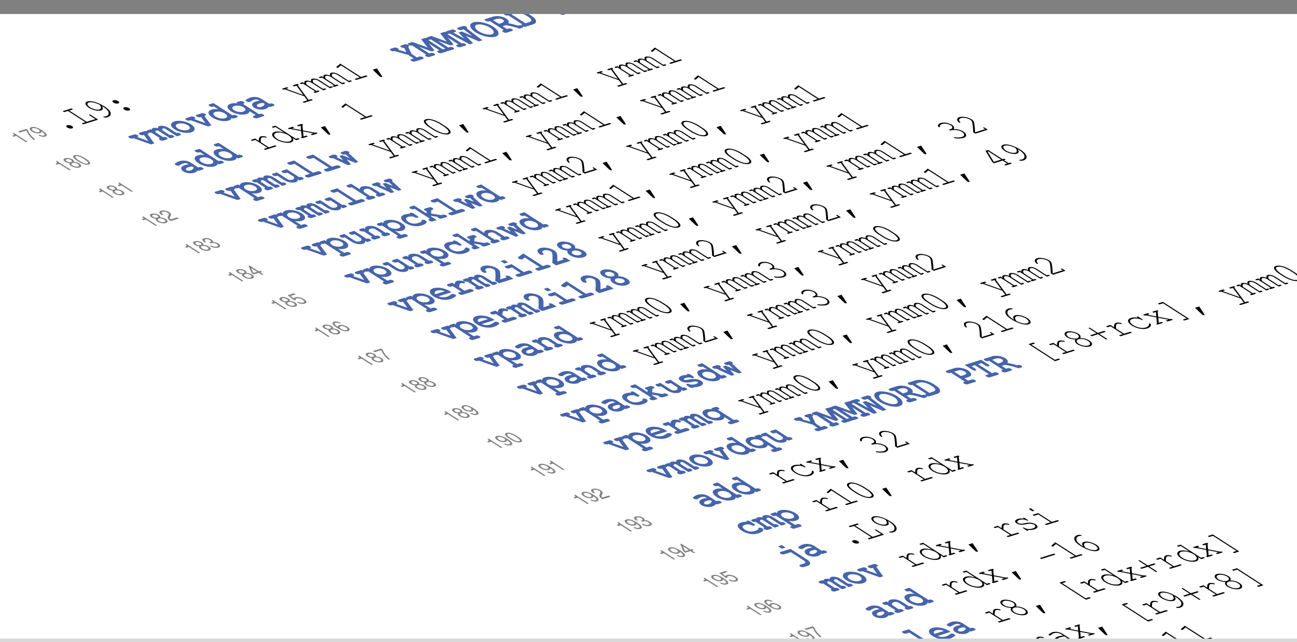
Karlsruhe Institute of Technology

Efficient Parallel C++
A practical course

Tobias Maier, Sascha Witt

Winter 2021/22

INSTITUTE OF THEORETICAL INFORMATICS · ALGORITHMICS GROUP



Goals

After this course, you should...

- be able to write efficient, parallel, clean, maintainable, testable C++
- know what and how to optimize
  - Testing and profiling
  - Cache efficiency, branch mispredictions, data structures,...
- be aware of common performance pitfalls
  - Copies, indirection, virtual functions,...
- know about common pitfalls of parallelization
  - False sharing, excessive synchronization,...

Cheat sheet available (we'll come back to that)

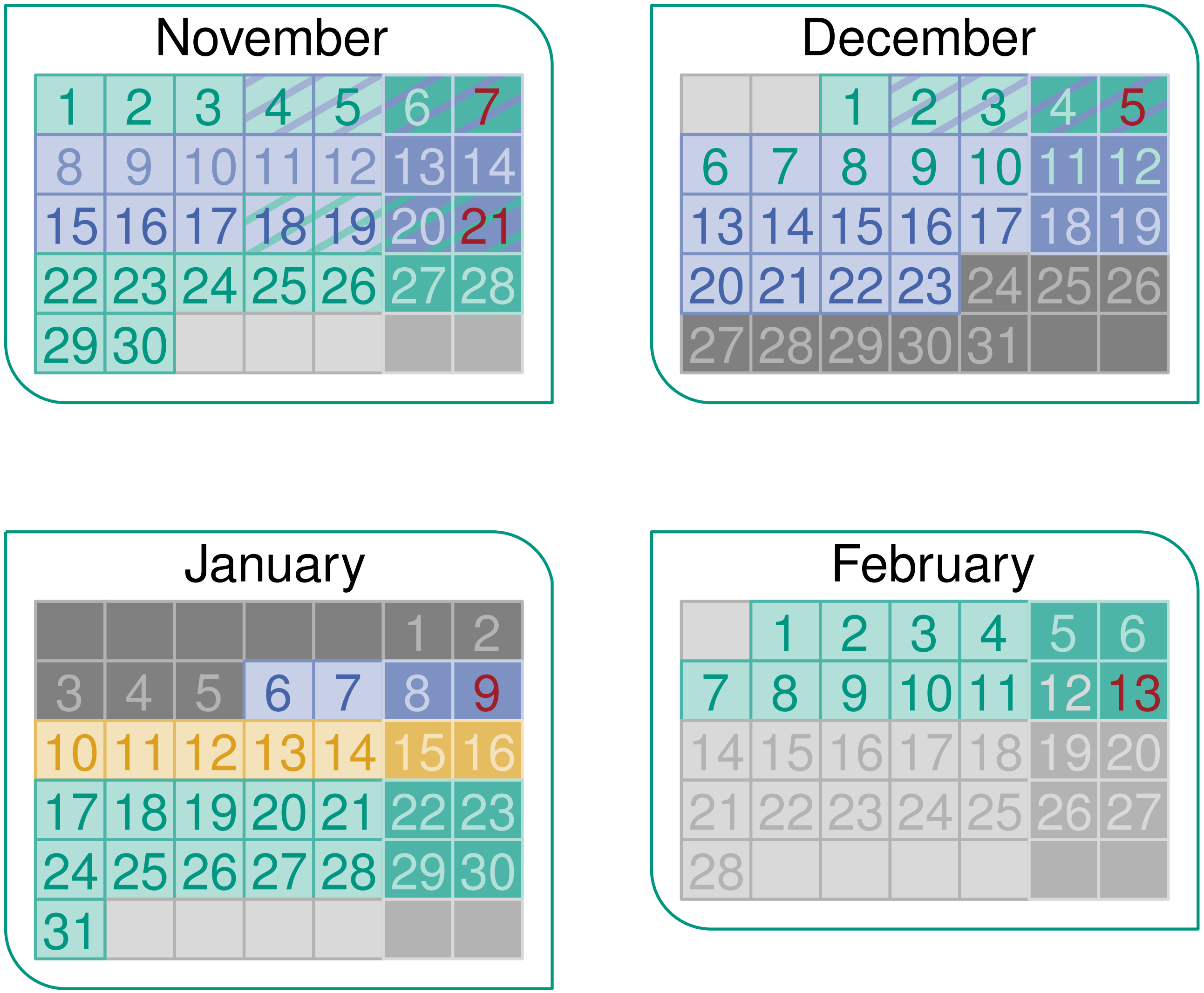
Exercises

There will be 5 exercises:

1. Common bottlenecks, templates	2 weeks
2. Cache efficiency, testing, profiling	2 weeks
3. Parallelization using OpenMP	2 weeks
4. Parallelization using `std::thread`	3 weeks
5. Personal project	$1 + 4$ weeks

- Processing times overlap by half a week.
- Exact topics are subject to change.
- You will need $\frac{1}{3}$ of the points of *each* exercise to pass.
- Turn-in consists of code, experimental evaluation and a short interview where we ask about your work.

Schedule



Tools

`https://git.scc.kit.edu/ITI10/efficient-parallel-cpp`

We use Git

- to distribute the exercises
- for turn-in
  - Fork the repo, then give *us* access to *your* repo.
  - Do not check in large files (i.e. graph data)!

Performance graphs

- R (e.g., *R for Data Science* [`http://r4ds.had.co.nz/`])
- Python (e.g., *Matplotlib* [`https://matplotlib.org/`])
- Gnuplot
- ...

Code quality

- clang-format
- clang-tidy
- ...

The most important rules

Always measure.

- It's not an optimization if it doesn't make your code faster.

Focus.

- In inner loops, a single instruction can make a huge difference.
- A function that accounts for only 3% of your running time cannot give you more than a 3% improvement.

Know what the compiler can and cannot do for you.

- And then verify that it is actually doing what you expect it to do.
- Compiler Explorer [`https://godbolt.org`] can be helpful.

General performance issues

Unnecessary copies

- Difference between *object*, *reference*, and *pointer*
- Call by *value*, *reference*, or *pointer*
- Pass small types by value.
- Prefer passing large types by *(const)* reference.
- move semantics are another way to reduce unnecessary copies.

Indirection

- Indirection often leads to cache misses.
- cf. `std::list` vs `std::vector`

Tobias Maier, Sascha Witt

Institute of Theoretical Informatics
Algorithmics Group

General performance issues

Heap allocation

- It's slow.
- It adds indirection.
- cf. `std::array` vs `std::vector`

Virtual functions

- vtables are a level of indirection.
- Virtual function calls generally cannot be inlined.
  - The `final` specifier can help with this.
- Templates can provide similar functionality, at compile time.

Problem-specific performance issues

Karlsruhe Institute of Technology

Cache efficiency

- Memory access is often a bottleneck.
- Things that can improve locality of reference include:
  - Less indirection
  - Linear (or predictable) access patterns
  - Different data representation (e.g., Structure of Arrays)

Branch mispredictions

- Modern CPU pipelines are long, and clearing them is expensive.
- Things that can help the branch predictor include:
  - Avoiding (data-dependent) branches
  - Vectorization

General parallelization issues

Data races

- Just don't.
- Use atomics, or locks where that is not possible.

False sharing

- Sometimes, not sharing is caring.
- one cache line has 64byte this is the minimum offset between two objects to avoid false sharing (`std::hardware_destructive_interference_size`).

Unnecessary synchronization

- Minimize shared data and communication.
- Work on local data, then combine in a separate step.

Problem-specific parallelization issues

Use local approximations instead of global data.

- Think about what you need to know locally about global data.

Distribute data efficiently.

- This is especially important on Non-Uniform Memory Access (NUMA) architectures.

Distribute work efficiently.

- Equal amounts of data do not imply equal amounts of work.
- Some workloads can be highly asymmetrical.