Objectives:
- Explain necessity of parallel/multithreaded algorithms.
- Describe different forms of parallel processing.
- Present commonly used architectures.
- Introduce a few basic terms.

Comments:
- Try to relate to the students environment:
  - Student laptop most likely has multi-core CPU.
  - Most gaming consoles, smartphones, tablets have at least a dual-core CPU.
  - Lab: machines could work together in a distributed manner.

Outcomes:
  1.1 Explain why learning about multithreaded algorithms is important.

---

Objectives:
- Develop simple parallel algorithm out of a sequential one.
- Introduce the notion of barriers. Need to synchronize threads.
- Amdahl’s Law.
- Use mutual exclusion to access shared resources (safety/critical sections/locks).
- Data races and deadlocks / Granularity and load balancing.

Comments:
- This quote is as relevant today as it was back in 1972 when computers were still in their infancy.
- Nowadays, computers are even more complex. Dealing with multi-core or many-core architectures requires an additional leap in programming.
- Moving from sequential programming to parallel programming requires a new and often unnatural way of thinking.
- It is important for a good programmer to be “humble”.

Outcomes:
  2.2 Develop a parallel algorithm based on an easy to parallelize sequential algorithm.
  2.3 Determine the speedup of a parallel algorithm.
  2.4 Detect potential pitfalls of parallel algorithms.
Students should be able to determine that the sums as part of the vector addition can be computed independently.

For the scalar product, the products can be computed independently.

Do not go into details on the issues of the scalar product (the summation of the products) as they are introduced later on.

Similar to the divide and conquer technique, the problem is divided into smaller (independent) subproblems.

The smaller subproblems are computed in parallel, instead of divide and conquer that is using recursion.

The operations (addition and assignment to result vector) on respective vector entries are completely independent of other vector entries.

Dividing into equal sized parts ensures that each one of the threads takes a similar amount of time for the computation.

\texttt{barrier}.\texttt{wait} is used to synchronize the threads. The algorithm has to wait for all threads to finish their computation before it returns the result vector.

Notation on how to use barriers. Barriers always need a parameter: the number of threads a barrier does synchronize.

The vector addition only has a parallel part. There is no need to combine the results of the subproblems. The sequential part is therefore empty.

The computation of the individual parts might differ even though they are equal sized.

The running time of the parallel part is reduced to 10 seconds in case we are using 4 threads.

The running times for the sequential parts stay the same.

Speedup of 1.6 for the left program (with 50% parallel code).

Speedup of 2 for the left program (with 66.6% parallel code).
For an infinite number of threads the running time for the parallel part would be computed instantly (0 seconds).
The use of infinitely many threads gives us an upper bound on the maximum speedup that can be achieved.
Only the running times for the sequential parts count.

Students should compare the scalarproduct computation to the vector addition.
Students should answer the question correctly with "NO".
Students should be able to explain the answer.
The update $c$ in all threads is a problem that could cause significant problem in the form of incorrect results.
Some sort of protection is necessary to allow a safe update of $c$.

Summarize and complete the answers of the students.
The interleaving and therefore the result of the scalarproduct might change from execution to execution.
Idempotent means that the performed update operation does not have an effect on the query that is performed by the other thread. It therefore does not matter if the second thread uses the value before or after the update for its query.

Two simple program fragments running in parallel do showcase problems that might arise through an unfortunate interleaving.
Both threads read a shared variable into a local variable and later use the local variable to update the shared variable.
For the "OK" example the read and the update are grouped and separated from each other.
If the read and update of both threads are interleaved then the update of one thread might cause an unexpected behavior within another thread that is under the impression that the correct value of shared is stored in its local variable.
Difficult to reproduce does not necessarily mean that you have a different result for every execution.

The data race could potentially happen in very few cases and therefore might be overlooked.

Data races that lead to different results in every execution are often easy to find and fix.

Pthreads are often used for C projects.

The functionality that Boost is offering for programming multithreaded algorithms will be integrated into future versions of C++ as part of a new standard. Most compilers do not yet completely support this upcoming standard.

boost::thread thrd(&hello)
creates a thread object and binds the function hello to it.

The thread and therefore the function hello is immediately executed.

The instruction thread.join() waits for the thread (the function hello) to finish.

A more advanced hello world example.

10 threads are created and bound to the hello function which now has an int t argument.

boost::bind(&hello, i) allows us to bind the function with the specific argument i to the thread object.

This bind function supports up to 9 arguments.

The threads are immediately executed.

The loop of thrds[i]->join() instructions lets the main function wait until all threads are finished.

Free the allocated storage at the end.
Using printf is thread safe meaning that it can safely be called in parallel.
Using cout is not thread safe in our setting because the statement consists actually of three output parts (after each `<<` operator)
Consequently, the output in case of printf is not garbled yet it is for cout.
The order of the numbers might differ in each run (for printf and cout).

- Summary of the `boost::thread` functionality.

- Barrier functionality is implemented by `boost::barrier`.
- Lock functionality (critical section) is implemented by `boost::mutex`.

- Dynamic balancing might also be useful for the parallel multiplication of sparse matrices that usually have a different representation.
- The representation of an object can be responsible for the choice of dynamic or static balancing.
Transfer the parallel implementation for the matrix multiplication to a parallel EXTEND algorithm.

Try to apply the technique used to efficiently parallelize the EXTEND algorithm to the Floyd-Warshall algorithm.

The analysis of the for-loop reveals that the outer loop cannot be split-up into distinct sections due to dependencies.

Start with the outer loop and check for dependencies and subsequently move to the inner loops to find parts of the algorithm that can be parallelized.

The technique to parallelize EXTEND is used on the second for-loop of the Floyd-Warshall algorithm.

The inner loop could be parallelized too, but it would result in a more fine-grained solution with smaller tasks. Additional synchronization would be required.

