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.

M-1 — Multithreaded Algorithms – Sorting Problem
- Students should know insertion sort and merge sort at this point.
- Students might have heard of Heapsort or Quicksort.
- It is important for the students to make the connection from divide and conquer to parallel algorithms.
- The students should have developed an intuition why this works.
- The creation of subproblems is the key. The subproblems are independent!

M-2 — Multithreaded Algorithms – Divide and Conquer
- The divide and conquer does not have to stop when all threads have been assigned a subproblem.
- After each thread has solved its subproblem we need to combine the results of the individual threads.
- It might not be possible to utilize all threads for the combining of the subproblems.

M-3 — Multithreaded Algorithms – Sorting Problem
- Start with a simple parallel sorting algorithm for 2 threads.
- Even if the subproblems have exactly equal size it cannot be guaranteed that the threads finish at the same time.
- Only one thread is necessary to do the combining of the two solutions.
The S marks the parts of the algorithm that are performed sequentially.
The P marks the parts of the algorithm that are done in parallel.
The **Synchronize** prevents one of the threads to continue to start the combining while the other thread is still working on solving the subproblem.

The solution of the problem can be computed in parallel.
The running time of the subproblems might differ.
The algorithm has to wait for the slowest thread.
The **MERGE** has to be performed sequentially.

The running time of the parallel part is reduced to 20 seconds in case we are using 2 threads.
The running times for the sequential parts stay the same.
Speedup of 1.33 for the left program (with 50% parallel code).
Speedup of 1.5 for the left program (with 66.6% parallel code).

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).
The sequential merge sort without the merge procedure (see next slide).

The merge procedure has been slightly modified in order to eliminate the $\infty$ element. Check for the indices $i$ and $j$.

The $L$ and $R$ arrays are now parameters to avoid the malloc/free for $L$ and $R$ in every merge call.

The while loop in `p_lg_merge_reduce` goes through the different stages of the logarithmic merge reduction.

The barrier ensures that the threads participating in the current stage wait for each other.

Threads that do not participate in a stage exit the loop. The second part of while loop condition ensures the fade-out.

---

