Part 6A: Designing parallel algorithms
Three concepts

- **Computational problem**
  - specifies what we want
  - e.g.: sort n numbers

- **Algorithm** that solves it efficiently
  - tells how to solve it, on a somewhat abstract level
  - e.g.: quicksort

- Efficient *implementation* of the algorithm
  - actual C++ code that works well on real computers
  - e.g.: std::sort implementation in the GNU C++ Library
Three concepts

• **Computational problem**
  • specifies what we want
  • e.g.: sort n numbers

• **Parallel algorithm** that solves it efficiently
  • tells how to solve it, on a somewhat abstract level
  • e.g.: parallel quicksort

• Efficient *parallel implementation* of the algorithm
  • actual C++ code that works well on real computers
  • e.g.: __gnu_parallel::sort
Three concepts

• **Computational problem**
  • specifies what we want
  • e.g.: sort *n* numbers

• **Parallel algorithm** that solves it efficiently
  • tells how to solve it, on a somewhat abstract level
  • e.g.: parallel quicksort

• Efficient *parallel implementation* of the algorithm
  • actual C++ code that works well on real computers
  • e.g.: `__gnu_parallel::sort`
We need new kinds of algorithms

• Some classical algorithms have opportunities for parallelism
  • example: many “divide and conquer” algorithms

• However, often we need to design entirely new algorithms!

• Wrong question:
  “how to implement this algorithm on a parallel computer?”

• Right question:
  “how to design a parallel algorithm for this problem?”
Parallel algorithms: terminology

• “Processor”:  
  • any form of parallelism often is described as if we had $p$ processors  
  • abstraction — shows what can be done independently in parallel  
  • practical realizations: superscalar execution, pipelining, CPU vector lanes, CPU threads, GPU threads, multiple GPUs, computing cluster ...

• “Work”: total number of operations by all processors

• “Depth”: longest sequential dependency chain  
  • how long does it take even if we had infinitely many processors
Sum

• Problem: calculate sum of $X = (x_0, x_1, ..., x_{n-1})$

• Trivial sequential algorithm

• Recursive parallel algorithm $\text{sum}(X)$:
  • if $n \leq 2$:
    • use sequential algorithm
  • if $n > 2$:
    • split $X$ in two halves $A$ and $B$
    • in parallel, calculate $a = \text{sum}(A)$ and $b = \text{sum}(B)$
    • return $a + b$

Some examples:
- $A = \text{first half}$
- $B = \text{second half}$
- $A = \text{odd indexes}$
- $B = \text{even indexes}$
**Sum**

- *In theory* we could parallelize sums as follows:
  - $O(n)$ processors, $O(n)$ work, $O(\log n)$ depth

- *In practice* this shows that there is **lots of potential for parallelism**, without doing much extra work
  - *do not* try to implement the recursive algorithm directly, use it as a source of ideas of how you could reorganize computation
  - just use enough levels to fully utilize your hardware
    - e.g.: 3 levels for OpenMP, 3 levels for SIMD, 2 levels for ILP?
  - usually we don’t have $n$ “processors” but only e.g. 256
The same idea works for any **associative binary operation**:

- sum
- product
- max
- min
- bitwise and, or, xor
- matrix multiplication ...
What can be parallelized?

• Nobody knows yet!

• Efficient parallel algorithms exist for many problems

• Some evidence that some problems are very hard to parallelize
  • some useful keywords for further study: complexity class $\textbf{NC}$, $\textbf{P}$-complete problems, conjecture $\textbf{P} \neq \textbf{NC}$
Next

• Part 6B: *parallel prefix sum* — a concrete example of an efficient parallel algorithm

• Part 6C: *pointer jumping* — a useful algorithm technique for parallel algorithms that handle linked data structures