High Performance Programming

Lecture 1:

"Introduction to Parallel Processing"

Lecturer: Ramoni Adeogun

AALBORG UNIVERSITET

Content

1. Course introduction

Course information

- **Course Responsible:**
  - Ramoni Adeogun, WCN Section (Email: ra@es.aau.dk, Room: FrB 7A3-218)
- Teacher: Stefan Nordborg Eriksen, WCN Section (Email: sne@es.aau.dk, Room: FrB 7A3-215)

- Literatures:
  1. Ananth Grama, et. al, Introduction to Parallel Computing, Second edition.
  2. Czarnul, P. Parallel Programming for Modern High Performance Computing Systems: Programming with OpenMP, MPI, CUDA, and OpenCL. CRC Press

Others:

1. Xavier, C. (2002). Introduction to Parallel Algorithms. Pearson Education
2. Behrooz P. Introduction to Parallel Processing: Algorithms and Architectures, Kluwer Academic Publishers

Objective and learning outcomes

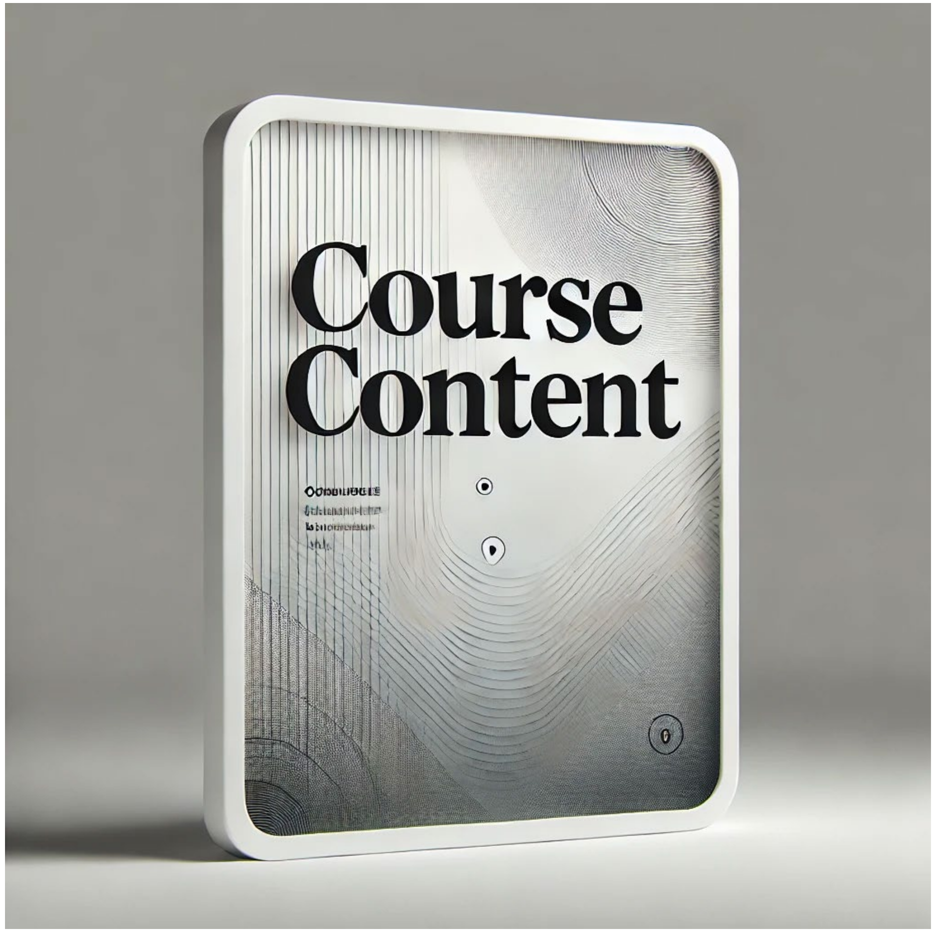
Objective: To equip students with the skills to design, optimize, and verify high-performance software solutions using parallelism, vectorization, and modern computing architectures like GPUs.

Learning outcomes (from the study regulation)

- **Knowledge**
  - data structures used to improve performance
  - basic understanding of limitations and bottlenecks in data science solutions
  - parallelism and the following issues they raise
  - vectorization of operations
  - GPU-based operations
  - types of tests and their use
  - quality measures for the correctness of computer science solutions, including: testing and verification
- **Skills**
  - can resonate and argue for bottlenecks in software programs and applications
  - can utilize parallelism in the chosen programming language and document the correctness of a given implementation
  - can use and perform tests in the development process of a program so that it is documented that its functionality is correct in a number of given cases
  - can use and perform verification of simple programs
  - can use correct professional terminology
- **Competencies**
  - can solve problems that require high performance by using parallelism in a computer program
  - can argue for the correctness of chosen solutions using tests and verification

Course content and teaching plan (tentative)

- A total of 10 lectures
  - Lecture 1: Introduction to parallel processing
  - Lecture 2: Basic data structures and communication
  - Lecture 3: Paradigms for parallel programs
  - Lecture 4: Analytical modelling of parallel algorithms
  - Lecture 5: Practical aspects of parallel programming
  - Lecture 6: Shared memory parallel programming
  - Lecture 7: Distributed memory parallel programming
  - Lecture 8: GPU programming
  - Lecture 9: Testing and verification
  - Lecture 10: Workshop



Credit: ChatGPT

Examination

- Examination will be oral.
- The focus of the examination is to assess you based on the expected learning outcomes of the course.
- The exam will cover both the fundamental theoretical concepts and practical aspects of high performance programming.
- Each student is awarded a grade on the 7-point scale.
- Exact form of the oral exam will be communicated later.

Recommendations

- Prepare for lectures by reading recommended course materials.

- Attend and participate actively during the lectures.

- Solve as many of the exercise problems to enhance your understanding of the concerned concepts.

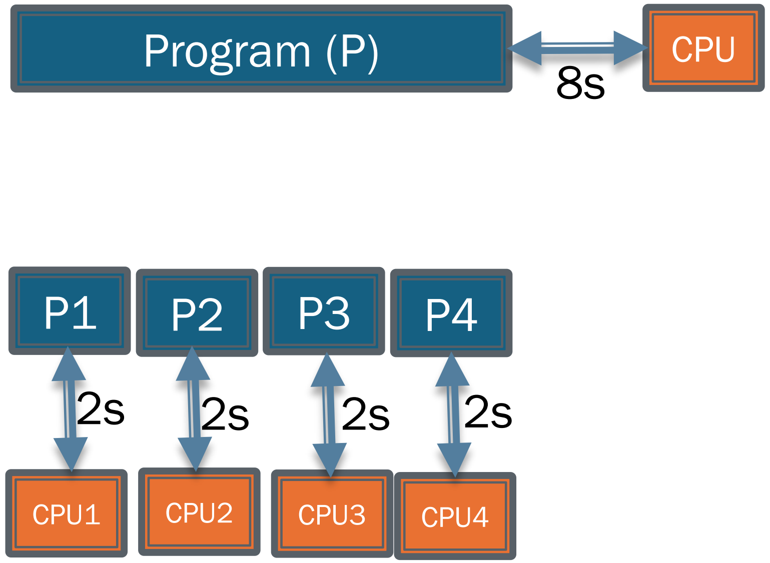
- Remember to read and understand each problem before attempting to solve it.

- **Remember**: a 5 ECTS course requires approx. **147 hours study time**.
    - Lectures and exercises constitute only about 50 hours.
    - It is your responsibility to appropriately distribute the remainder for reading course materials, preparing for lectures, and examination.

2. Introduction to parallel processing

What is Parallel Processing?

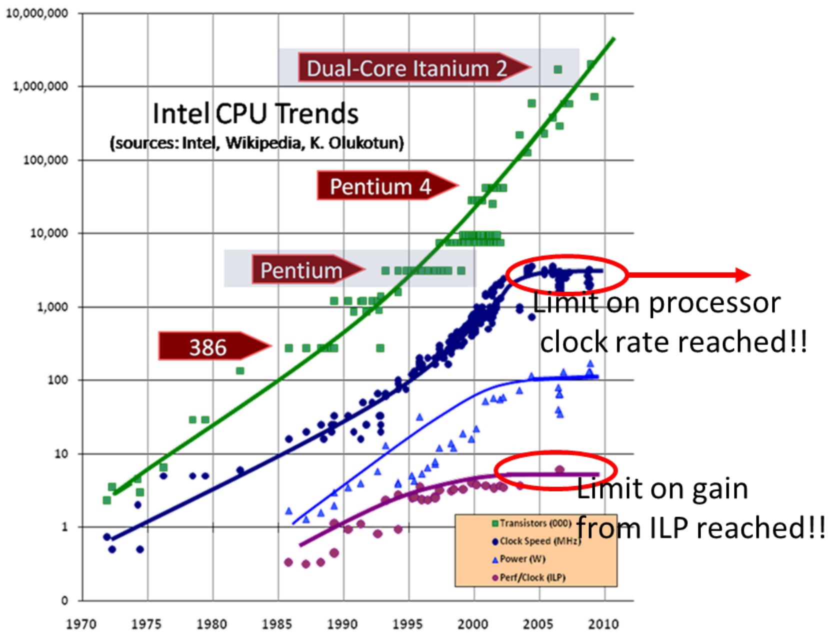
- Parallel processing refers to the **simultaneous** execution of multiple tasks or operations to increase computational efficiency.
    - dividing a problem into smaller sub-problems that can be solved simultaneously.
    - multiple processors or cores to work on parts of a problem simultaneously.
    - task decomposition and task scheduling to exploit the inherent parallelism of the problem.



Why is parallel processing important?

- Limitations of sequential computing
  - **Moore's Law** slowdown: increase in performance per core has slowed down
    - a single-core processor no longer provides the necessary performance improvements.
  - Sequential algorithms may not scale for large datasets.

- Achieving needed **performance improvement** for large computational problems in applications including:
  - Climate modeling and weather prediction.
  - Computational biology and genomics.
  - Engineering simulations (e.g., fluid dynamics, structural analysis).
  - Real-time data analytics and machine learning.



Parallel processing is the **primary way** to achieve significantly higher application performance for the foreseeable future

Why High-Performance Programming?

1. Higher speed (solve problems faster)

   Important when there are "hard" or "soft" deadlines; e.g., 24-hour weather forecast

2. Higher throughput (solve more problems)

   Important when we have many similar tasks to perform;
   e.g., transaction processing

3. Higher computational power (solve larger problems)

   e.g., weather forecast for a week rather than 24 hours,
   or with a finer mesh for greater accuracy

From Serial Algorithmic to Parallel Thinking

- So far most or all your courses has assumed that only one thing happens at a time in a program
  - sequential programming → each statement executes in sequence.

- Removing this assumption creates challenges:
  - **Programming:** How can we divide work among threads of execution and coordinate (synchronize) among them?

  - **Algorithms:** How can activities in parallel speed-up a program? → more throughput: work done per unit time

  - **Data structures:** May need to support concurrent access → multiple threads operating on data at the same time.

Short Discussion

Identify aspects of your group project(s) that may benefit from parallel processing?

What do you consider the major show-stoppers for parallelization?

Which of these aspects of parallel processing do you consider important for a computer engineer?

- Data structure
- Algorithms
- Programming

Levels of parallelism

- Levels of parallelism
    - the hierarchical nature or granularity at which computations can execute in parallel.
    - often correspond to hardware and software abstractions.

1. Bit-Level Parallelism [1970 to ~1985]
    - Exploits the parallelism inherent in the hardware to process multiple bits of data simultaneously.
        - a 64-bit processor can process more data per clock cycle than a 32-bit processor.
2. Instruction-Level Parallelism [~1985 til date]
    - overlaps the execution of multiple instructions within a single processor using techniques like pipelining, out-of-order execution, and superscalar execution.
    - ILP is intrinsic to modern CPU designs and helps improve performance without explicit parallel programming.
3. Statement-Level Parallelism
    - Focuses on executing multiple statements of a program concurrently.
        - Compiler optimizations like loop unrolling and vectorization target this level by enabling the parallel execution of statements.
4. Task-Level Parallelism
    - Involves splitting a program into distinct tasks that can run independently or concurrently.
    - explicitly controlled by the programmer using threads, processes, or distributed computing paradigms like MPI.

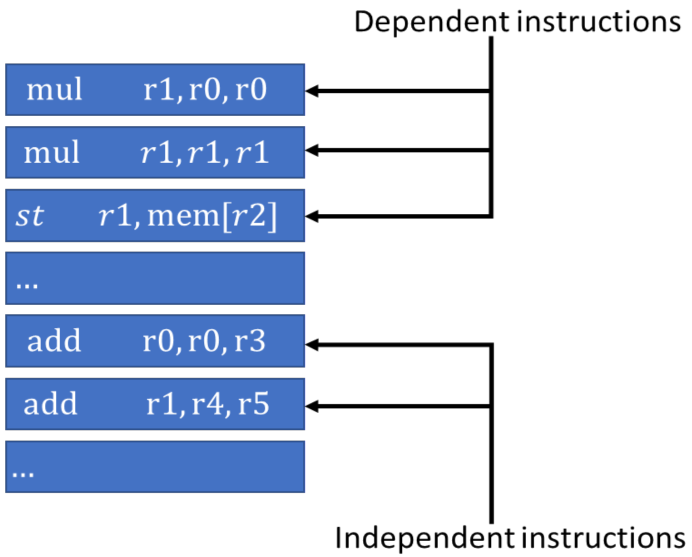
Parallelism in single-core architectures

- What does a processor do? It runs programs
  - Execute instruction referenced by the program counter (PC) → instruction execution will modify machine state: contents of registers, memory, CPU state, etc
  - Move to the next instruction → execute it....

- Processors did in fact leverage parallel execution to make programs run faster, it was just invisible to the programmer

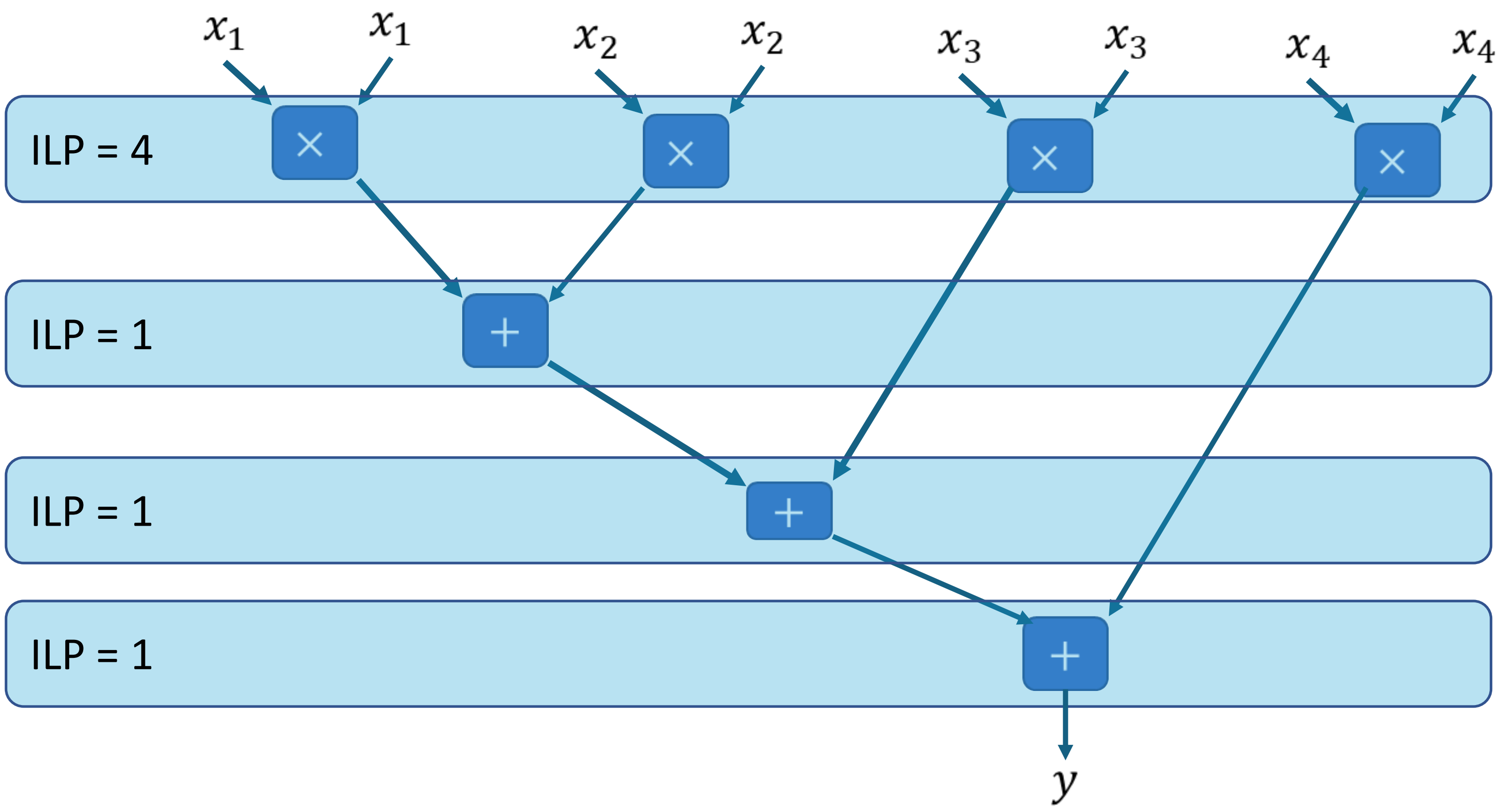
- Instruction level parallelism (ILP)
  - Instructions must appear to be executed in program order.
  - **independent** instructions can be executed
    - simultaneously by a processor without impacting program correctness

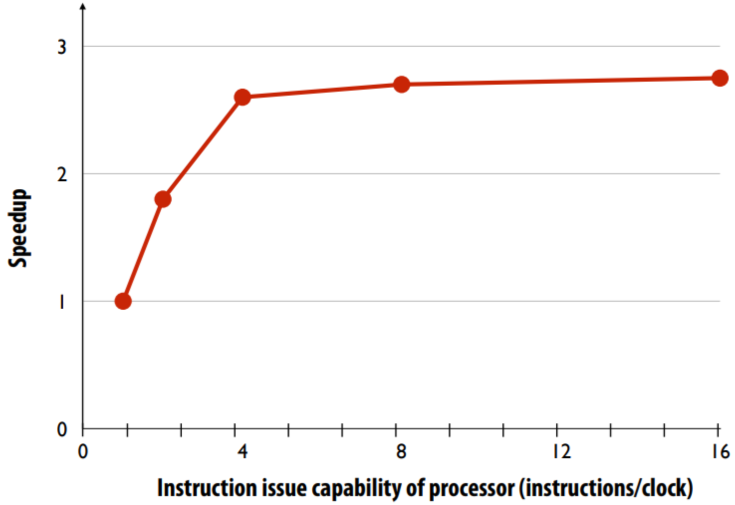
  - **Superscalar execution:** processor dynamically finds
    - independent instructions in an instruction sequence
    - and executes them in parallel



Instruction level parallelism (ILP): Example and limitations

- Instruction: $y = \sum_{k=1}^{4} x_k \times x_k$





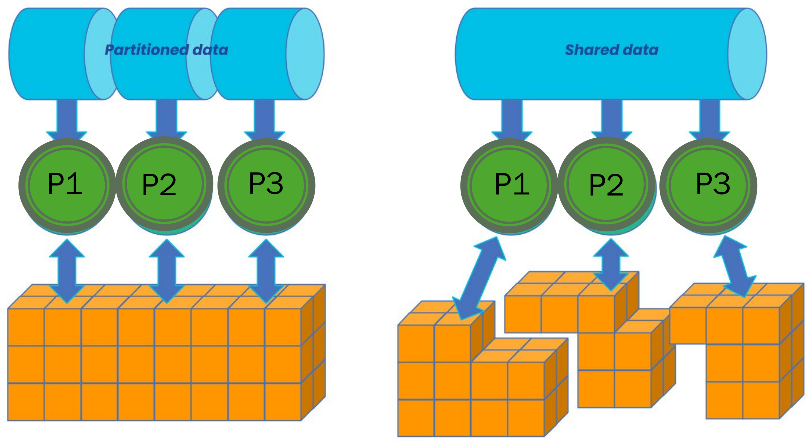
Most available ILP is exploited by a processor capable of issuing four instructions per clock → little gain from using one with more capability

Types of parallelism

- **Task parallelism**
  - Different tasks or operations run in parallel, but each task is independent.
  - Different components of a program running simultaneously (e.g., data input, calculation, and output).

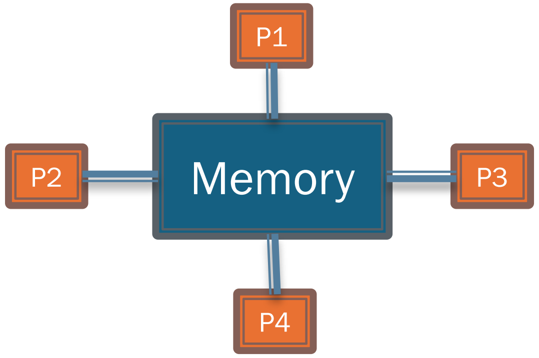
- **Data parallelism**
  - The same operation is applied simultaneously to multiple pieces of data.
  - Ex: Matrix operations where each element can be processed in parallel.

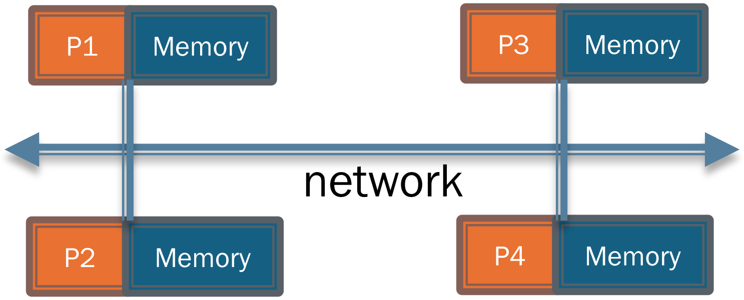
- **Pipeline parallelism**
  - Tasks are divided into stages, and different stages can be executed concurrently.
  - Ex: image processing pipeline → one part of the image is processed by one unit and the next part by another

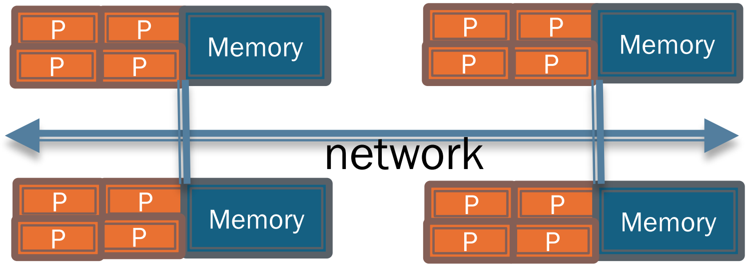


Parallel programming models

- Shared Memory Model
  - Processors share access to a common memory space, allowing them to communicate directly by reading and writing to shared memory.
    - Fast and uniform data sharing due to processors proximity to memory.
    - Global address leads to user friendly memory programming.
    - No scalability → more processors increases traffic on shared path.
    - Programmer responsibility for correct global memory access.
- Distributed Memory Model
  - Each processor has its own local memory, and communication between processors occurs via message-passing.
- Hybrid Model
  - Combines both shared and distributed memory models.







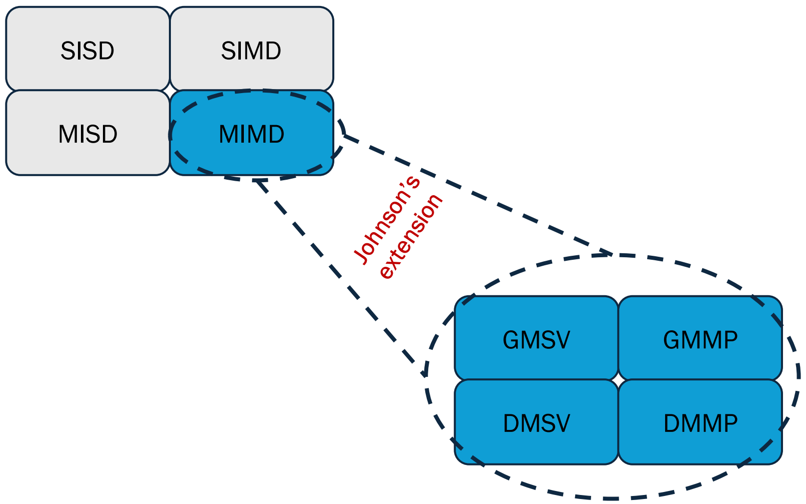
Parallel computing architectures: a taxonomy

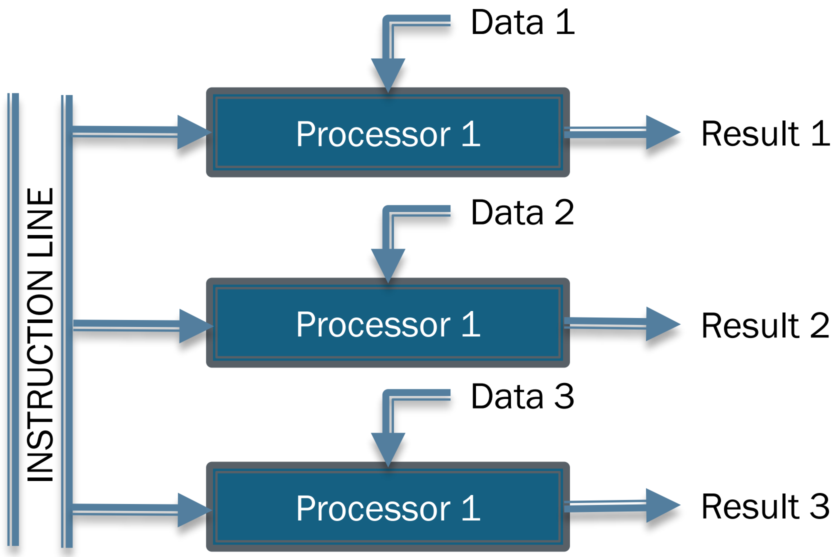
The Flynn's taxonomy

- Based on
    - Multiple/Single Instruction (MI/SI)
    - Multiple/Single Data (MD/SD)

- Main architectural classes:
    - Single Instruction Single Data (SISD)
    - Single Instruction Multiple Data (SIMD)
    - Multiple Instruction Single Data (MISD)
    - Multiple Instruction Multiple Data (MIMD)

- Johnson's extension of MIMD
    - Based on:
        - Shared variables/Message Passing (SV/MP)
        - Global memory/distributed memory (GM/DM)





Architectures for modern HPC systems

- Multicore Processors:
  - Multiple cores on a single processor chip, allowing for simultaneous execution of multiple threads.
  - Ex: Intel’s multi-core CPUs and AMD’s Ryzen processors.
- Manycore Coprocessors
  - feature hundreds to thousands of cores optimized for data-parallel operations.
  - used for massively parallel workloads, such as matrix multiplications and deep learning model training.
  - Examples include Intel’s Xeon Phi and NVIDIA’s GPUs.
- Clusters and Supercomputers
  - Large-scale parallel systems made up of many interconnected nodes, each with its own memory and processors.
  - Ex: Cray supercomputers, Google’s data centers.
- GPU Architectures
  - GPUs are specialized processors designed for parallelism, with thousands of cores to handle many computations simultaneously.
  - Ex: NVIDIA GPUs, used for tasks like matrix multiplication, deep learning, and graphics rendering.

Challenges in parallel programming

- **Concurrency Issues**
  - Race Conditions: Occur when multiple threads/processes access shared data simultaneously without proper synchronization.
  - Deadlock: A condition where processes are waiting for each other indefinitely, resulting in no progress.

- **Load Balancing:**
  - Distributing work evenly across processors to avoid underutilization and idling cores.
  - Granularity: The size of the tasks assigned to each processor can impact efficiency.

- **Scalability:**
  - As the number of processors increases, the overhead of managing them (communication, synchronization) also increases.

- **Memory Management:**
  - Ensuring that memory is used efficiently in parallel applications is critical to prevent bottlenecks.

Parallel programming tools

- OpenMP:
  - A set of compiler directives for shared memory parallelism in C, C++, and Fortran.
  - Easier to use for simple parallelism in multi-core systems.
  - Syntax and usage in C/C++.

- MPI:
  - A standard for message passing in distributed memory systems.
  - Allows programs to communicate by sending and receiving messages.

- CUDA:
  - Programming model for NVIDIA GPUs.
  - Uses GPU cores to perform large-scale parallel computations, ideal for data-intensive tasks like matrix multiplication.

- OpenCL:
  - A cross-platform framework for parallel programming on CPUs, GPUs, and other processors.
  - Supports heterogeneous computing systems.

3. Parallel computing models

Parallel Computing Models

- Parallel algorithms are designed with an assumption of an architecture of a parallel computer.
  - Different models of machines have been assumed for parallel algorithm design

- Requirements for parallel models
  - **Simplicity**:
    - A parallel model should allow easy analysis of various performance measures (speed, communication, memory utilization, etc.).
    - Results should be as hardware-independent as possible.

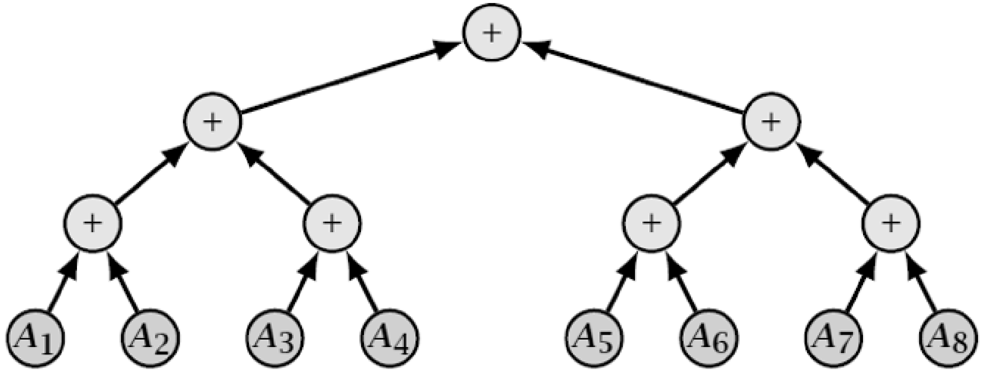
  - **Implementability**:
    - Parallel algorithms developed in a model should be easily realizable on a parallel machine.
  - Theoretical analysis should carry over and give meaningful performance estimates.

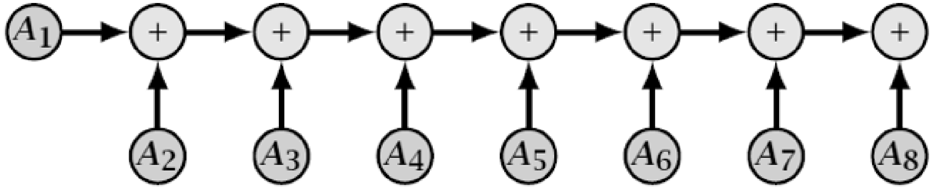
- A real satisfactory model does not exist!
  - We will study: Computation Graph; PRAM and Network models (Lecture 2)

DAG model

Directed Acyclic Graphs (DAGs) (or computation graphs) are often used to automatically parallelize numerical computations:

- nodes represent operations (single instructions or larger blocks)
  - In-degree is at most 2
  - Nodes without incoming edges correspond to input data
- edges represent dependencies (precedence constraints)
- branching instructions cannot be modelled
- completely hardware independent
- scheduling is not defined





DAG Model

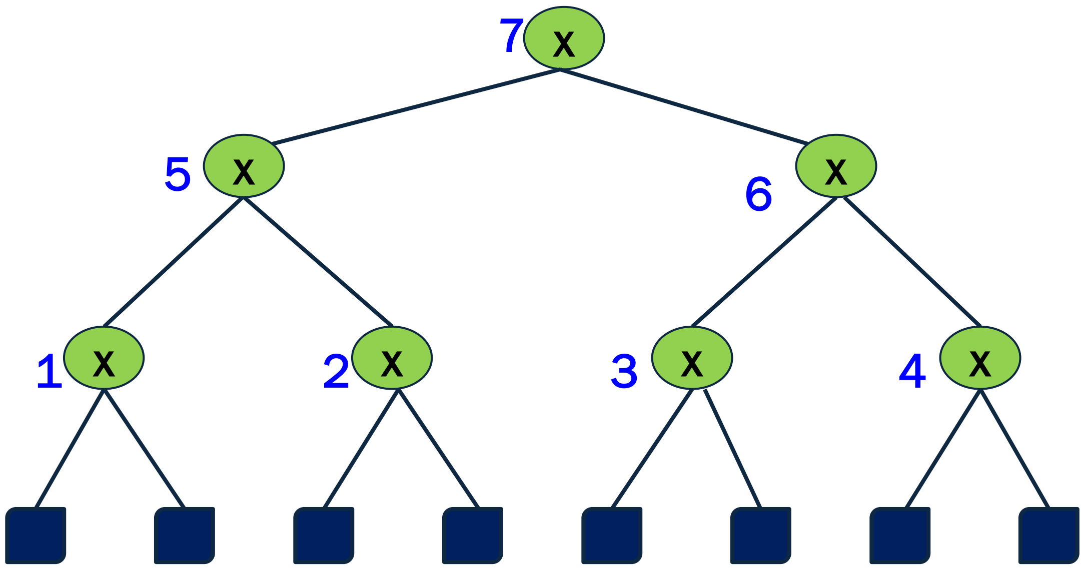
- The DAG itself is not a complete algorithm A scheduling implements the algorithm on a parallel machine by assigning a time-step, t_v and a processor p_v to every node.

- A scheduling of a DAG $G = (V, E)$ on p processors is an assignment of pairs (t_v, p_v) to every internal node $v \in V$ subject to:
  - $p_v \in \{1, \dots, p\}; \;\; t_v \in \{1, \dots, T\}$
  - $t_u = t_v \rightarrow p_u \neq p_v$
  - $(u, v) \in E \rightarrow t_v \geq t_u + 1$

Where a non-internal node x (an input node) has $t_x = 0$.

DAG Example

- Example: product of 8 numbers



- The task of multiplying 8 numbers - using 4 processors in 3 units of time
  - Complete binary tree with n nodes is of height $\log_2(n)$
  - Multiplying n numbers can be done with $n/2$ processors in $\log_2(n)$ time units.

- Task assignment to processors → scheduling: $SCH(p) := (p, t)$

Schedule	Explanation
SCH(1) = (1,1)	Processor 1 performs sub-task 1 at time 1
SCH(2) = (2,1)	Processor 2 performs sub-task 2 at time 1
SCH(3) = (3,1)	Processor 3 performs sub-task 3 at time 1
SCH(4) = (4,1)	Processor 4 performs sub-task 4 at time 1
SCH(5) = (1,2)	Processor 1 performs sub-task 5 at time 2
SCH(6) = (2,2)	Processor 2 performs sub-task 6 at time 2
SCH(7) = (1,3)	Processor 1 performs sub-task 7 at time 3

$$\text{Schedule length} = 3$$

PRAM Model

In Parallel Random-Access Machine (PRAM)

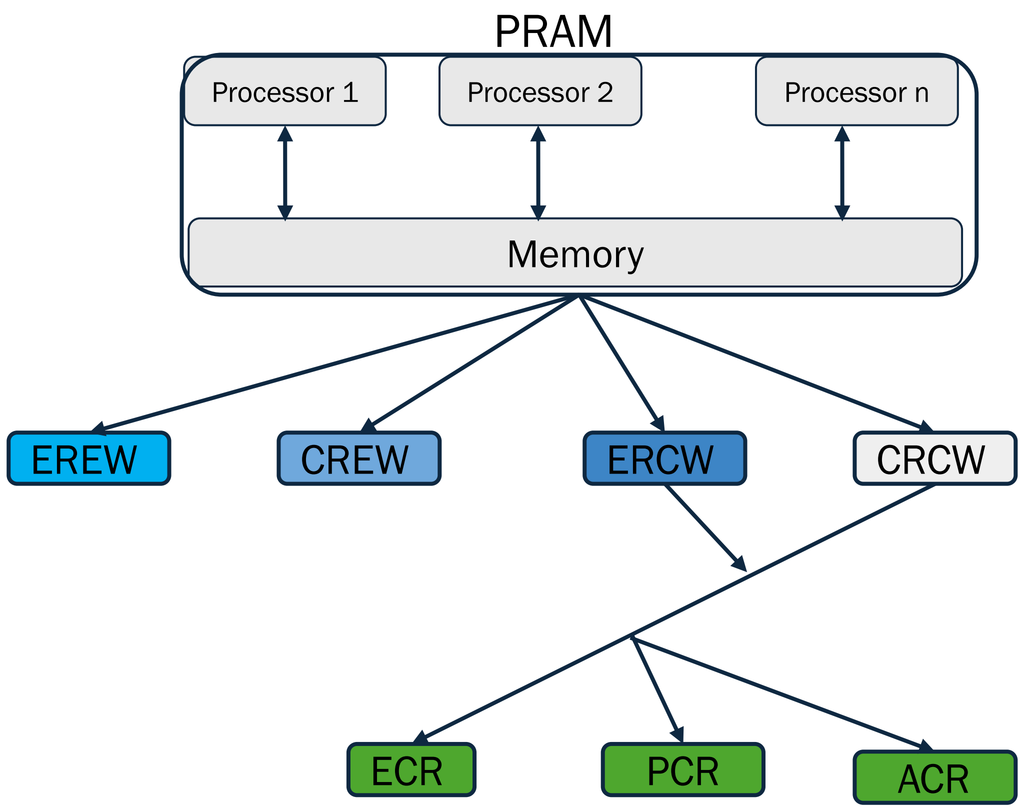
- also called **shared-memory model**
- all processors are connected in parallel to a global shared memory
  - all processors can read/write to memory
  - communications occur only via the shared memory
  - are synchronized via a common clock

PRAM model for SIMD architectures:

- Exclusive Read Exclusive Write (EREW)
- Concurrent Read Exclusive Write (CREW)
- Exclusive Read Concurrent Write (CRCW)
- Concurrent Read Concurrent Write (CRCW)

What happens if ≥ 2 processors attempt concurrent write?

- Conflict resolution, e.g.-
  - Equality/Common Conflict Resolution (ECR)
  - Priority Conflict Resolution (PCR)
  - Arbitrary/random Conflict Resolution (ACR)



PRAM – Simultaneous Read/Write Example

- Consider the following concurrent write operations on X by 3 processors

$$p_1(50 \rightarrow X), \qquad p_2(60 \rightarrow X), \qquad p_3(70 \rightarrow X)$$

What is the outcome of the following (if p_1 has the highest priority)?

- Common CRCW or ERCW:
- Priority CRCW:
- Random CRCW:
- *Sum CRCW:

Parallel processing Effectiveness

Performance metrics

- Speed-up: $S_p(n) = \frac{T_1^*(n)}{T_p(n)}$
- Efficiency: $E_p(n) = \frac{T_1^*(n)}{pT_p(n)}$
- Redundancy: $R_p(n) = \frac{W_p(n)}{W_1(n)}$
- Utilization: $U_p(n) = \frac{W_p(n)}{pT(p)}$
- Communication overhead: …… (next lecture)

What is the significance of each of these metrics for a parallel program?

Performance metrics

- How are these parallel performance metrics related?
  - $1 \leq S_p(n) \leq p$
  - $U_p(n) = R_p(n)E_p(n)$
  - $E_p(n) = \frac{S_p(n)}{p}$
  - $\frac{1}{p} \leq E_p(n) \leq U_p(n) \leq 1$
  - $1 \leq R_p(n) \leq \frac{1}{E_p(n)} \leq p$

Amdahl's Law and Scalability

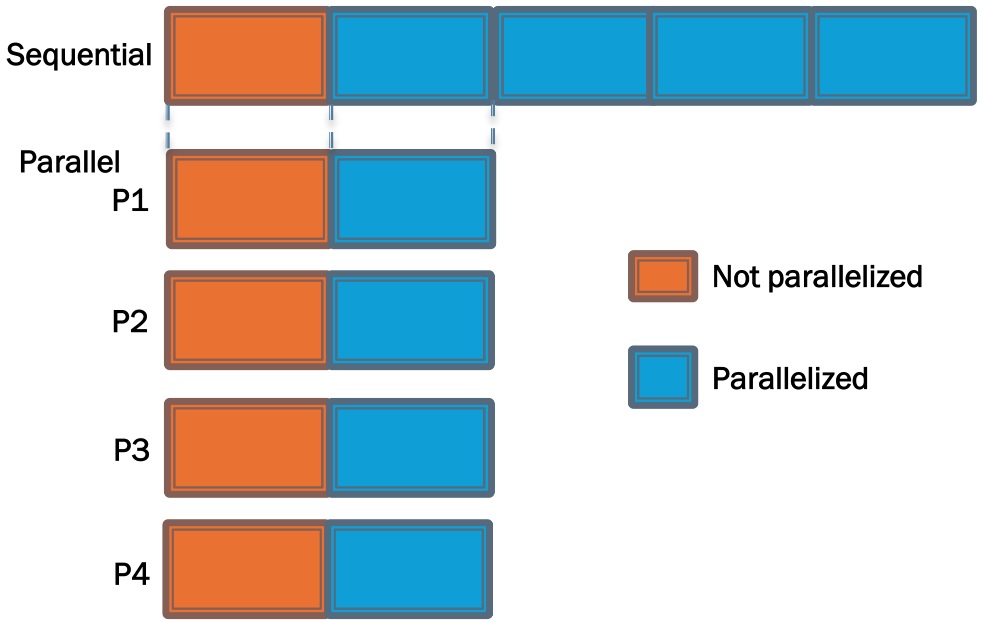
- **Amdahl's Law**
  - Describes the theoretical maximum speedup of a program when only part of the program can be parallelized.

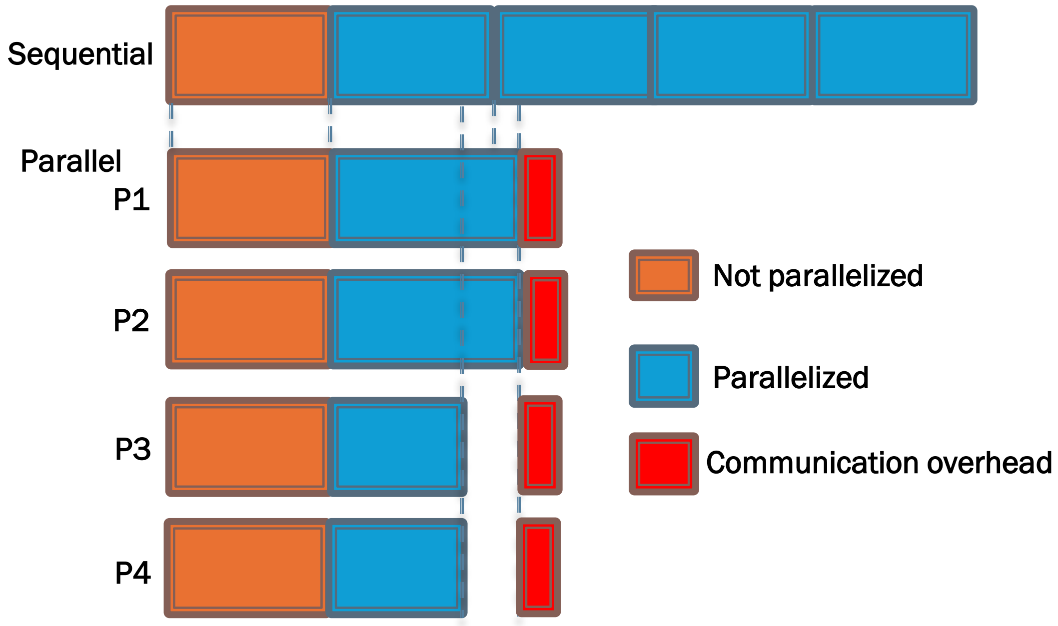
$$\text{Speedup} = \frac{1}{(1-\alpha) + \frac{\alpha}{p}}$$

    - α: proportion of the program that can be parallelized.
    - p: number of processors.
  - Even if you have many processors, the sequential part of the program will limit the overall speedup.
- **Scalability:**
  - The ability of a parallel program to effectively utilize additional resources.
  - Superlinear scalability can occur if communication overhead descreases with more processors.

Amdahl's Law – examples

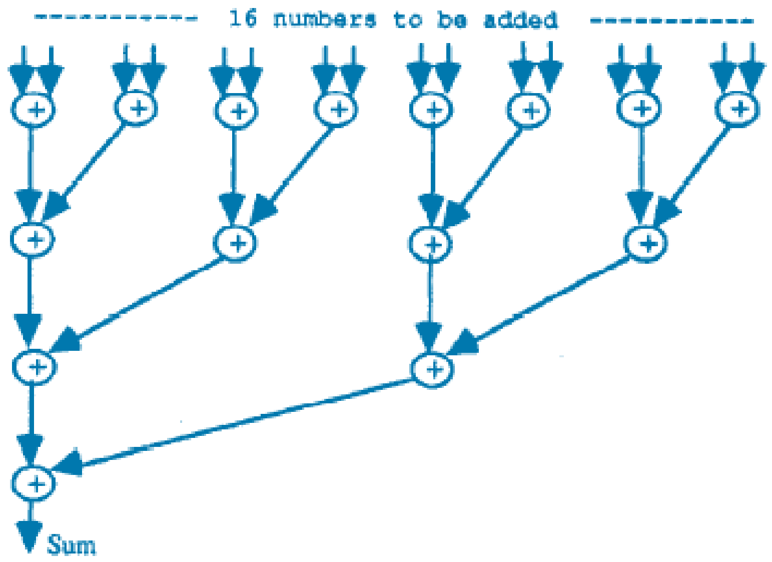
- Execution of Program A using $p = 4$ processors
  - 80% can be parallelized → 20% cannot be parallelized
- Parallel running time: $(1-0.8)+\frac{0.8}{4}=0.4 \rightarrow 40\%$ of the sequential execution time.





Example: Sum of 16 numbers with 8 processors

- Example: Sum of 16 numbers with $p = 8$.



- Assuming unit-time additions and ignoring all overheads:

$$W_8(16) = \ \dots\dots\dots;$$

$$T_8(16) = \ \dots\dots$$

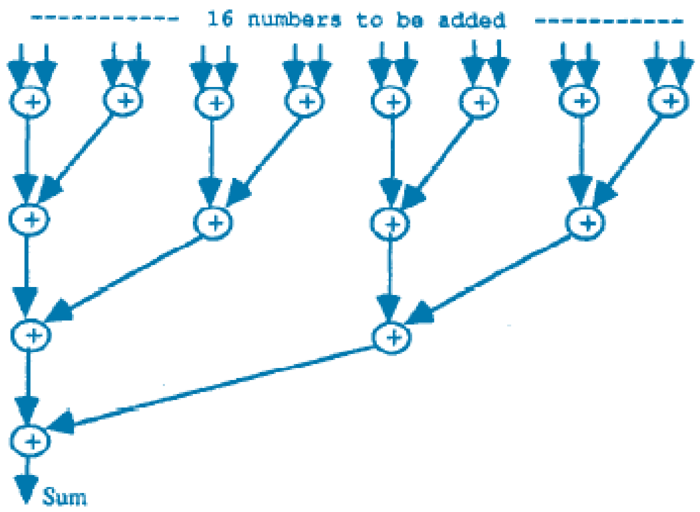
$$E_8(16) = \ \dots\dots\dots$$

$$S_8(16) = \ \dots\dots\dots$$

$$R_8(16) = \ \dots\dots\dots.$$

Example: Sum of 16 numbers with 8 processors

- New assumption:
  - Vertically alligned operations are performed by the same processor
  - Each interprocessor transfer requires one unit of work (time)



- Performance:

$$W_8\,(16) = 22$$

$$T_8\,(16) = 7$$

$$E_8\,(16) = \;\; 15/(8 \times 7) = 0.27 \Rightarrow 27\%$$

$$S_8\,(16) = \;\; 15/7 = 2.14$$

$$R_8\,(16) = \;\; 22/15 = 1.47$$

- Why the difference? Which solution is more practical?

Summary and Exercises

Summary

- Today, single-core performance is improving very slowly
    - To run programs significantly faster, programs must utilize multiple processing elements
        - you need to know how to design parallel algorithms and write parallel code

- Developing parallel algorithms and writing parallel programs can be challenging
    - Requires problem partitioning, communication, synchronization
    - Knowledge of machine characteristics is important
        - Performance may vary significantly from machine to machine

- Different models exist for studying parallel machine characteristics
    - Computation graph
    - Network models - Chain, ring, mesh, hypercubes, etc (next lecture)
    - PRAM - global shared memory model
        - unrealistic in practice due to lack of uniform memory access; global synchronization near impossible
        - good for understanding parallel techniques but not for algorithm development

- Measuring performance of parallel programs:
    - Speedup, efficiency, redundancy, utilization, communication cost (next lecture)

Exercises

- Exercise problems are available on Moodle.