

Parallel Algorithms and Programming Performance and Challenges

Thomas Ropars

Email: thomas.ropars@univ-grenoble-alpes.fr

Website: tropars.github.io

In this lecture

- Measuring performance of parallel programs
 - Beyond execution time
- Challenges of parallel programming
 - 100% efficiency is not always achievable

References

- The lecture notes of F. Desprez
- The lecture notes of K. Fatahalian
 - [CS149: Parallel Computing @Stanford](#)
 - [15418: Parallel Computer Architecture and Programming @CMU](#)
- The teaching material of the eduWRENCH project
 - [Pedagogic modules](#)

Performance of parallel programs

Execution time of non-interactive programs

Non-interactive programs

- We consider programs whose main purpose is not to interact with a user
 - It implies that the execution time does not depend on the user activity
 - Simplify reasoning about performance

Execution time

- **Work:** The amount of computation to be executed by a program
- **Compute speed:** The amount of work that can be executed by the hardware per unit of time

$$\textit{Execution time} = \frac{\textit{Work}}{\textit{Compute Speed}}$$

Execution time of non-interactive programs

Non-interactive programs

- We consider programs whose main purpose is not to interact with a user
 - It implies that the execution time does not depend on the user activity
 - Simplify reasoning about performance

Execution time

- **Work:** The amount of computation to be executed by a program
- **Compute speed:** The amount of work that can be executed by the hardware per unit of time

$$\textit{Execution time} = \frac{\textit{Work}}{\textit{Compute Speed}}$$

How to measure Work and Compute Speed?

Measuring the Work (FLOP)

Multiple possible ways of measuring the work

- Application specific (can be high level)
 - Number of images to process
 - Number of items to sort
 - etc.
- At the level of instructions
 - Number of instructions to execute
 - Problems:
 - All instructions do not have the same cost
 - Not all instructions are *useful* (for the final result)

Floating-point operations (FLOP)

- Most compute-intensive programs are manipulating floating point numbers
- The FLOP represent the *useful* work

Measuring the compute speed (FLOPS)

Floating-point operation per seconds (FLOP/s or FLOPS)

- Can be used to evaluate the capacity of the hardware
 - Defines the peak performance of a computing system

Back to the execution time

- A program requires executing 1 TFlop
- A system can execute 10 GFlop/s
- We can estimate its execution time:

Measuring the compute speed (FLOPS)

Floating-point operation per seconds (FLOP/s or FLOPS)

- Can be used to evaluate the capacity of the hardware
 - Defines the peak performance of a computing system

Back to the execution time

- A program requires executing 1 TFlop
- A system can execute 10 GFlop/s
- We can estimate its execution time:

$$\textit{Execution time} = \frac{1 \times 10^{12}}{10 \times 10^9} = 100s$$

Other usage of FLOPS

- FLOPS can also be used to evaluate the efficiency of an algorithm on a given hardware
 - Through measurements + comparison with the theoretical value

CPI (Cycles per instruction)

Definition:

$$CPI = \frac{\textit{execution time}}{\textit{total number of instructions}}$$

- Another metric that can give an idea of how well a program is behaving on a given hardware:
 - Measures how often a processor stalls
 - For instance, can indicate a bad use of the caches

Question:

- Is it possible to achieve $CPI < 1$ on one processor core?
 - Remember ILP

Performance of parallel programs

- The execution time measures the absolute performance
 - Does not tell us if a parallel program is good
- Other metrics need to be introduced
 - Speedup
 - Efficiency
 - Scalability

Speedup

Speedup

For a sequential execution time T_s , and a parallel execution time T_p :

$$\text{Speedup} = \frac{T_s}{T_p}$$

- When executing on N computing resources, we would like the speedup to be N
 - This is in general not going to be the case

Question: Can the speedup be more than N?

-

Speedup

Speedup

For a sequential execution time T_s , and a parallel execution time T_p :

$$\text{Speedup} = \frac{T_s}{T_p}$$

- When executing on N computing resources, we would like the speedup to be N
 - This is in general not going to be the case

Question: Can the speedup be more than N?

- **Super-linear speedup**
- May happen in different cases:
 - Less instruction executed in the parallel version of the code (search)
 - Better usage of the cache/memory/storage hierarchy

Amdahl's law

The speedup of parallel code is limited by the sequential part of the code

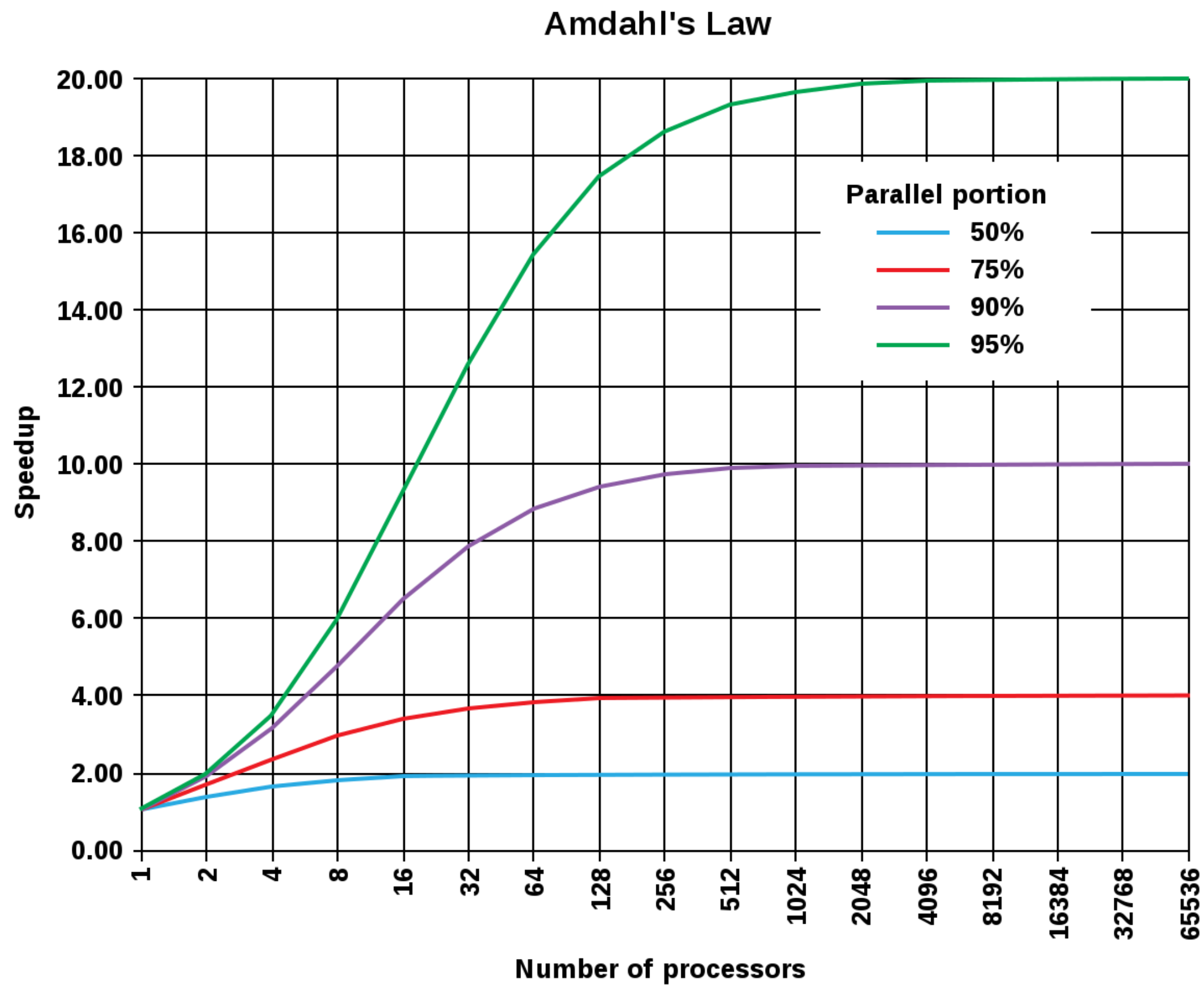
- For a program running on N computing resources, where a fraction P of the code is parallel and $S = 1 - P$ is the serial fraction:

$$Speedup(N) = \frac{1}{\frac{P}{N} + S}$$

- Hence, the maximum speedup is:

$$Speedup_{max} = \frac{1}{S} = \frac{1}{1 - P}$$

Amdahl's law



Credit: Daniels220 at English Wikipedia

Efficiency

- The efficiency measures how efficiently the computing resources are used.
- For a program running on N computing resources:

$$Efficiency(N) = \frac{Speedup(N)}{N}$$

- Ideally an efficiency of 100% would be achieved

Scalability

Scalability measures the evolution of the efficiency when the number of processors used increases.

Strong scaling: Compute a problem N times ****faster**** using N computing resources

Weak scaling: Compute a problem N times ****bigger**** in the same amount of time using N computing resources

What limits strong scaling ?

What limits weak scaling ?

Scalability

Scalability measures the evolution of the efficiency when the number of processors used increases.

Strong scaling: Compute a problem N times ****faster**** using N computing resources

Weak scaling: Compute a problem N times ****bigger**** in the same amount of time using N computing resources

What limits strong scaling ?

- Amdahl's law
- Achieving strong scaling implies minimizing the serial work

What limits weak scaling ?

Scalability

Scalability measures the evolution of the efficiency when the number of processors used increases.

Strong scaling: Compute a problem N times ****faster**** using N computing resources

Weak scaling: Compute a problem N times ****bigger**** in the same amount of time using N computing resources

What limits strong scaling ?

- Amdahl's law
- Achieving strong scaling implies minimizing the serial work

What limits weak scaling ?

- Achieving weak scaling implies ensuring that the amount of serial work and the amount of communication remains constant when the problem size increases

Some comments about scalability and speedup

- Speedup should be computed based on the most efficient sequential algorithm
 - A parallel algorithm might not perform well sequentially
- The algorithm that performs the best at small scale is not necessary the one that scales best
 - An algorithm that has a synchronization/communication cost that increases linearly with the number of computing resources: $10 \times N$
 - Another algorithm that has a synchronization/communication cost that increases quadratically: $2 \times N^2$

At the end the most important is the absolute performance

- A very slow algorithm that scales well is not interesting

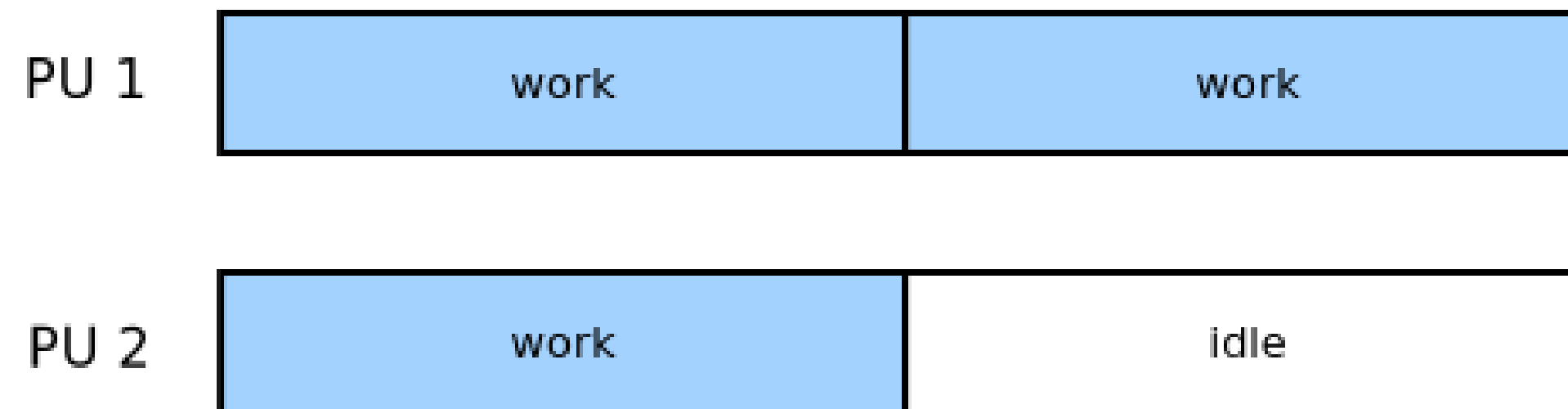
Challenges for the Performance of parallel programs

Idle time

- We saw that in general, a parallel efficiency of 100% is not achievable
 - One of the main reason is **Idle time**
- Reasons for idle time:
 - Load imbalance
 - Management of I/Os
 - Task dependencies

Load imbalance

Load imbalance describes a situation where the work is not equally distributed among the processing units



- In this situation, the efficiency can be computed as a function of the idle time:

$$Efficiency(N) = 1 - \frac{\frac{\sum idle\ times}{N}}{execution\ time}$$

- Explanation:
 - We compute the average idle time per processing unit
 - We compute a ratio to the total execution time

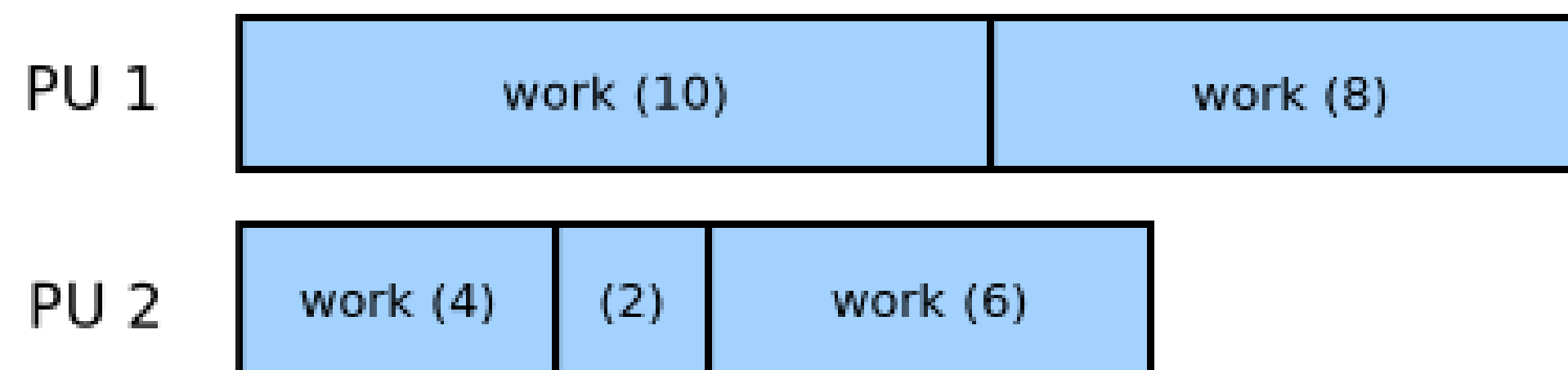
Load imbalance

Case of *identical* tasks

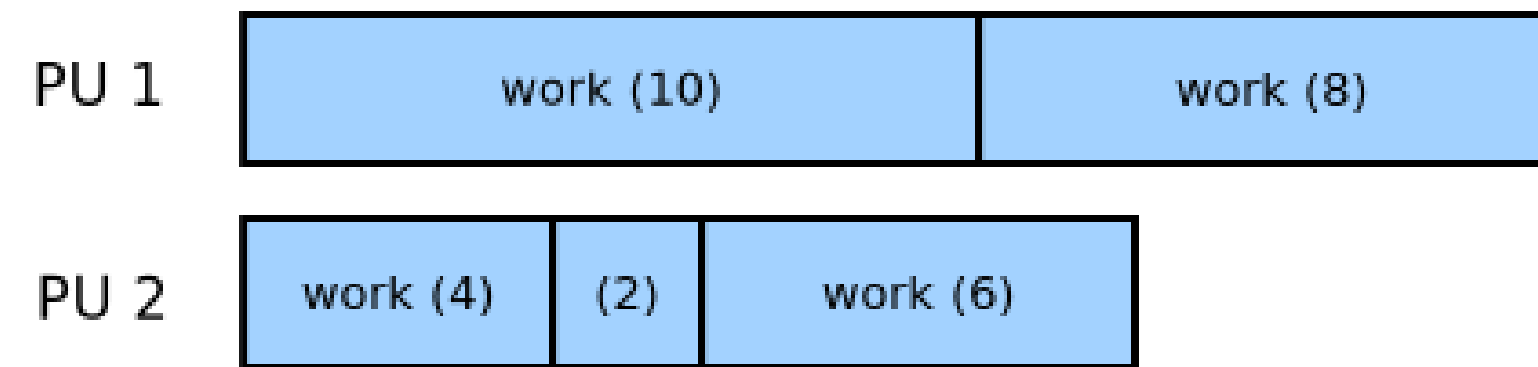
- Load imbalance can appear when the number of tasks to execute is not a multiple of the number of processing units
 - Example: 2 PUs -- 3 tasks
 - See example on the previous slide
- Here identical = takes same amount of time to execute

Case of *non-identical* tasks

- In practice, it happens often that not all tasks take the same amount of time to execute
 - In this case, load imbalance is almost unavoidable

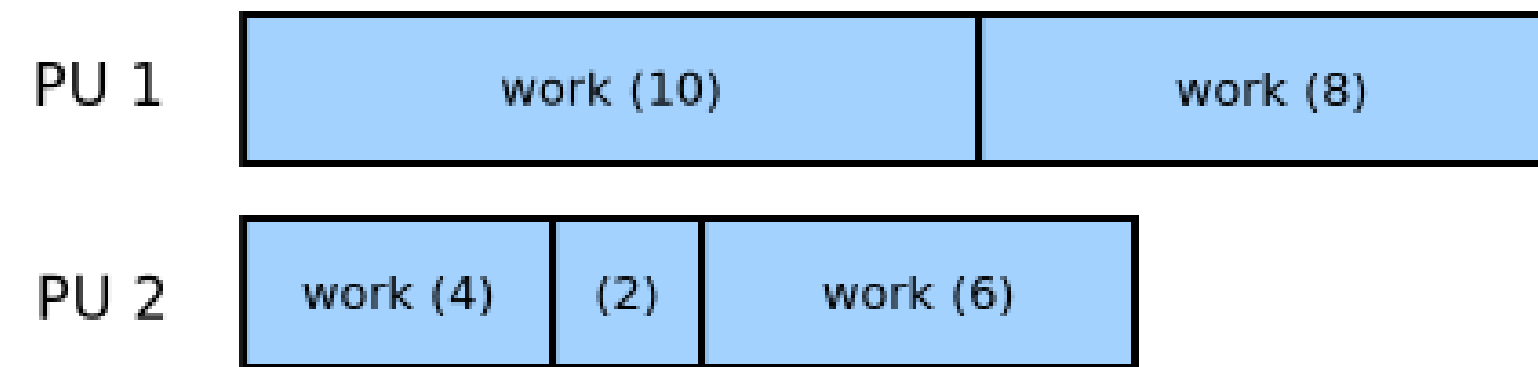


Exercises on load imbalance



- Compute the parallel efficiency in this scenario
- Can we assign the tasks differently to the processing units to achieve a better efficiency?

Exercises on load imbalance

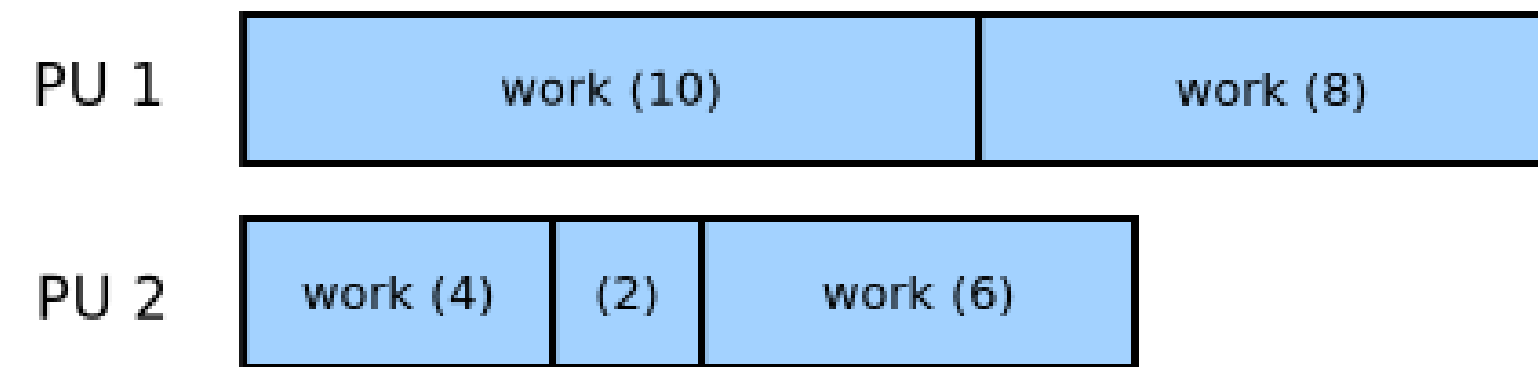


- Compute the parallel efficiency in this scenario

$$Efficiency = 1 - \frac{6}{18} = 1 - 0.166 = 0.83$$

- Can we assign the tasks differently to the processing units to achieve a better efficiency?

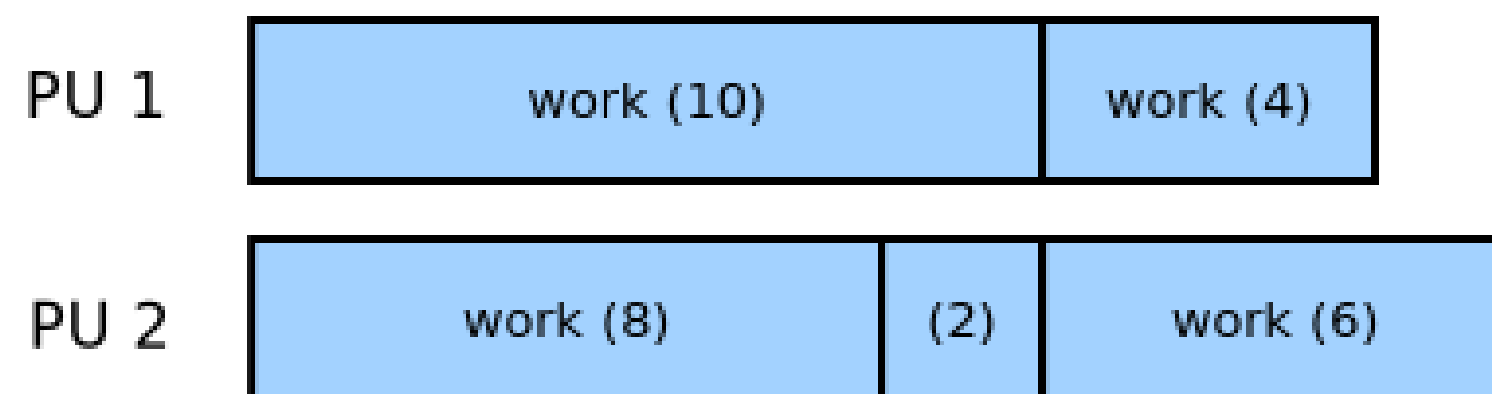
Exercises on load imbalance



- Compute the parallel efficiency in this scenario

$$Efficiency = 1 - \frac{6}{18} = 1 - 0.166 = 0.83$$

- Can we assign the tasks differently to the processing units to achieve a better efficiency?



$$Efficiency = 1 - \frac{2}{16} = 0.9375$$

More on load imbalance

- When the number of tasks is small, it is possible to compute the optimal solution
 - Assuming that we are able to accurately evaluate the time to execute each task
- When the number of big, it becomes too costly to try computing the optimal solutions
- Alternative solutions to static scheduling can be implemented:
 - Dynamic scheduling
 - The PUs get new tasks when they are idle
 - Work stealing
 - The PUs *steal* tasks from busy PUs when they are idle

Dealing with I/Os

- Tasks might need to perform I/O operations to the storage system
 - Read input data
 - Write results
- Operating systems (together with the hardware) implement a set of mechanisms to limit the impact of I/O operations on performance
 - The idea is to overlap I/O operations and computation
 - Interrupts
 - DMA engine for data transfers
- Assuming that I/O time is less than compute time, it can make I/O almost *invisible* with a sequential program

Dealing with I/Os

Example

- A 4-task parallel program
 - Each task read a 10-MB file before starting computing
 - Takes 1 second on the target platform
 - Each task performs 400 GFlop of computation
 - A core can perform 100 GFlops
 - 4 seconds per task

Execution with a single core

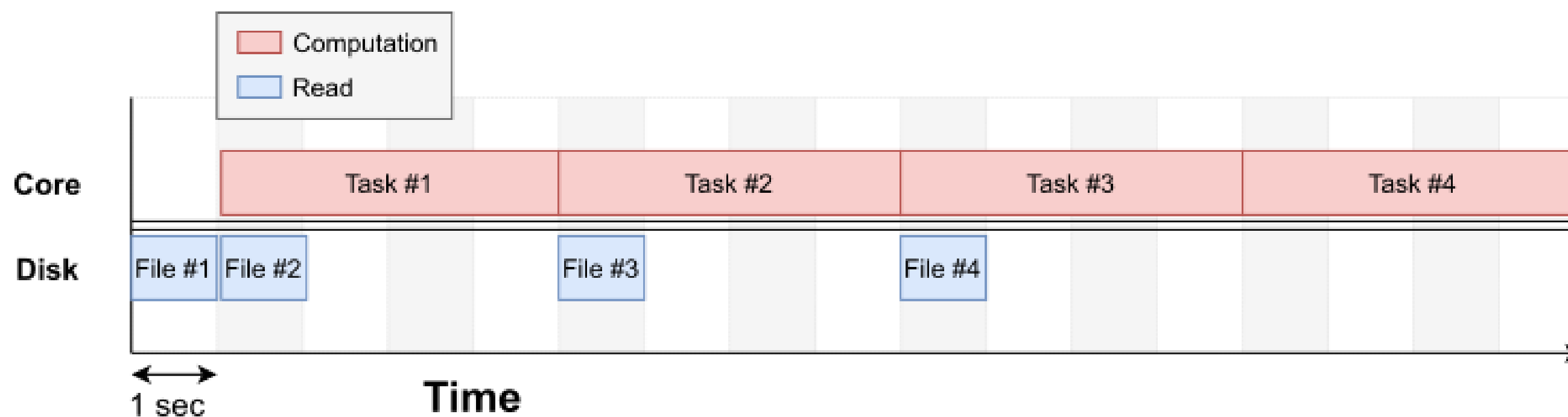
Credits: figure from eduWRENCH

Dealing with I/Os

Example

- A 4-task parallel program
 - Each task read a 10-MB file before starting computing
 - Takes 1 second on the target platform
 - Each task performs 400 GFlop of computation
 - A core can perform 100 GFlops
 - 4 seconds per task

Execution with a single core

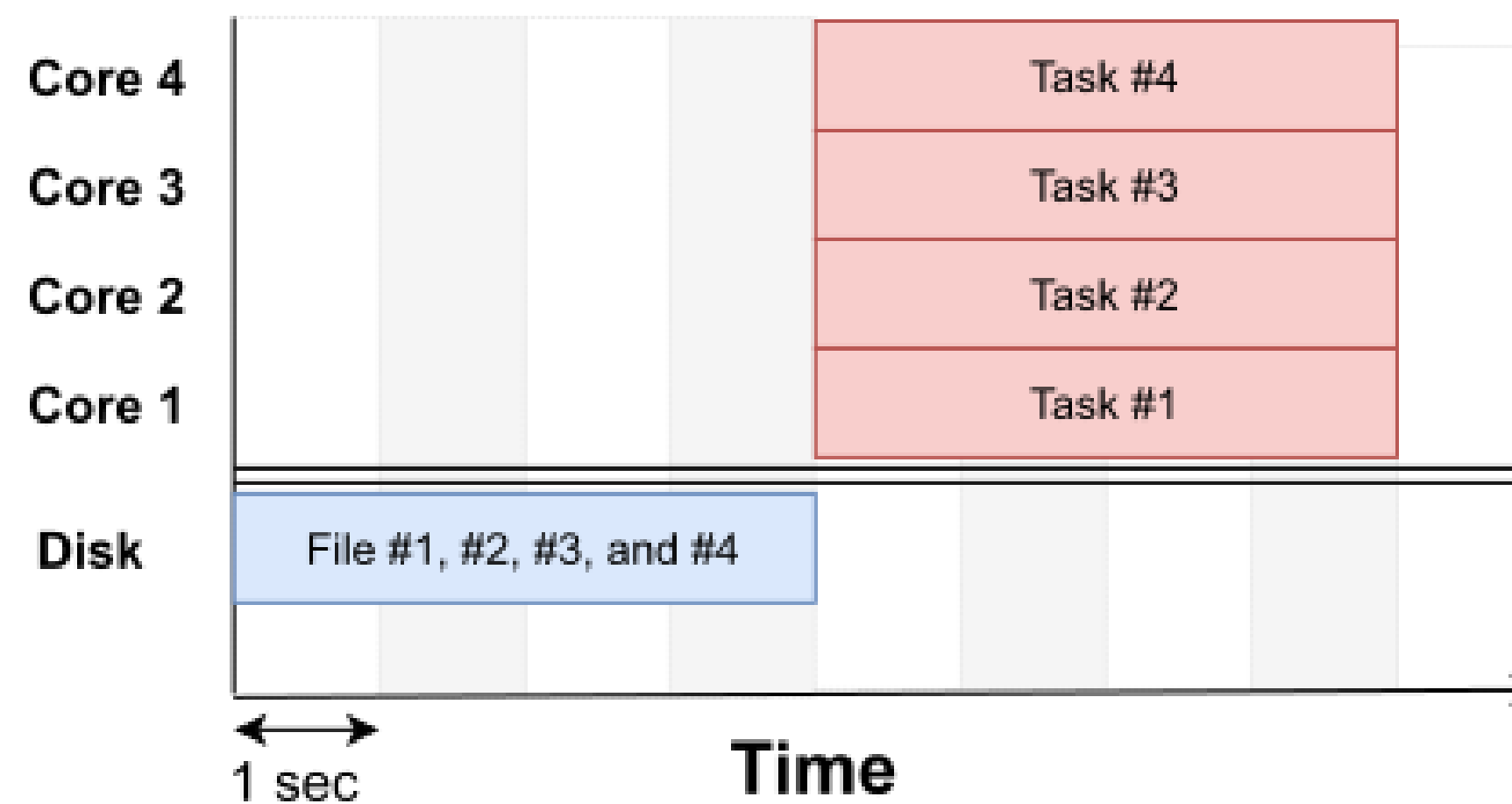


Credits: figure from eduWRENCH

Dealing with I/Os

What happens if we use multiple cores ?

- Execution on a 4-core processor
- Worst-case scenario: We do all reads before starting processing

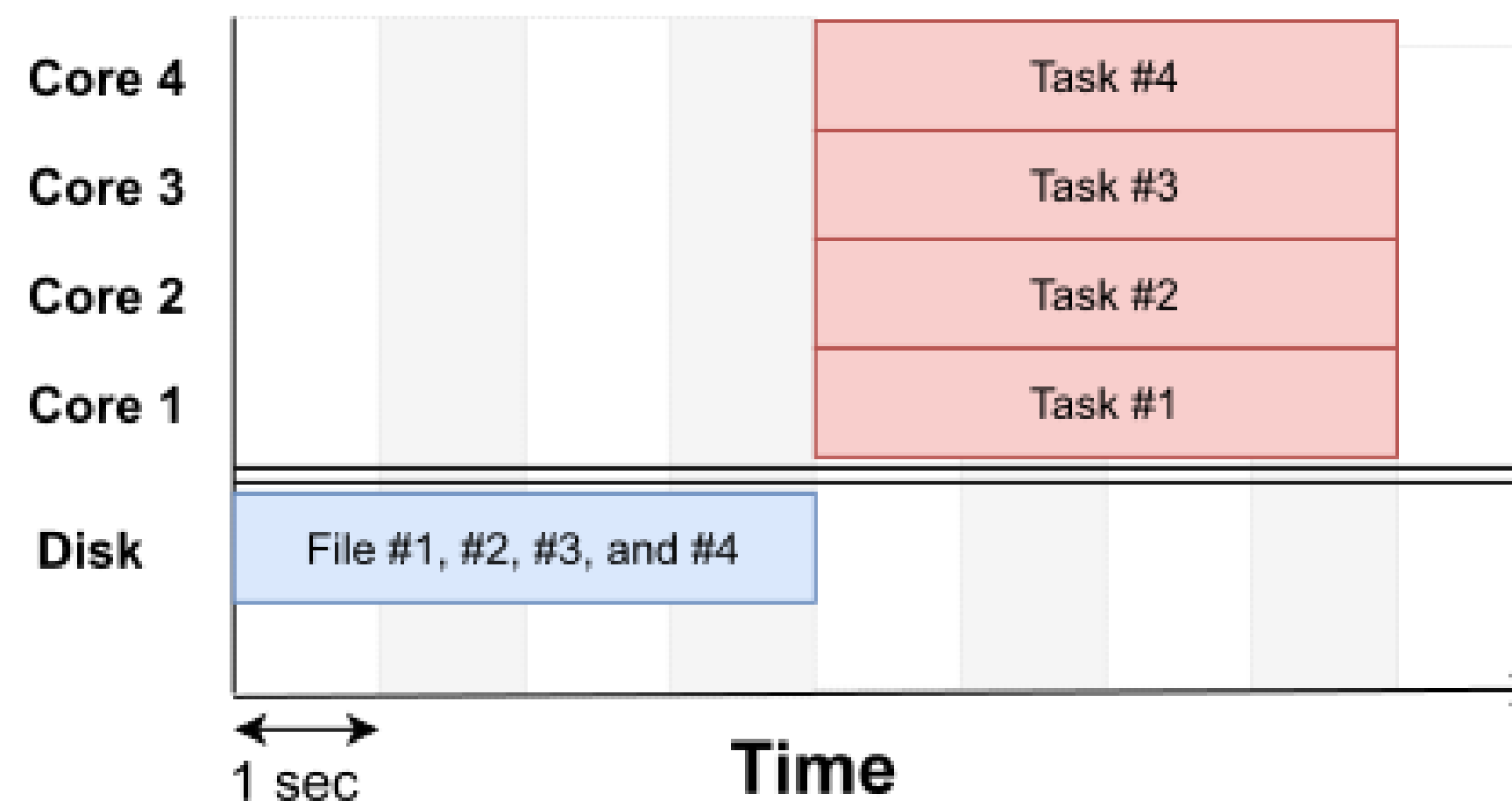


Speedup and efficiency

Dealing with I/Os

What happens if we use multiple cores ?

- Execution on a 4-core processor
- Worst-case scenario: We do all reads before starting processing



Speedup and efficiency

$$Speedup = 17/8 = 2.125$$

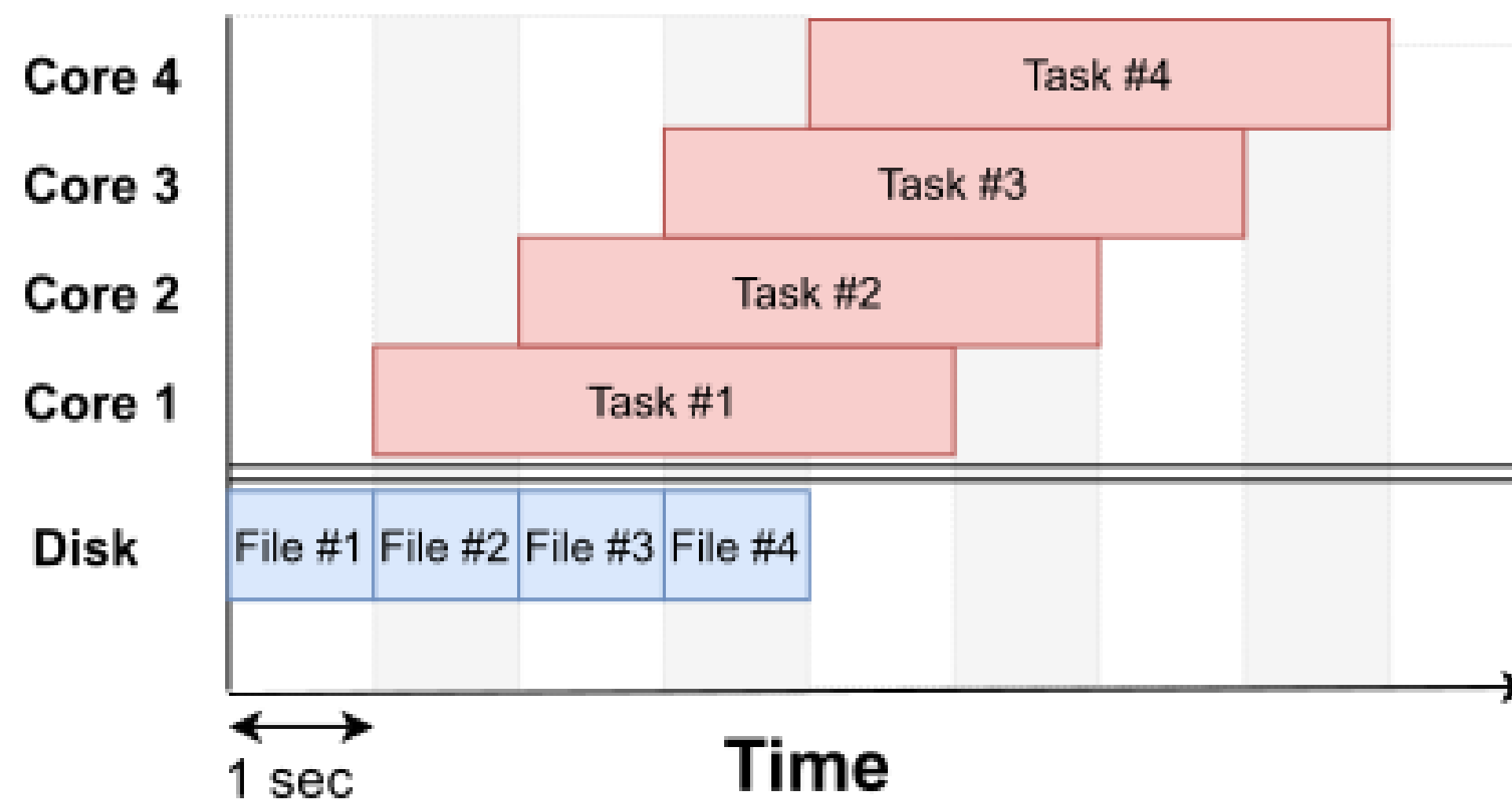
$$Efficiency = \frac{2.125}{4} = 0.53$$

Note that here computing the efficiency based on idle time would give a different result. This is because the sequential execution was already including idle time.

Dealing with I/Os

What happens if we use multiple cores ?

- To improve performance we can still try to overlap I/O operations and computations

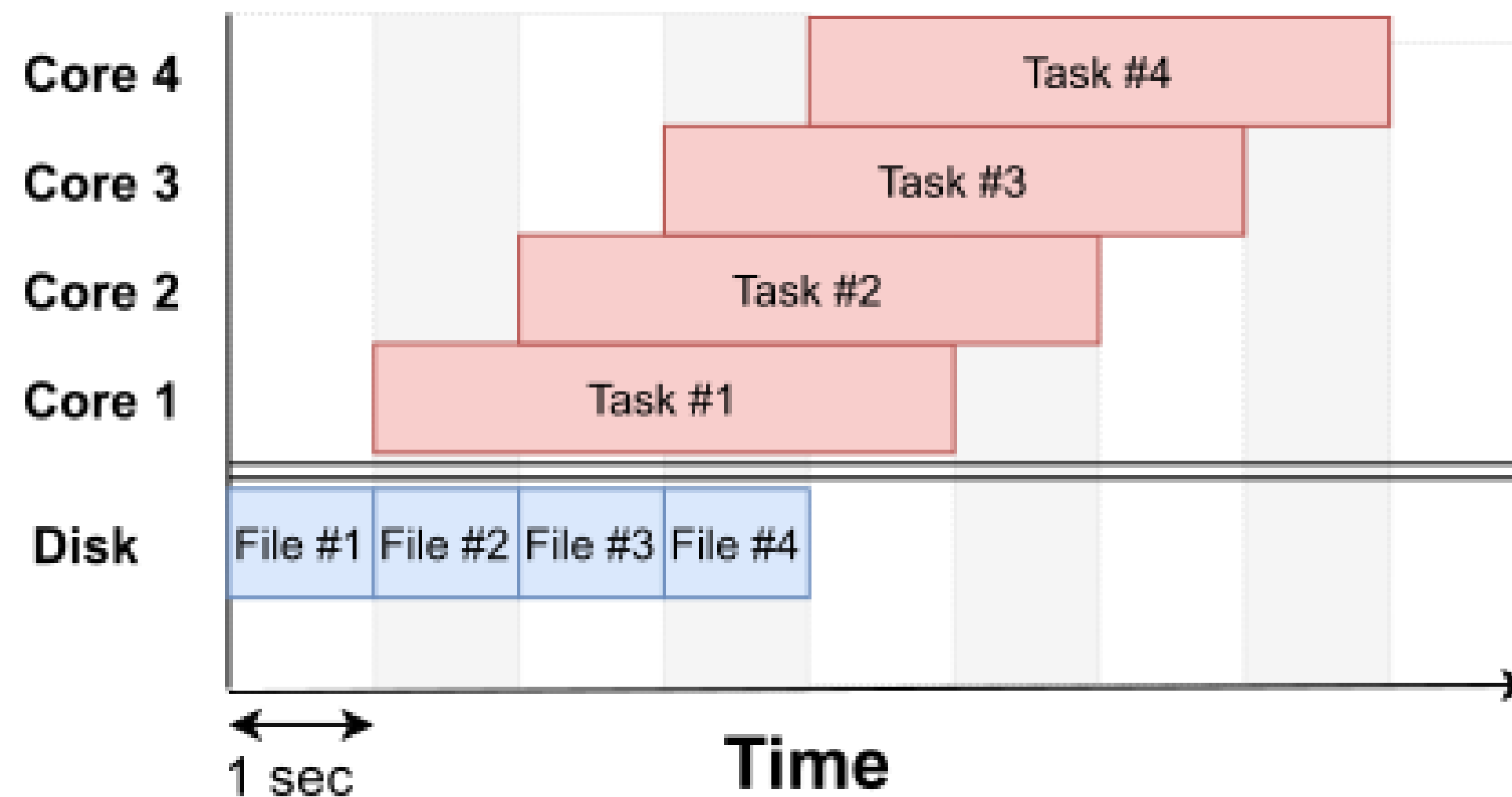


Credits: figure from eduWRENCH

Dealing with I/Os

What happens if we use multiple cores ?

- To improve performance we can still try to overlap I/O operations and computations



There is no performance improvement compared to the previous solution

Credits: figure from eduWRENCH

Exercise about I/Os

- A parallel program consists of 2 tasks:
 - Task 1 reads 20 MB of input, computes 500 Gflop, writes back 100 MB of output
 - Task 2 reads 100 MB of input, computes for 500 Gflop, writes back 100 MB of output
- We execute this program on a computer with two cores that compute at 100 Gflop/sec and with a disk with 100 MB/sec read and write bandwidth.

Is it better to run Task 1 or Task 2 first?

Exercise about I/Os

- A parallel program consists of 2 tasks:
 - Task 1 reads 20 MB of input, computes 500 Gflop, writes back 100 MB of output
 - Task 2 reads 100 MB of input, computes for 500 Gflop, writes back 100 MB of output
- We execute this program on a computer with two cores that compute at 100 Gflop/sec and with a disk with 100 MB/sec read and write bandwidth.

Is it better to run Task 1 or Task 2 first?

- Running Task 1 first allows starting computing earlier

Exercise about I/Os

- A parallel program consists of 2 tasks:
 - Task 1 reads 20 MB of input, computes 500 Gflop, writes back 100 MB of output
 - Task 2 reads 100 MB of input, computes for 500 Gflop, writes back 100 MB of output
- We execute this program on a computer with two cores that compute at 100 Gflop/sec and with a disk with 100 MB/sec read and write bandwidth.

Is it better to run Task 1 or Task 2 first?

- Running Task 1 first allows starting computing earlier
- T1 first -- Exec time = 7.2 s
- T2 first -- Exec time = 8 s

Task dependencies

- Until now, we have assumed that tasks can be executed in any order
 - It is not always the case

Definitions

```
There is a dependency between task A and task B, if B cannot start executing until A is done
```

- The typical reason for having task dependencies is that Task B needs the output of task A

DAG of tasks

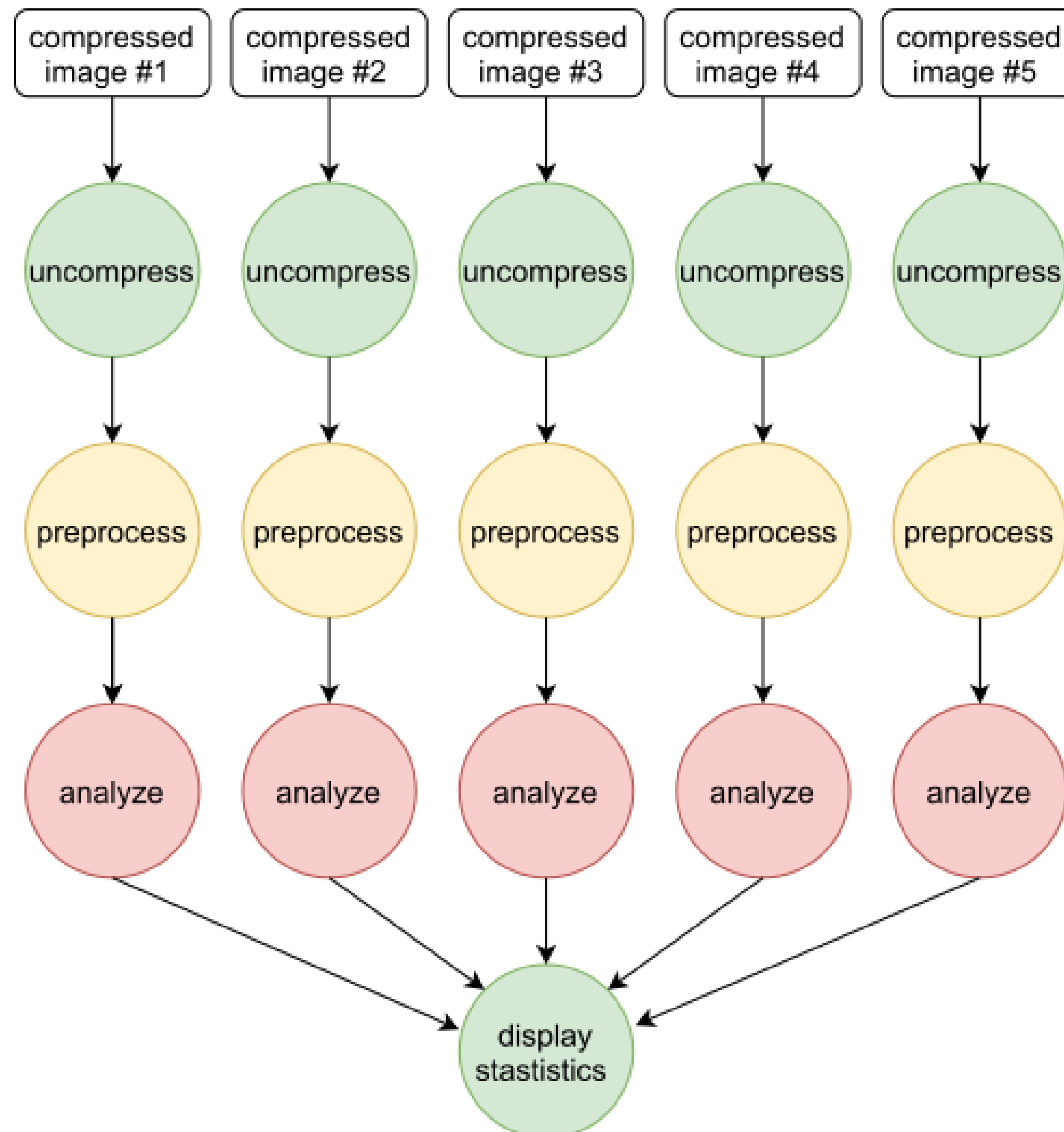
- It can be convenient to represent dependencies between tasks using a **Directed Acyclic Graph**
 - Vertices are tasks
 - Edges are dependencies

Example of DAG

- Program that counts the number of car objects in a set of compressed street images.
- It includes the following steps:
 - Each image needs to be uncompressed
 - Each image is pre-processed to remove noise
 - Each image is analyzed to find cars
 - Car count statistics are displayed

Example of DAG

DAG assuming 5 images



Some concepts related to DAGs

DAG level

A task is on *level n* of the DAG if the *longest path from the entry task(s) to this task is of length n*

- The entry tasks are the tasks that do not depend on any other tasks
- The path length is measured in number of traversed vertices

Maximum level width

The maximum level width of a DAG the *maximum number of tasks in one DAG level*

- Helps determining the maximum number of PUs to use
- If the maximum level width is 4
 - Using 4 PUs should provide a speedup compared to using 3 PUs
 - It does not necessarily implies that 5 PUs would not improve performance
 - We do not have to wait for all tasks from one level to terminate before starting the tasks from the next level

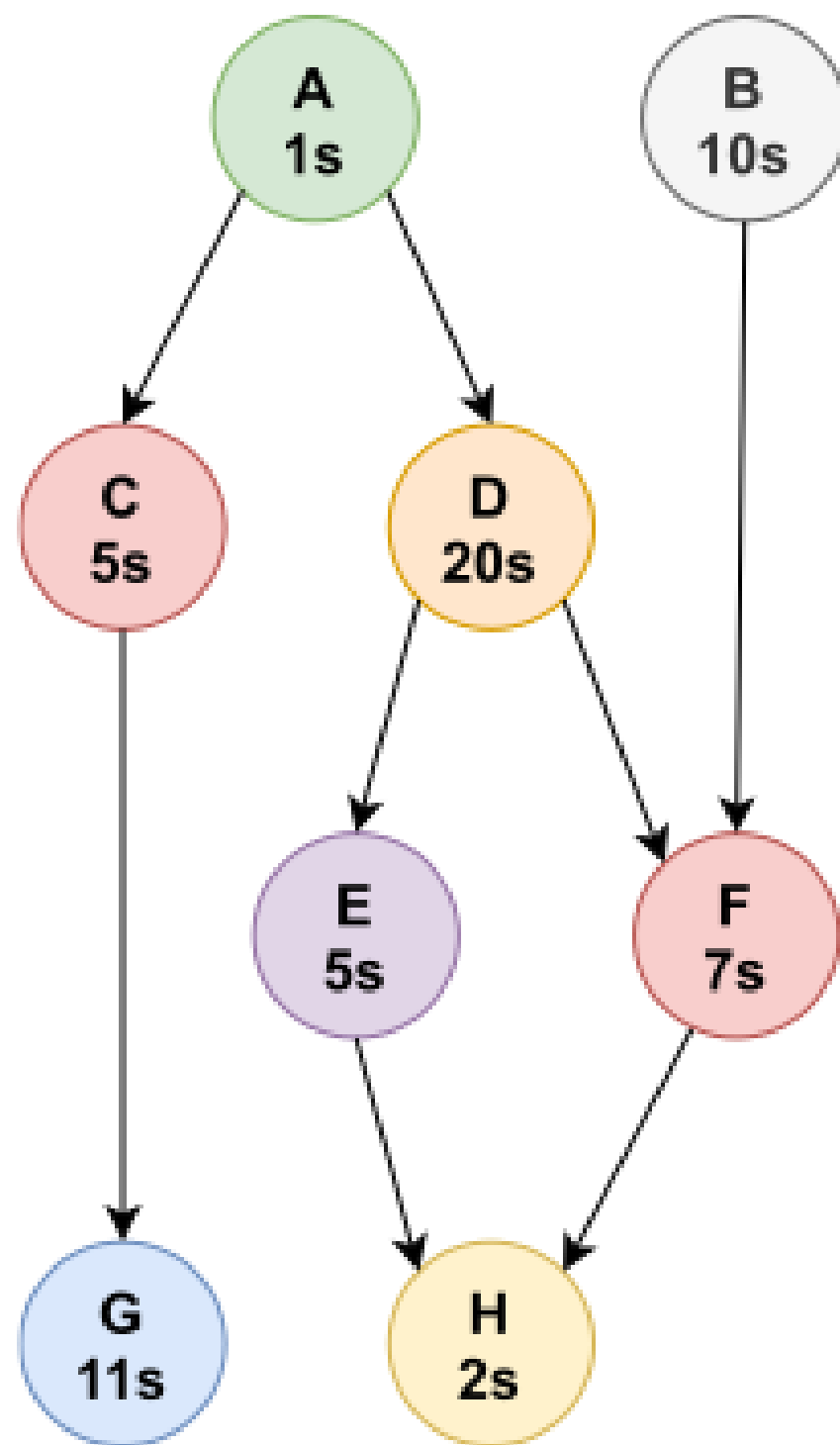
Some concepts related to DAGs

Critical path

The critical path is **the longest path in the DAG from the entry task(s) to the exit task(s)**

- The path length is measured in task duration, including the entry and the exit task(s)
- Allows evaluating the maximum performance that can be obtained
 - No matter the number of PUs, the program cannot execute faster than the length of the critical path

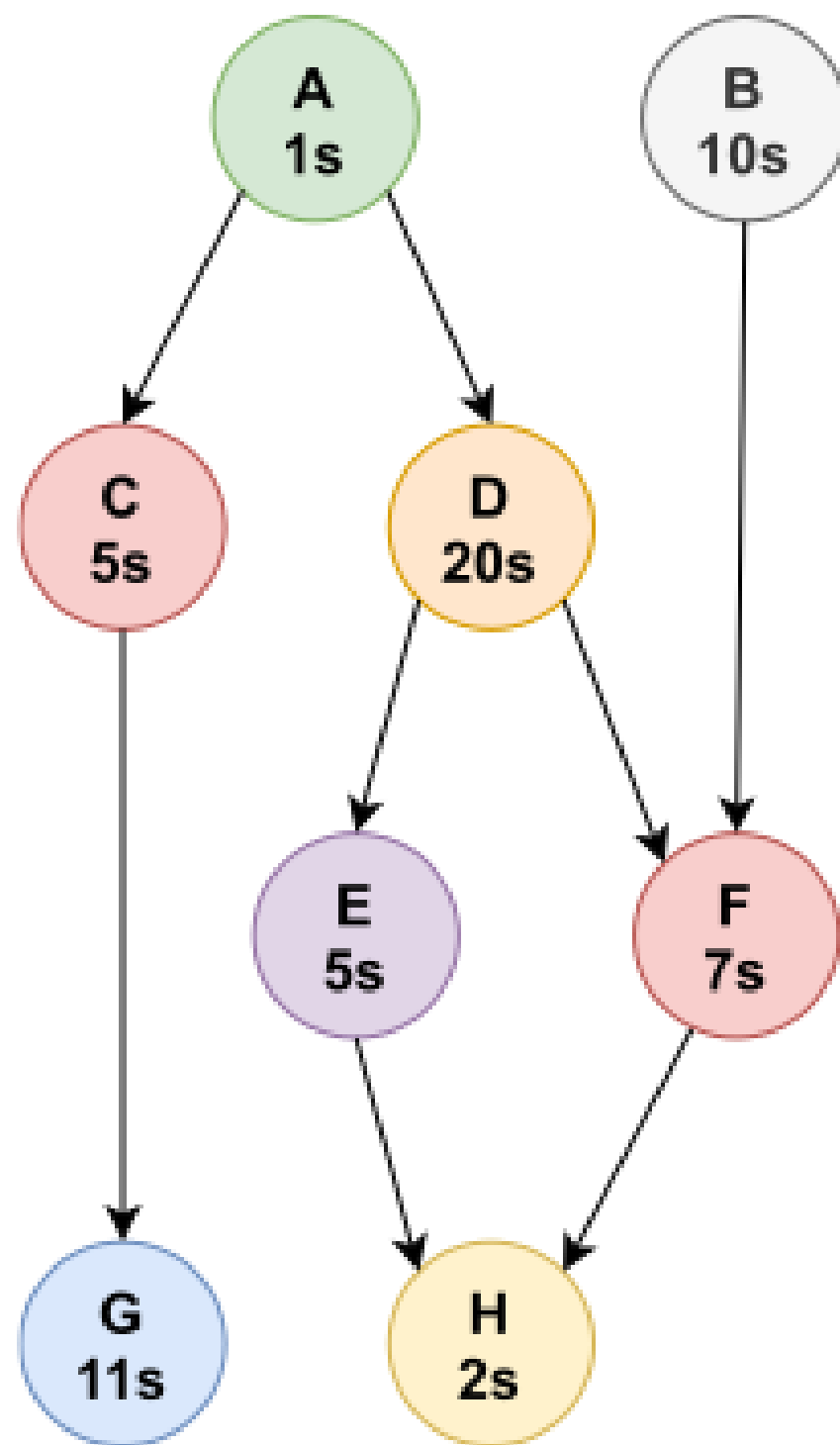
Exercise about DAGs



- Task level of each task:
- Maximum width:
- Critical path:

Credits: figure from eduWRENCH

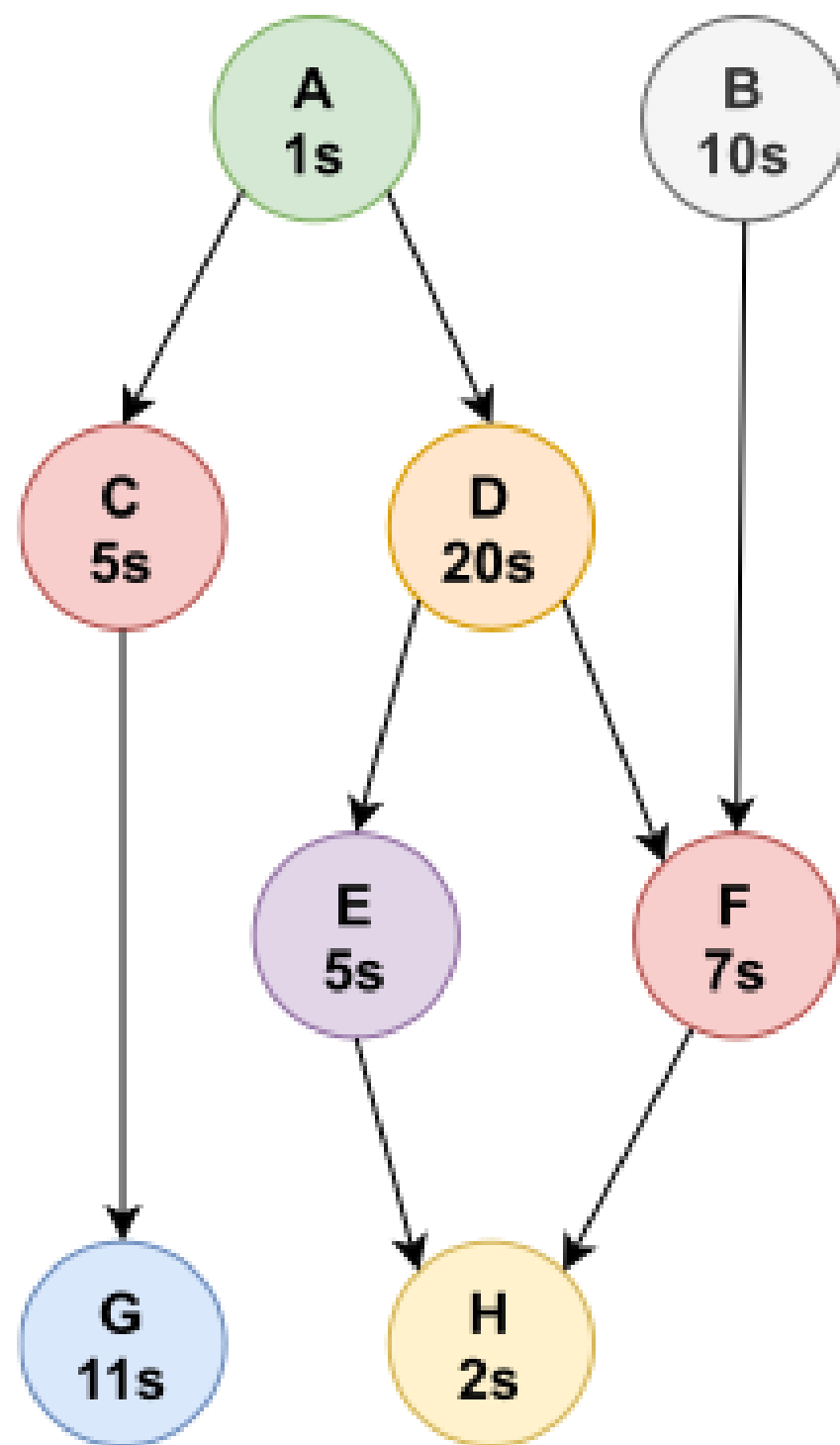
Exercise about DAGs



- Task level of each task:
 - Level 0: A, B
 - Level 1: C, D
 - Level 2: E, F, G
 - Level 3: H
- Maximum width:
- Critical path:

Credits: figure from eduWRENCH

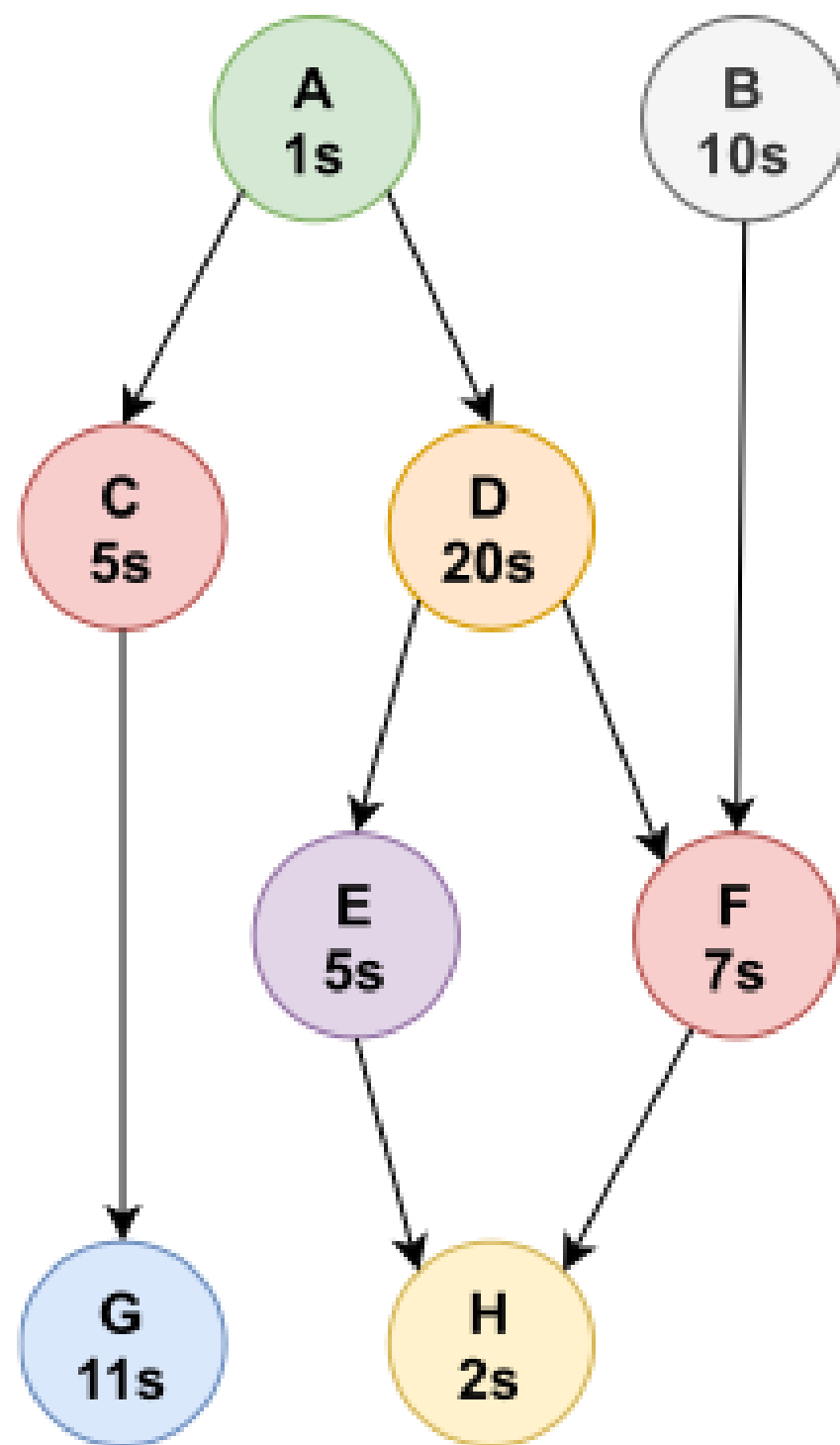
Exercise about DAGs



- Task level of each task:
 - Level 0: A, B
 - Level 1: C, D
 - Level 2: E, F, G
 - Level 3: H
- Maximum width:
 - Level 2 has 3 tasks
- Critical path:

Credits: figure from eduWRENCH

Exercise about DAGs



- Task level of each task:
 - Level 0: A, B
 - Level 1: C, D
 - Level 2: E, F, G
 - Level 3: H
- Maximum width:
 - Level 2 has 3 tasks
- Critical path:
 - A - D - F - H
 - $1 + 20 + 7 + 2 = 30s$

Credits: figure from eduWRENCH

Chosing which task to run

- Choose a task that is ready to run
 - All parents tasks are finished
 - **What if multiple tasks are ready to run?**

Multiple strategies are possible

Chosing which task to run

- Choose a task that is ready to run
 - All parents tasks are finished
 - **What if multiple tasks are ready to run?**

Multiple strategies are possible

- Task on the critical path first (in general a good strategy)
- Task with the largest work first
- task with the smallest work first

No strategy is always best

Other constraint on the execution of tasks: Memory

- Each task consumes memory space
 - Load input data
 - Write output data
- Consuming more memory than the total physical memory space is not recommended
 - Induces swapping memory pages from/to disk
 - **Very slow**, to be avoided

Memory constraints should be taken into account when scheduling tasks

Conclusion

Take-away points

Several metrics to measure the performance of parallel programs

- Execution time
- Speedup
- Efficiency
- Scalability

Amdahl's law implies that infinite scalability is impossible

Problems that impair performance/scalability

- Load imbalance
- I/O operations
- Task dependencies