# 30. Parallel Programming I

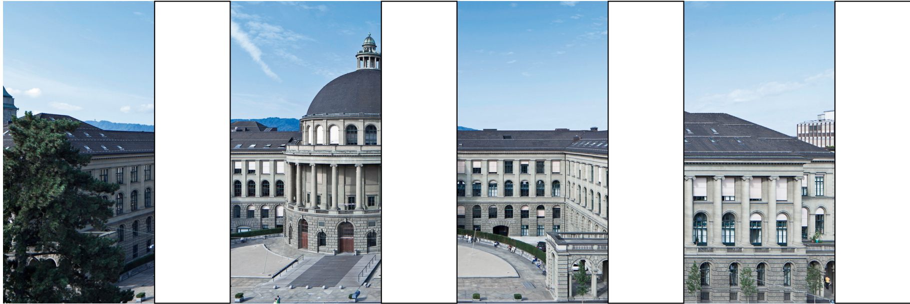
Moore's Law, Hardware Architectures, Parallel Execution , Multi-Threading, Parallelism and Concurrency, C++ Threads, Scalability: Amdahl and Gustafson , Scheduling

[Task-Scheduling: Cormen et al, Kap. 27] [Concurrency, Scheduling: Williams, Kap. 1.1 – 1.2]

# Motivation: Paint a Picture (1 Artist)

Model: sequential execution

# Motivation: Paint a Picture (4 Artists)

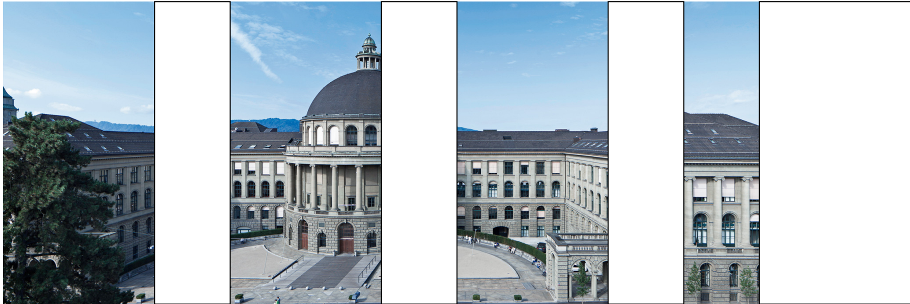


Model: parallel execution

# Motivation: Paint a Picture (4 Artists, 3 Brushes)

Model: concurrent execution

# Motivation: Paint a Picture (4 Artists, 1 Brush)




Model: concurrent execution

# Models

- **Sequential:** the program is executed step-by-step in the prescribed order
- **Parallel:** tasks are executed in parallel. No synchronisation necessary, enough resources available.
- **Concurrent:** Tasks are executed in parallel. But there is a need for synchronisation: tasks have to be interrupted.

# Why Parallelism and Concurrency?

- **Reactive / Interactive / Multi-User- Systems** particularly graphical user interfaces $\Rightarrow$ Concurrency
- Computation-heavy tasks, like data processing and analysis, where **performance** is important $\Rightarrow$ Parallelism
- **Natural parallelism / concurrency**: distributed systems, device drivers.

# A Bit of Technical Background: CPUs

Today's typical computers (desktops, phones, ...) offer (at least)

- CPU: central processing unit, general-purpose computation device
- GPU: graphics processing unit, incredibly efficient for linear- algebra computations (games, graphics; machine learning).

Today's CPUs are typically multi-core:

- Each core is essentially a dedicated CPU that can execute code
- Examples:
  - Intel i7-8700K has 6 cores (from 2017)
  - Intel i9-12900KF has 16 cores (from 2021)
  - AMD Ryzen 9 5950X has 16 cores (from 2020)
  - Apple M1 Max has 10 cores (from 2021)

Intel i7 (2017)

# Moore’s Law

Observation by Gordon E. Moore:

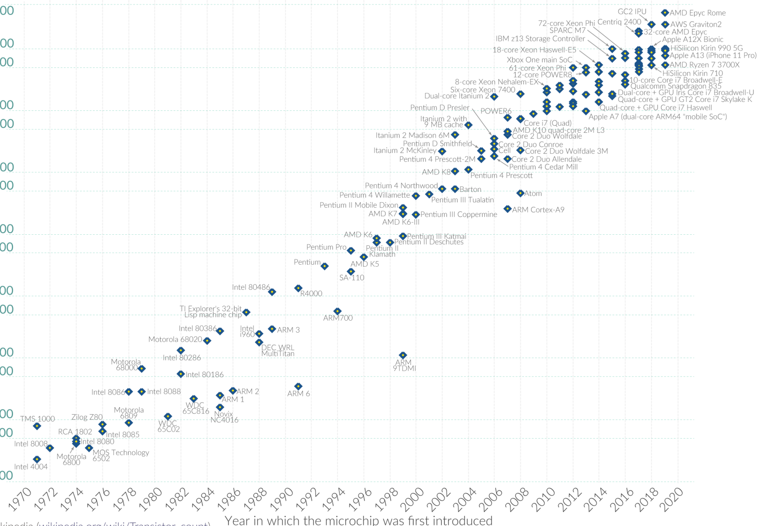
The number of transistors on integrated circuits doubles approximately every two years.

Gordon E. Moore (1929)

# Moore’s Law: The number of transistors on microchips doubles every two years

Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years.
This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.




Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)
OurWorldinData.org – Research and data to make progress against the world’s largest problems.

Licensed under CC-BY by the authors Hannah Ritchie and Max Roser.

ourworldindata.org, https://en.wikipedia.org/wiki/Transistor_count

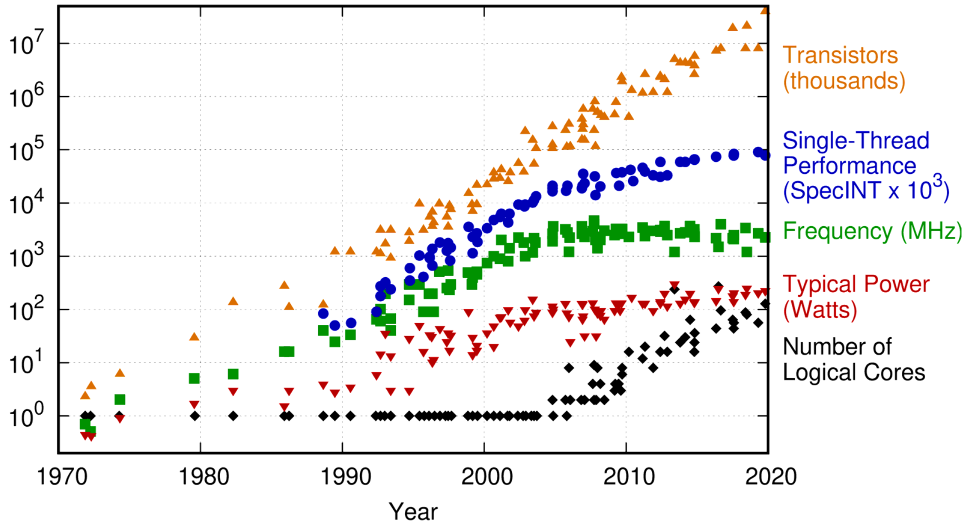
# For a long time...

- the sequential execution became faster ("Instruction Level Parallelism", "Pipelining", Higher Frequencies)
- more and smaller transistors = more performance
- programmers simply waited for the next processor generation to improve the performance of their programs.

# Today

- the frequency of processors does not increase significantly and more (heat dissipation problems)
- the instruction level parallelism does not increase significantly any more
- the execution speed is dominated by memory access times (but caches still become larger and faster)

48 Years of Microprocessor Trend Data




Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2019 by K. Rupp

https://github.com/karlrupp/microprocessor-trend-data

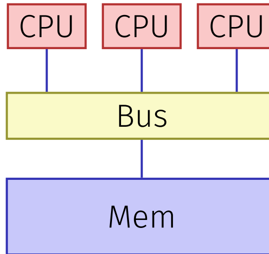
# Multicore

- Use transistors for more compute cores
- Parallelism in the software

$\Rightarrow$ programmers have to write **parallel programs** to benefit from new hardware

# (Simplified) Assumption: Computing Model

Independent Computing Cores




Shared Memory

## 30.1 Multi-Threading, Parallelism and Concurrency

# Processes and Threads

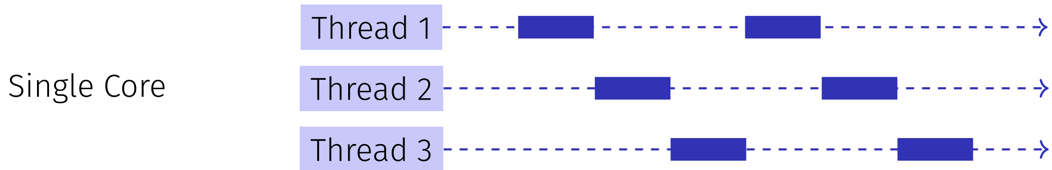
- **Process**: instance of a program
  - each process has a separate context, even a separate address space (”can only see its own memory“)
  - OS manages processes (resource control, scheduling, synchronisation)
- **Thread**: thread of execution of a program
  - Threads share the address space
  - fast context switch between threads

# Why Multithreading?

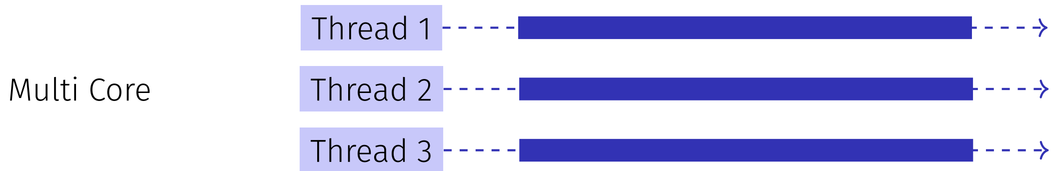
- Avoid “polling” resources (files, network, keyboard)
- Interactivity (e.g. responsivity of GUI programs)
- Several applications / clients in parallel
- Parallelism (performance!)

# Multithreading conceptually

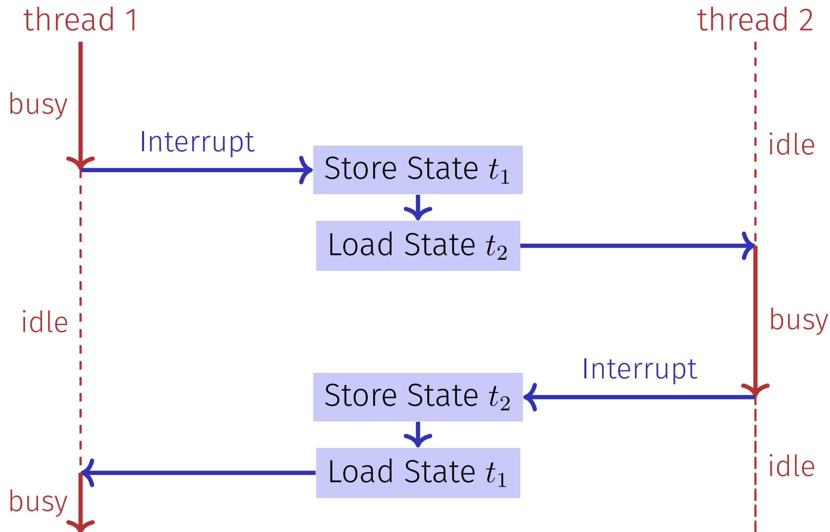
Single Core




Multi Core

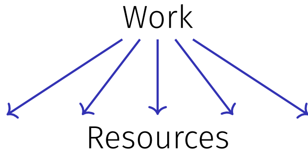


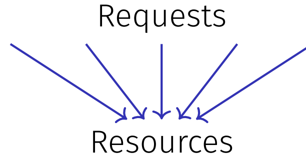
# Thread switch on one core (Preemption)



# Parallelism vs. Concurrency

- **Parallelism:** Use extra resources to solve a problem faster
- **Concurrency:** Correctly and efficiently manage access to shared resources
- The notions overlap. With parallel computations there is nearly always a need to synchronise.



# Thread Safety

**Thread Safety** means that in a concurrent application of a program this always yields the desired results.

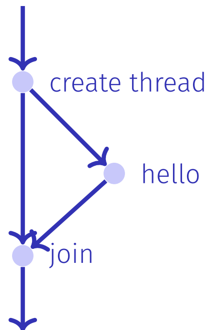
Many optimisations (Hardware, Compiler) target towards the correct execution of a *sequential* program.

Concurrent programs need an annotation that switches off certain optimisations selectively.

## 30.2 C++ Threads

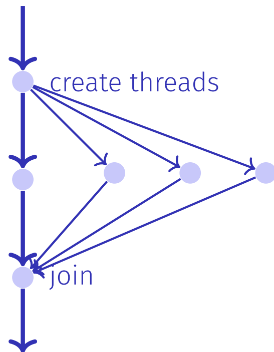
# C++11 Threads

```cpp
#include <iostream>
#include <thread>

void hello(){
  std::cout << "hello\n";
}

int main(){
  // create and launch thread t
  std::thread t(hello);
  // wait for termination of t
  t.join();
  return 0;
}
```



# C++11 Threads

```
void hello(int id){
  std::cout << "hello from " << id << "\n";
}

int main(){
  std::vector<std::thread> tv(3);
  int id = 0;
  for (auto & t:tv)
    t = std::thread(hello, ++id);
  std::cout << "hello from main \n";
  for (auto & t:tv)
    t.join();
  return 0;
}
```



# Nondeterministic Execution!

One execution:

```
hello from main
hello from 2
hello from 1
hello from 0
```

Other execution:

```
hello from 1
hello from main
hello from 0
hello from 2
```

Other execution:

```
hello from main
hello from 0
hello from hello from 1
2
```

# Technical Detail

To let a thread continue as background thread:

```
void background();

void someFunction(){
  ...
  std::thread t(background);
  t.detach();
  ...
} // no problem here, thread is detached
```

# More Technical Details

- With allocating a thread, reference parameters are copied, except explicitly std::ref is provided at the construction.
- Can also run Functor or Lambda-Expression on a thread
- In exceptional circumstances, joining threads should be executed in a catch block

More background and details in chapter 2 of the book *C++ Concurrency in Action*, Anthony Williams, Manning 2012. also available online at the ETH library.

# What can (not) be Parallelized

```
int pow8(int b){
  int b2 = b * b;
  int b4 = b2 * b2;
  return b4 * b4;
}

int main(){
  int x;
  std::cin >> x;
  x = pow8(x);
  std::cout << x;
}
```

- Program computes $x^8$
- All parts of the program must be executed in fixed order
  - Input → Computation → Output
  - b2 → b4 → b8
- No two computations are independent
- and must all be executed sequentially

# What can (not) be Parallelized

```
int pow8(int b){
  int b2 = b * b;
  int b4 = b2 * b2;
  return b4 * b4;
}

int main(){
  std::vector<int> v = ....;
  for (int& x : v)
    x = pow8(x);
  ...
}
```

- Program computes $x^8$ for each $x \in v$
- The computation of $x_i^8$ does not depend on the computation of $x_j^8$.
- We can parallelise the computation of all $x^8$
- Parallelisation *can* reduce runtime if
  - the computation to be parallelised runs long enough)
  - sufficient CPUs are available

# What can (not) be Parallelized

```
int main(){
  std::vector<int> v = ....;
  for (int& x : v)
    x = pow8(x);
  ...
}
```

- Example is obviously parallelisable
- Solche Probleme nennt man **emberassingly parallel**

- Many problems and algorithms are different:
  - can be parallelized but that requires a deeper analyis
  - need preprocessing or postprocessing in order to decompose the problem into parallelisable subproblems or to combine the partial results, respectively.
- Example: Matrix-Multiplication, Mergesort

## 30.3 Scalability: Amdahl and Gustafson

# Scalability

In parallel Programming:

- Speedup when increasing number $p$ of processors
- What happens if $p \to \infty$?
- Program scales linearly: Linear speedup.

# Parallel Performance

Given a fixed amount of computing work $W$ (number computing steps)

$T_1$: Sequential execution time

$T_p$: Parallel execution time on $p$ CPUs

- Perfection: $T_p = T_1/p$
- Performance loss: $T_p > T_1/p$ (usual case)
- Sorcery: $T_p < T_1/p$

# Parallel Speedup

Parallel speedup $S_p$ on $p$ CPUs:

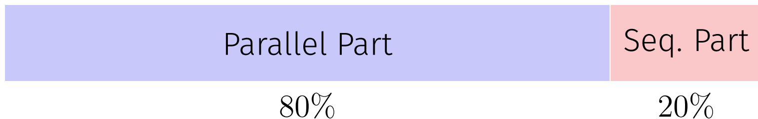
$$S_p = \frac{W/T_p}{W/T_1} = \frac{T_1}{T_p}.$$

- Perfection: linear speedup $S_p = p$
- Performance loss: sublinear speedup $S_p < p$ (the usual case)
- Sorcery: superlinear speedup $S_p > p$

Efficiency:$E_p = S_p/p$

# Reachable Speedup?

Parallel Program




$$T_1 = 10$$
$$T_8 = \frac{10 \cdot 0.8}{8} + 10 \cdot 0.2 = 1 + 2 = 3$$
$$S_8 = \frac{T_1}{T_8} = \frac{10}{3} \approx 3.3 < 8 \quad (!)$$

# Amdahl's Law: Ingredients

Computational work $W$ falls into two categories

- Paralellisable part $W_p$
- Not parallelisable, sequential part $W_s$

Assumption: $W$ can be processed sequentially by **one** processor in $W$ time units ($T_1 = W$):

$$T_1 = W_s + W_p$$
$$T_p \geq W_s + W_p/p$$

# Amdahl's Law

$$S_p = \frac{T_1}{T_p} \leq \frac{W_s + W_p}{W_s + \frac{W_p}{p}}$$

# Amdahl's Law

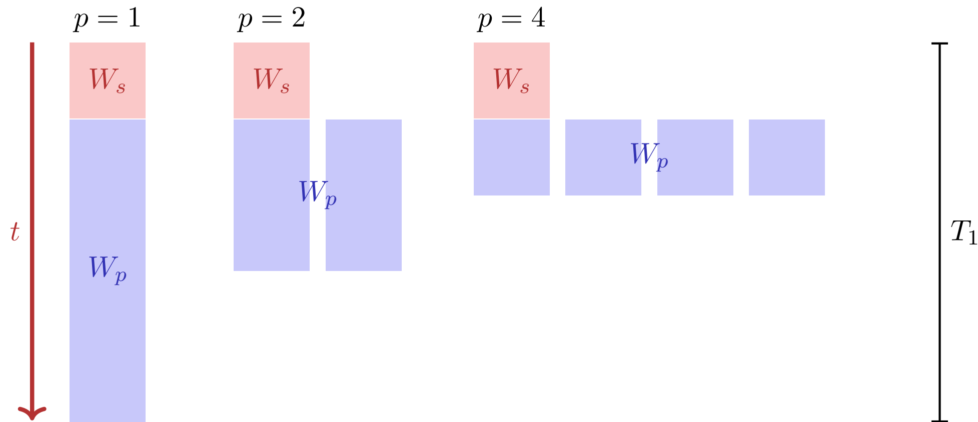
With sequential, not parallelizable fraction $\lambda$: $W_s = \lambda W$, $W_p = (1-\lambda)W$:

$$S_p \leq \frac{1}{\lambda + \frac{1-\lambda}{p}}$$

Thus

$$S_\infty \leq \frac{1}{\lambda}$$

# Illustration Amdahl's Law



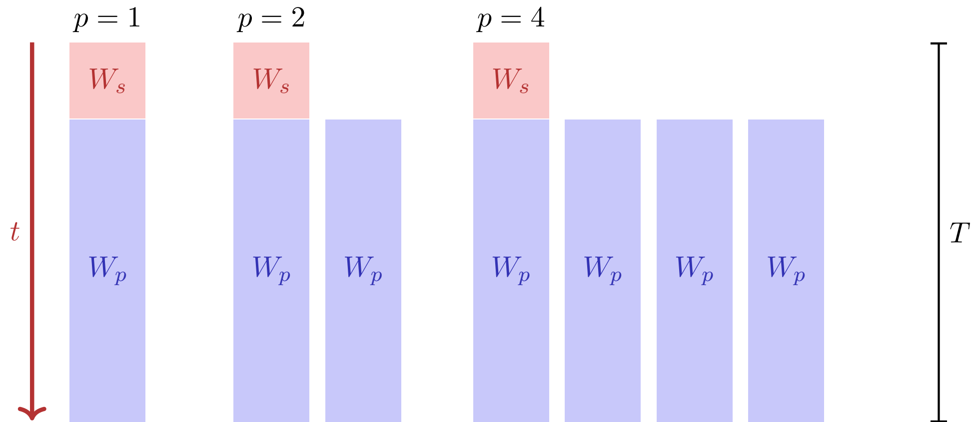
# Amdahl's Law is bad news

All non-parallel parts of a program can cause problems

# Gustafson's Law

- Fix the time of execution
- Vary the problem size.
- Assumption: the sequential part stays constant, the parallel part becomes larger

# Illustration Gustafson's Law



# Gustafson's Law

Work that can be executed by one processor in time $T$:

$$W_s + W_p = T$$

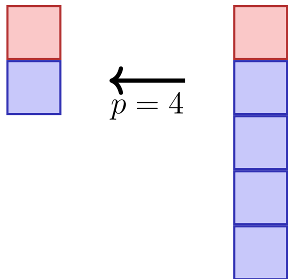
Work that can be executed by $p$ processors in time $T$:

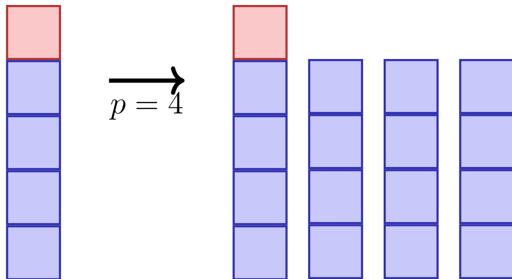
$$W_s + p \cdot W_p = \lambda \cdot T + p \cdot (1 - \lambda) \cdot T$$

Speedup:

$$\begin{aligned} S_p = \frac{W_s + p \cdot W_p}{W_s + W_p} &= p \cdot (1 - \lambda) + \lambda \\ &= p - \lambda(p - 1) \end{aligned}$$

# Amdahl vs. Gustafson



# Amdahl vs. Gustafson

The laws of Amdahl and Gustafson are models of speedup for parallelization.

Amdahl assumes a fixed **relative** sequential portion, Gustafson assumes a fixed **absolute** sequential part (that is expressed as portion of the work $W_1$ and that does not increase with increasing work).

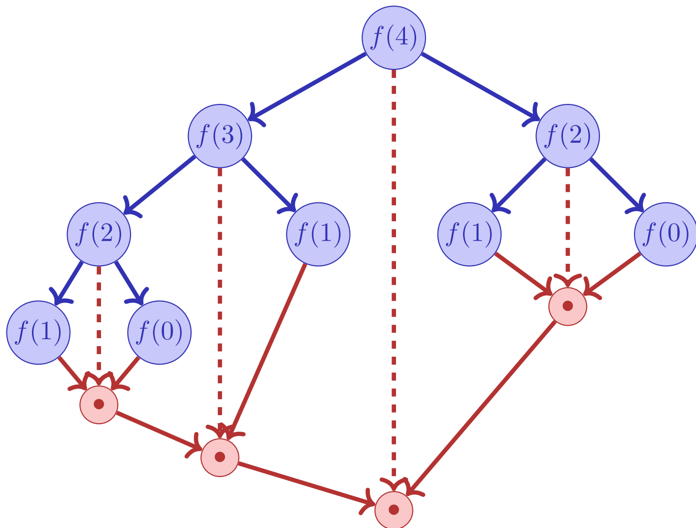
The two models do not contradict each other but describe the runtime speedup of different problems and algorithms.

## 30.4 Scheduling

# Example: Fibonacci

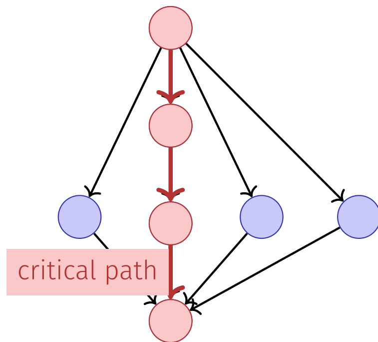
```
int fib_task(int x){
    if (x < 2) {
      return x;
  } else {
      auto f1 = std::async(fib_task, x-1);
      auto f2 = std::async(fib_task, x-2);
      return f1.get() + f2.get();
    }
}
```

# Task-Graph



# Question

- Each Node (task) takes 1 time unit.
- Arrows depict dependencies.
- Minimal execution time when number of processors = $\infty$?

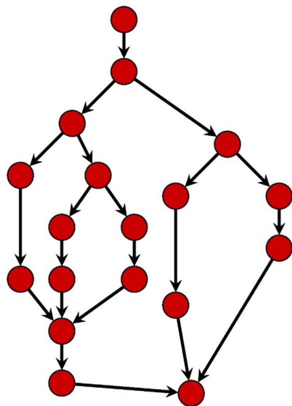


# Performance Model

- $p$ processors
- Dynamic scheduling
- $T_p$: Execution time on $p$ processors

# Performance Model

- $T_p$: Execution time on $p$ processors
- $T_1$: **Work**: time for executing total work on one processor
- $T_1/T_p$: Speedup



# Performance Model

- $T_\infty$: **Span**: critical path, execution time on $\infty$ processors. Longest path from root to sink.
- $T_1/T_\infty$: **Parallelism:** wider is better
- Lower bounds:

$$T_p \geq T_1/p \quad \text{Work law}$$
$$T_p \geq T_\infty \quad \text{Span law}$$

# Greedy Scheduler

Greedy scheduler: at each time it schedules as many as availbale tasks.

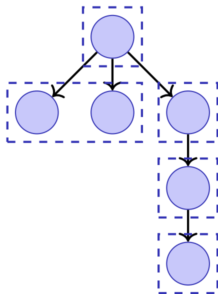
*Theorem 38*

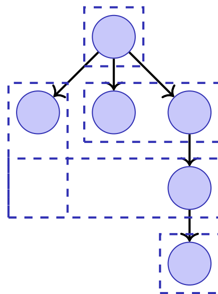
*On an ideal parallel computer with $p$ processors, a greedy scheduler executes a multi-threaded computation with work $T_1$ and span $T_\infty$ in time*

$$T_p \leq T_1/p + T_\infty$$

# Example

Assume $p = 2$.





$T_p = 5$

$T_p = 4$

# Proof of the Theorem

Assume that all tasks provide the same amount of work.

- Complete step: $p$ tasks are available.
- incomplete step: less than $p$ steps available.

Assume that number of complete steps larger than $\lfloor T_1/p \rfloor$. Executed work $\geq \lfloor T_1/p \rfloor \cdot p + p = T_1 - T_1 \mod p + p > T_1$. Contradiction. Therefore maximally $\lfloor T_1/p \rfloor$ complete steps.

We now consider the graph of tasks to be done. Any maximal (critical) path starts with a node $t$ with $\deg^-(t) = 0$. An incomplete step executes all available tasks $t$ with $\deg^-(t) = 0$ and thus decreases the length of the span. Number incomplete steps thus limited by $T_\infty$.

# Consequence

if $p \ll T_1/T_\infty$, i.e. $T_\infty \ll T_1/p$, then

$$T_p \leq T_1/p + T_\infty \quad \Rightarrow \quad T_p \lesssim T_1/p$$

**Fibonacci**

$T_1(n)/T_\infty(n) = \Theta(\phi^n/n)$. For moderate sizes of $n$ we can use a lot of processors yielding linear speedup.

# Example: Parallelism of Mergesort

- Work (sequential runtime) of Mergesort $T_1(n) = \Theta(n \log n)$.
- Span $T_\infty(n) = \Theta(n)$
- Parallelism $T_1(n)/T_\infty(n) = \Theta(\log n)$ (Maximally achievable speedup with $p = \infty$ processors)