

# 30. Parallel Programming I

---

Moore's Law and the Free Lunch, Hardware Architectures, Parallel Execution, Flynn's Taxonomy, Multi-Threading, Parallelism and Concurrency, C++ Threads, Scalability: Amdahl and Gustafson, Data-parallelism, Task-parallelism, Scheduling

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

# The Free Lunch

The free lunch is over <sup>50</sup>

---

<sup>50</sup>"The Free Lunch is Over", a fundamental turn toward concurrency in software, Herb Sutter, Dr. Dobb's Journal, 2005

# Moore's Law

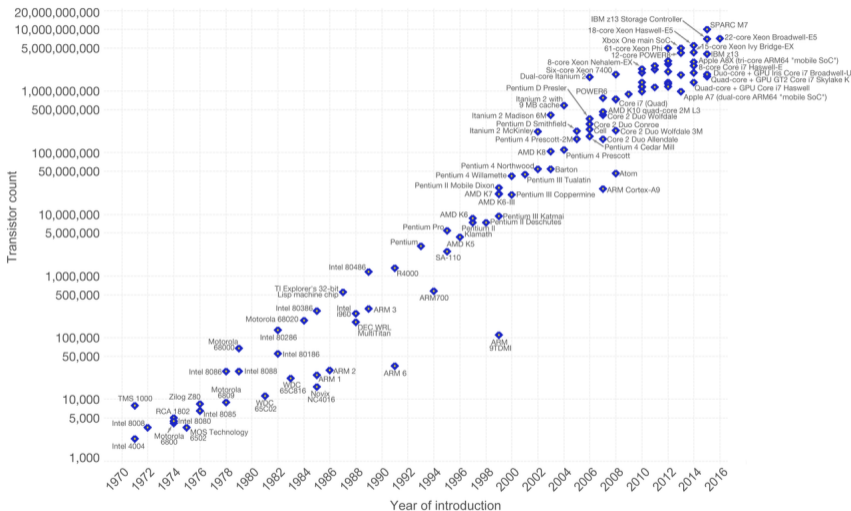
Observation by Gordon E. Moore:  
The number of transistors on integrated circuits  
doubles approximately every two years.



Gordon E. Moore (1929)

# Moore's Law – The number of transistors on integrated circuit chips (1971–2016)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.



Data source: Wikipedia ([https://en.wikipedia.org/wiki/Transistor\\_count](https://en.wikipedia.org/wiki/Transistor_count))  
The data visualization is available at [OurWorldinData.org](https://ourworldindata.org). There you find more visualizations and research on this topic.

Licensed under [CC-BY-SA](https://creativecommons.org/licenses/by-sa/4.0/) by the author Max Roser.

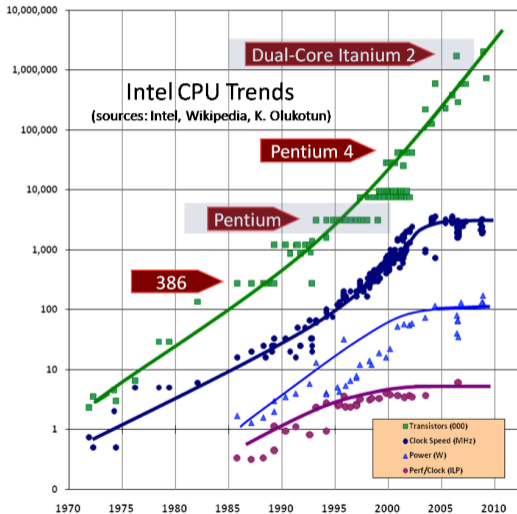
# For a long time...

- 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

# 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)

# Trends



# Multicore

- Use transistors for more compute cores
- Parallelism in the software
- Programmers have to write parallel programs to benefit from new hardware

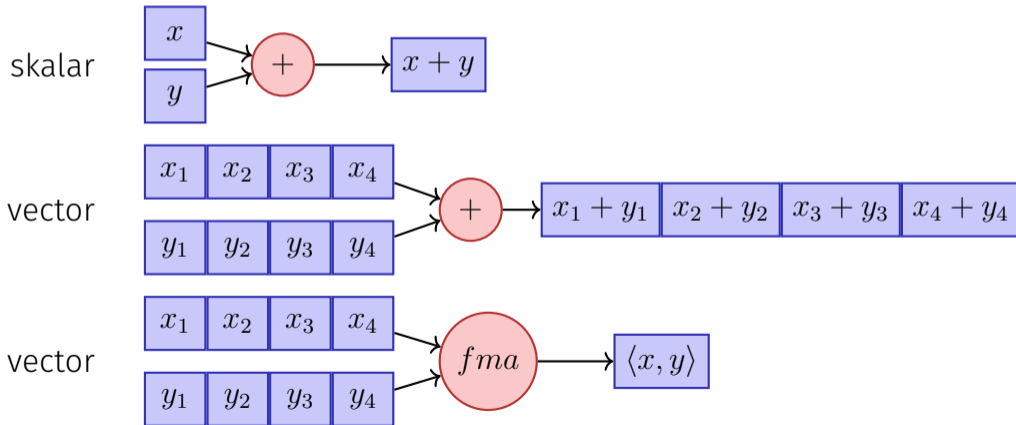


# Forms of Parallel Execution

- Vectorization
- Pipelining
- Instruction Level Parallelism
- Multicore / Multiprocessing
- Distributed Computing

# Vectorization

Parallel Execution of the same operations on elements of a vector (register)



# Pipelining in CPUs



## Multiple Stages

- Every instruction takes 5 time units (cycles)
- In the best case: 1 instruction per cycle, not always possible (“stalls”)

**Paralellism** (several functional units) leads to **faster execution**.

# ILP – Instruction Level Parallelism

Modern CPUs provide several hardware units and execute independent instructions in parallel.

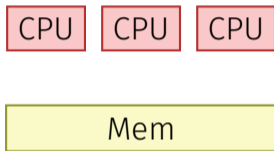
- Pipelining
- Superscalar CPUs (multiple instructions per cycle)
- Out-Of-Order Execution (Programmer observes the sequential execution)
- Speculative Execution ()

## 30.2 Hardware Architectures

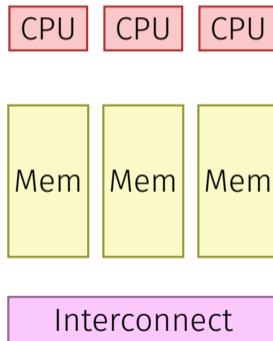
---

# Shared vs. Distributed Memory

Shared Memory



Distributed Memory



# Shared vs. Distributed Memory Programming

- Categories of programming interfaces
  - Communication via message passing
  - Communication via memory sharing
- It is possible:
  - to program shared memory systems as distributed systems (e.g. with message passing MPI)
  - program systems with distributed memory as shared memory systems (e.g. partitioned global address space PGAS)

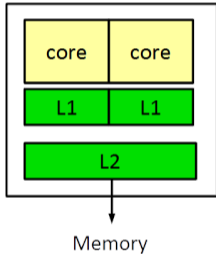
# Shared Memory Architectures

- Multicore (Chip Multiprocessor - CMP)
- Symmetric Multiprocessor Systems (SMP)
- Simultaneous Multithreading (SMT = Hyperthreading)
  - one physical core, Several Instruction Streams/Threads: several virtual cores
  - Between ILP (several units for a stream) and multicore (several units for several streams). Limited parallel performance.
- Non-Uniform Memory Access (NUMA)

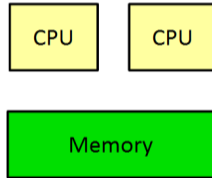
Same programming interface



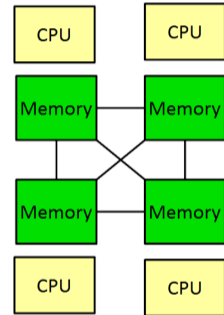
# Overview



CMP



SMP

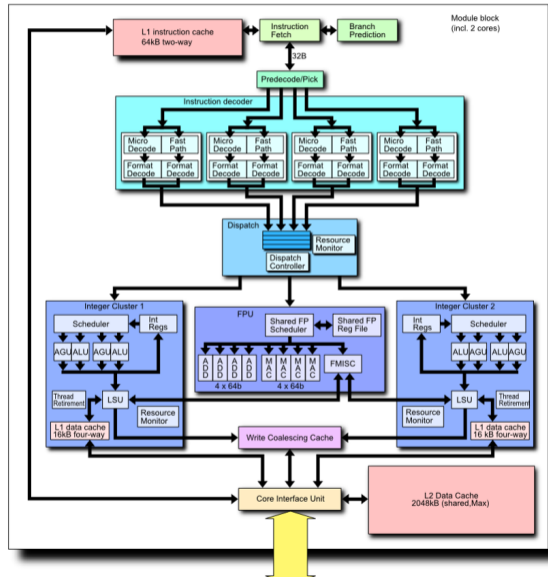


NUMA

# An Example

AMD Bulldozer: between CMP and SMT

- 2x integer core
- 1x floating point core

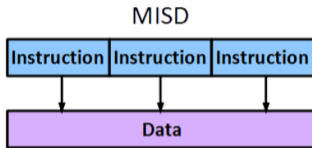
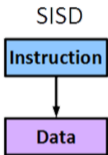


# Flynn's Taxonomy

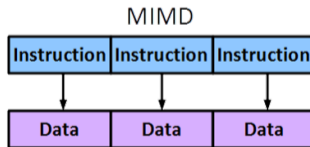
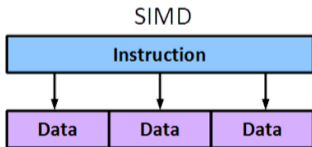
Single-Core

Fehlertoleranz

SI = Single Instruction  
MI = Multiple Instructions



SD = Single Data  
MD = Multiple Data



Vector Computing / GPU

Multi-Core

# Massively Parallel Hardware

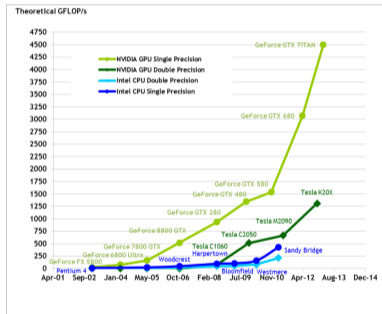
[General Purpose] Graphical Processing Units  
([GP]GPUs)

## ■ Revolution in High Performance Computing

- Calculation 4.5 TFlops vs. 500 GFlops
- Memory Bandwidth 170 GB/s vs. 40 GB/s

## ■ SIMD

- High data parallelism
- Requires own programming model. Z.B. CUDA / OpenCL



## 30.3 Multi-Threading, Parallelism and Concurrency

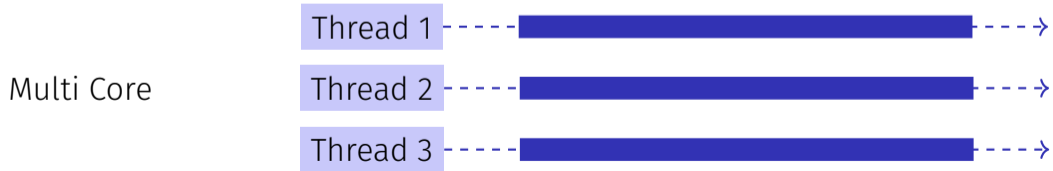
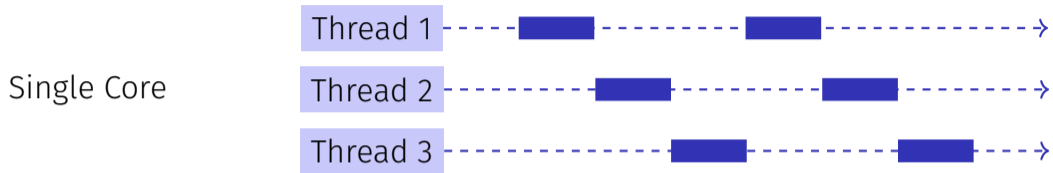
# Processes and Threads

- Process: instance of a program
  - each process has a separate context, even a separate address space
  - OS manages processes (resource control, scheduling, synchronisation)
- Threads: threads of execution of a program
  - Threads share the address space
  - fast context switch between threads

# Why Multithreading?

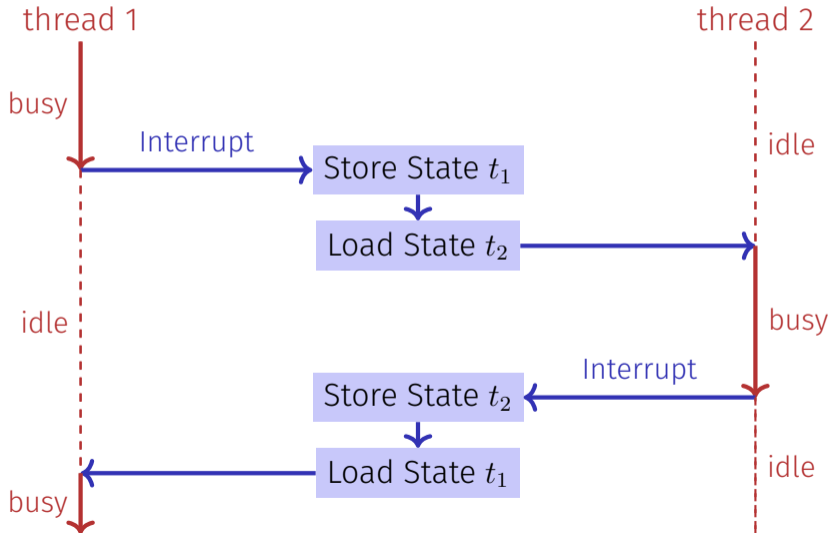
- Avoid “polling” resources (files, network, keyboard)
- Interactivity (e.g. responsivity of GUI programs)
- Several applications / clients in parallel
- Parallelism (performance!)

# Multithreading conceptually





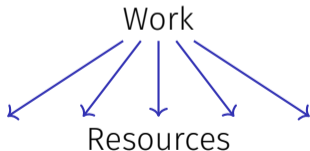
# Thread switch on one core (Preemption)



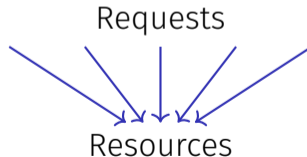
# Parallelität vs. Concurrency

- **Parallelism:** Use extra resources to solve a problem faster
- **Concurrency:** Correctly and efficiently manage access to shared resources
- Begriffe überlappen offensichtlich. Bei parallelen Berechnungen besteht fast immer Synchronisierungsbedarf.

## Parallelism



## Concurrency



# 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.

# Example: Caches

- Access to registers faster than to shared memory.
- Principle of locality.
- Use of Caches (transparent to the programmer)

If and how far a cache coherency is guaranteed depends on the used system.



## 30.4 C++ Threads

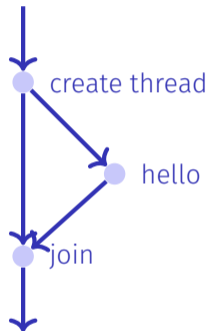
---

# C++11 Threads

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

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

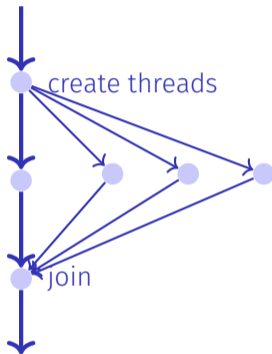
int main(){
    // create and launch thread t
    std::thread t(hello);
    // wait for termination of t
    t.join();
    return 0;
}
```



# 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.

## 30.5 Scalability: Amdahl and Gustafson

---

# Scalability

In parallel Programming:

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

# Parallel Performance

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

Sequential execution time  $T_1$

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:

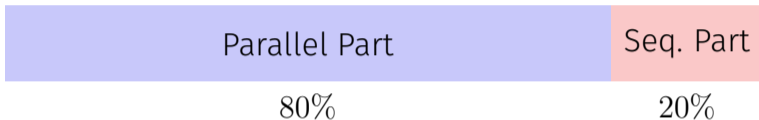
$$S_p = \frac{W/T_p}{W/T_1} = \frac{T_1}{T_p}.$$

- 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

- Parallellisable 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

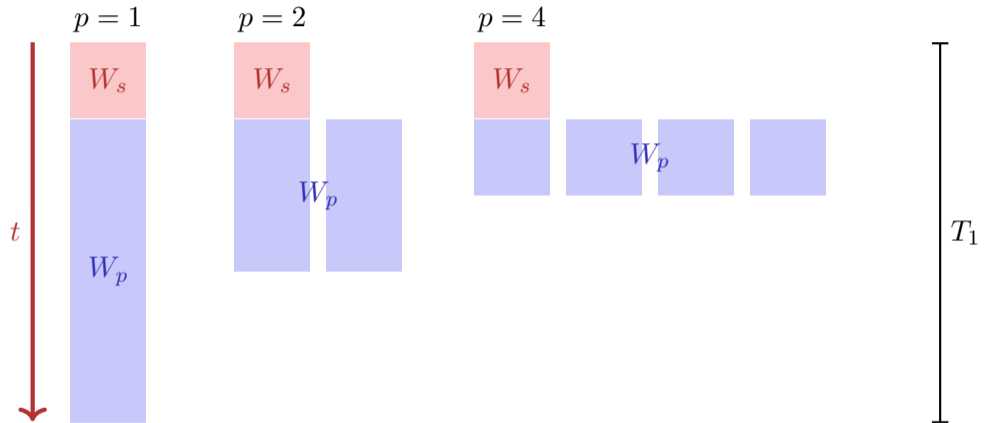
With sequential, not parallelizable fraction  $\lambda$ :  $W_s = \lambda W$ ,  $W_p = (1 - \lambda)W$ :

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

Thus

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

# Illustration Amdahl's Law



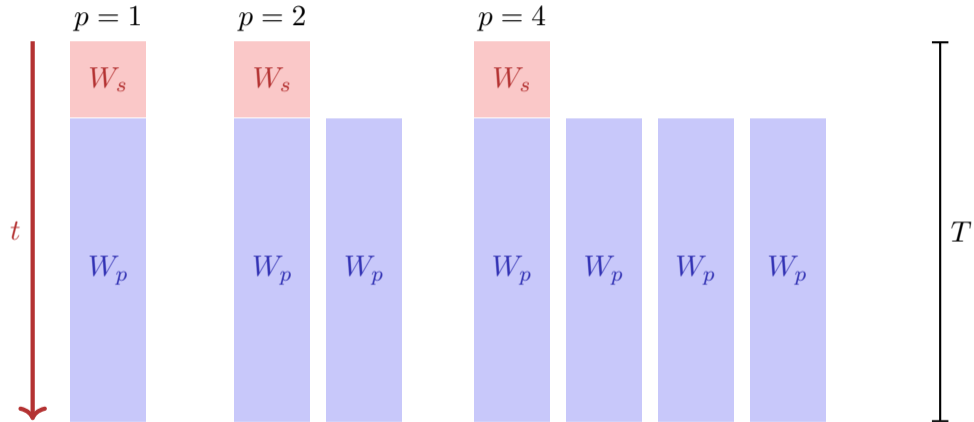
# Amdahl's Law is bad news

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  $T$ :

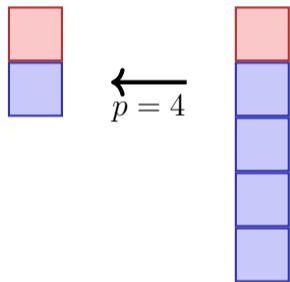
$$W_s + p \cdot W_p = \lambda \cdot T + p \cdot (1 - \lambda) \cdot T$$

Speedup:

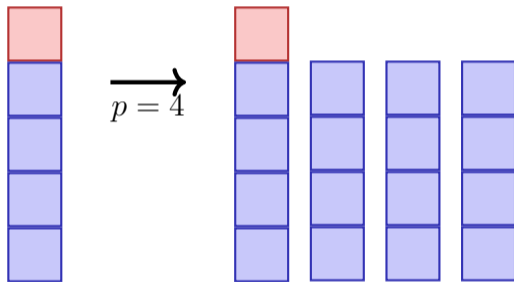
$$\begin{aligned} S_p &= \frac{W_s + p \cdot W_p}{W_s + W_p} = p \cdot (1 - \lambda) + \lambda \\ &= p - \lambda(p - 1) \end{aligned}$$

# Amdahl vs. Gustafson

Amdahl



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.

## 30.6 Task- and Data-Parallelism

---

# Parallel Programming Paradigms

- **Task Parallel:** Programmer explicitly defines parallel tasks.
- **Data Parallel:** Operations applied simultaneously to an aggregate of individual items.

## Example Data Parallel (OMP)

```
double sum = 0, A[MAX];  
#pragma omp parallel for reduction (+:ave)  
for (int i = 0; i < MAX; ++i)  
    sum += A[i];  
return sum;
```

## Example Task Parallel (C++11 Threads/Futures)

```
double sum(Iterator from, Iterator to)
{
    auto len = from - to;
    if (len > threshold){
        auto future = std::async(sum, from, from + len / 2);
        return sumS(from + len / 2, to) + future.get();
    }
    else
        return sumS(from, to);
}
```

# Work Partitioning and Scheduling

- Partitioning of the work into parallel task (programmer or system)
  - One task provides a unit of work
  - Granularity?
- Scheduling (Runtime System)
  - Assignment of tasks to processors
  - Goal: full resource usage with little overhead

# Example: Fibonacci P-Fib

**if**  $n \leq 1$  **then**

| **return**  $n$

**else**

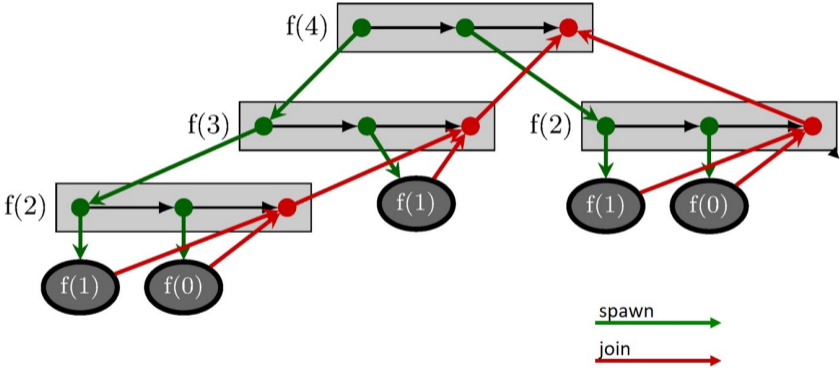
|  $x \leftarrow$  **spawn** P-Fib( $n - 1$ )

|  $y \leftarrow$  **spawn** P-Fib( $n - 2$ )

| sync

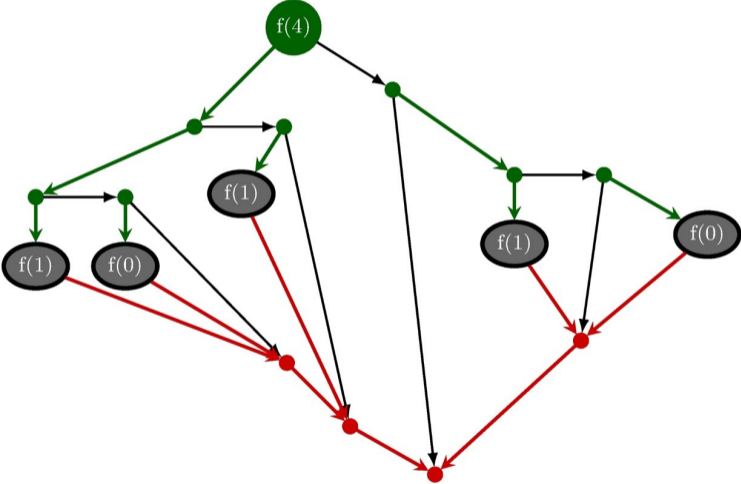
| **return**  $x + y$ ;

# P-Fib Task Graph



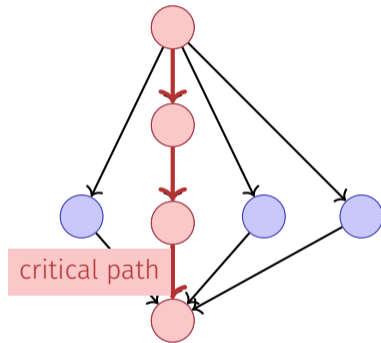


# P-Fib Task Graph



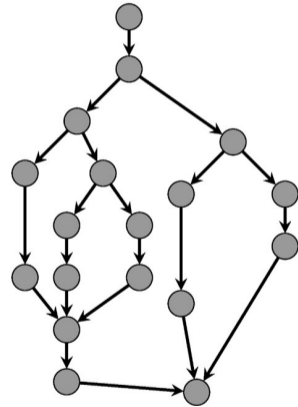
# Question

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



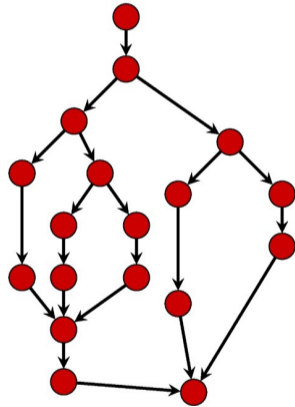
# Performance Model

- $p$  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





# Greedy Scheduler

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

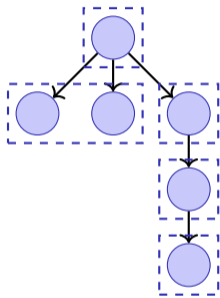
## *Theorem 45*

*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*

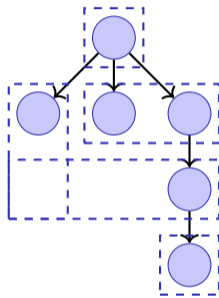
$$T_p \leq T_1/p + T_\infty$$

# Beispiel

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  $\geq \lfloor T_1/p \rfloor \cdot p + p = T_1 - T_1 \bmod p + p > T_1$ . 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

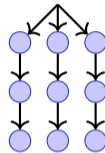
if  $p \ll T_1/T_\infty$ , i.e.  $T_\infty \ll T_1/p$ , then  $T_p \approx T_1/p$ .

## Fibonacci

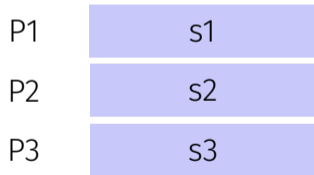
$T_1(n)/T_\infty(n) = \Theta(\phi^n/n)$ . For moderate sizes of  $n$  we can use a lot of processors yielding linear speedup.

# Granularity: how many tasks?

- #Tasks = #Cores?
- Problem if a core cannot be fully used
- Example: 9 units of work. 3 core.  
Scheduling of 3 sequential tasks.

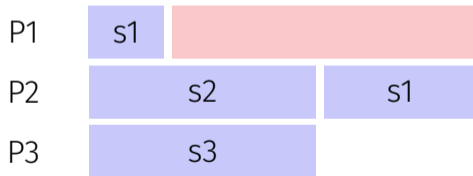


Exclusive utilization:



Execution Time: 3 Units

Foreign thread disturbing:



Execution Time: 5 Units

# Granularity: how many tasks?

- #Tasks = Maximum?
- Example: 9 units of work. 3 cores. Scheduling of 9 sequential tasks.



Exclusive utilization:

P1	s1	s4	s7
P2	s2	s5	s8
P3	s3	s6	s9

Execution Time:  $3 + \epsilon$  Units

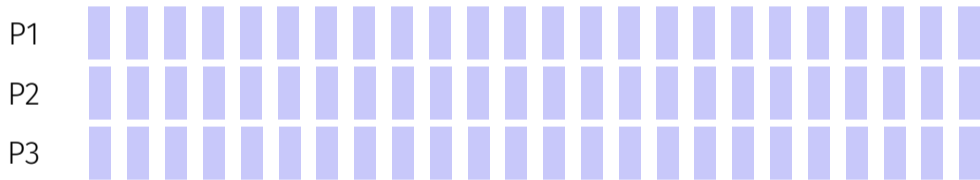
Foreign thread disturbing:

P1	s1			
P2	s2	s4	s5	s8
P3	s3	s6	s7	s9

Execution Time: 4 Units. Full utilization.

# Granularity: how many tasks?

- #Tasks = Maximum?
- Example:  $10^6$  tiny units of work.



Execution time: dominiert vom Overhead.

# Granularity: how many tasks?

Answer: as many tasks as possible with a sequential cutoff such that the overhead can be neglected.

# 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)

