

27. 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 is over ⁵³

⁵³"The Free Lunch is Over", a fundamental turn toward concurrency in software, Herb Sutter, Dr. Dobb's Journal, 2005

Moore's Law



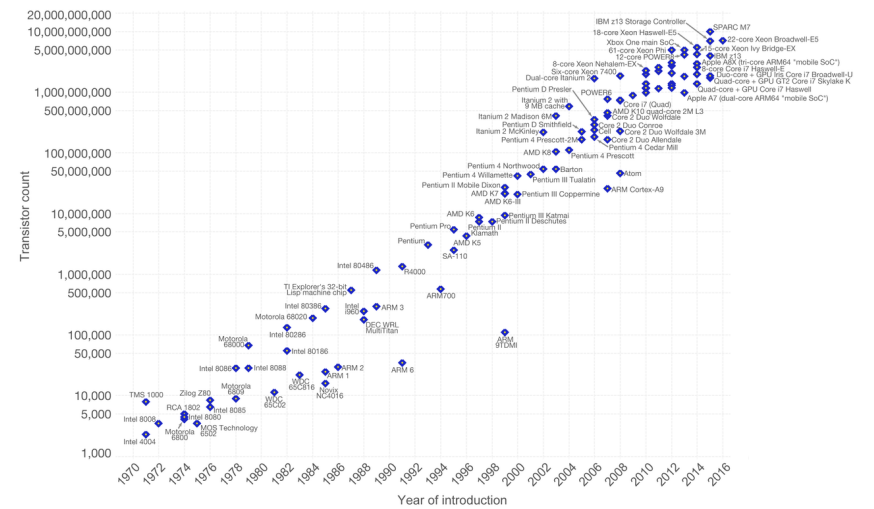
Gordon E. Moore (1929)

Observation by Gordon E. Moore:

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

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)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

Licensed under CC-BY-SA 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

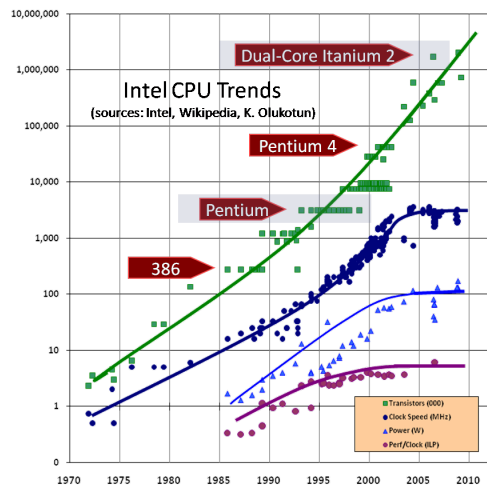
Today

- 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)

848

849

Trends



<http://www.gotw.ca/publications/concurrency-ddj.htm>

Multicore

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

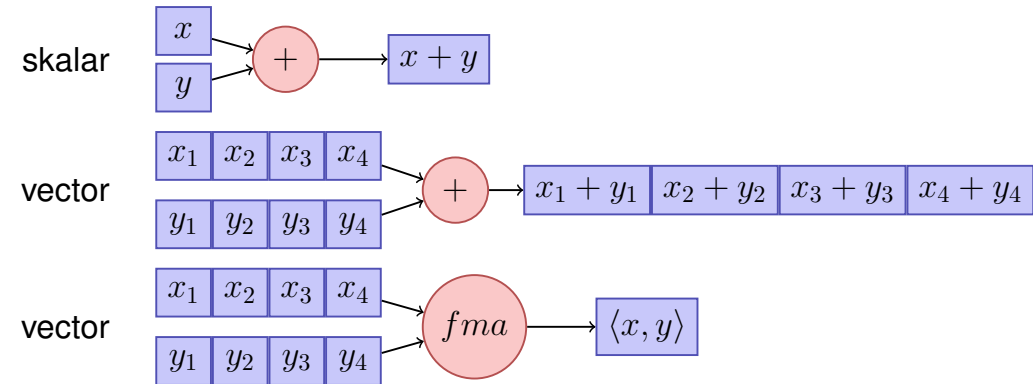
851

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)



852

853

Pipelining in CPUs



Multiple Stages

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

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

ILP – Instruction Level Parallelism

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

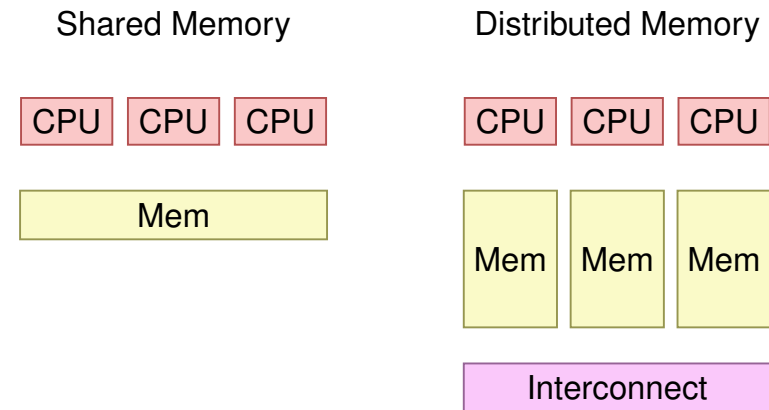
- Pipelining
- Superscalar CPUs (multiple instructions per cycle)
- Out-Of-Order Execution (Programmer observes the sequential execution)
- Speculative Execution ()

854

855

27.2 Hardware Architectures

Shared vs. Distributed Memory



856

857

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

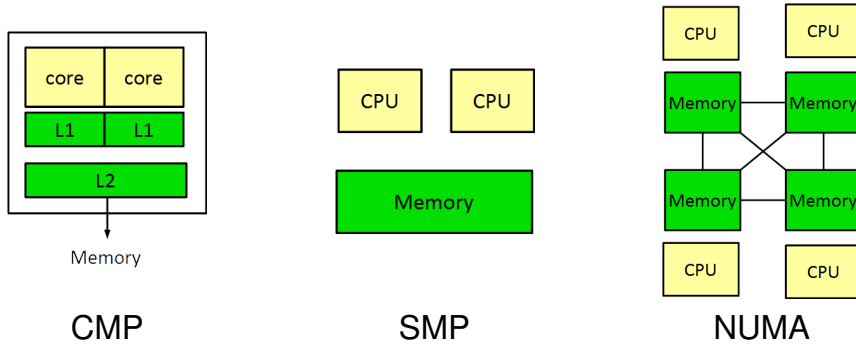
- Multicore (Chip Multiprocessor - CMP)
- Symmetric Multiprocessor Systems (SMP)
- Simultaneous Multithreading (SMT = Hyperthreading)
 - one physical core, Several Instruction Streams/Threads: several virtual cores
 - Between ILP (several units for a stream) and multicore (several units for several streams).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

Same programming interface

858

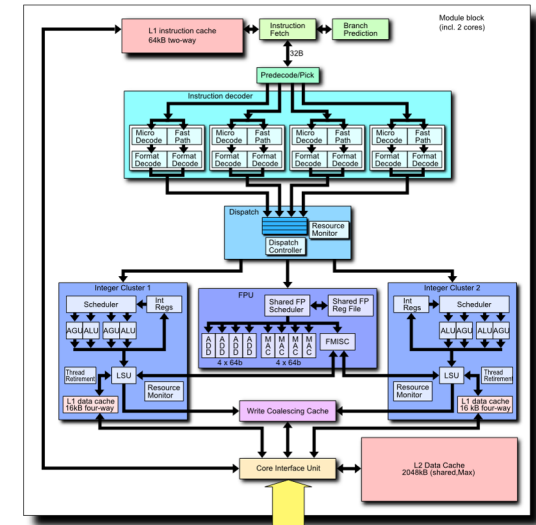
859

Overview



An Example

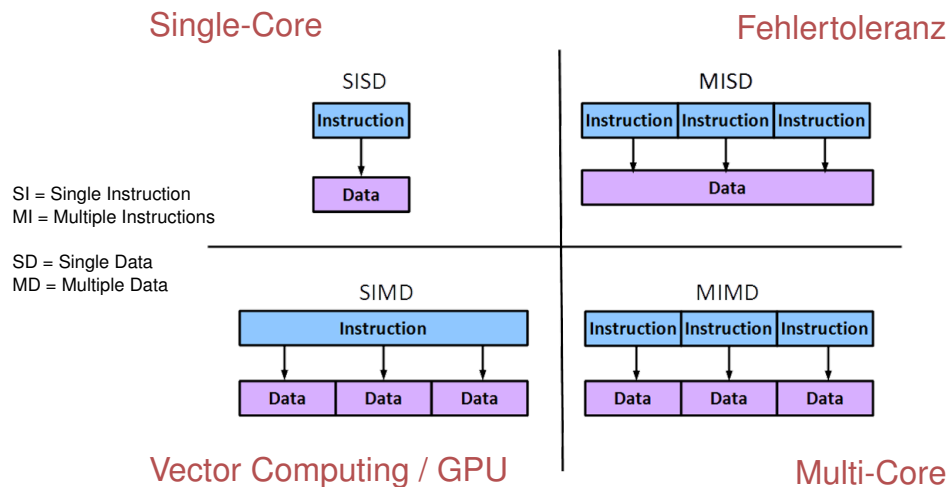
- AMD Bulldozer: between CMP and SMT
- 2x integer core
 - 1x floating point core



860

Wikipedia
861

Flynn's Taxonomy

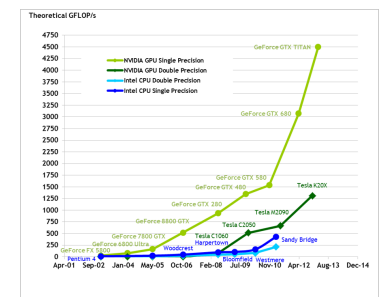


862

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



863

27.3 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

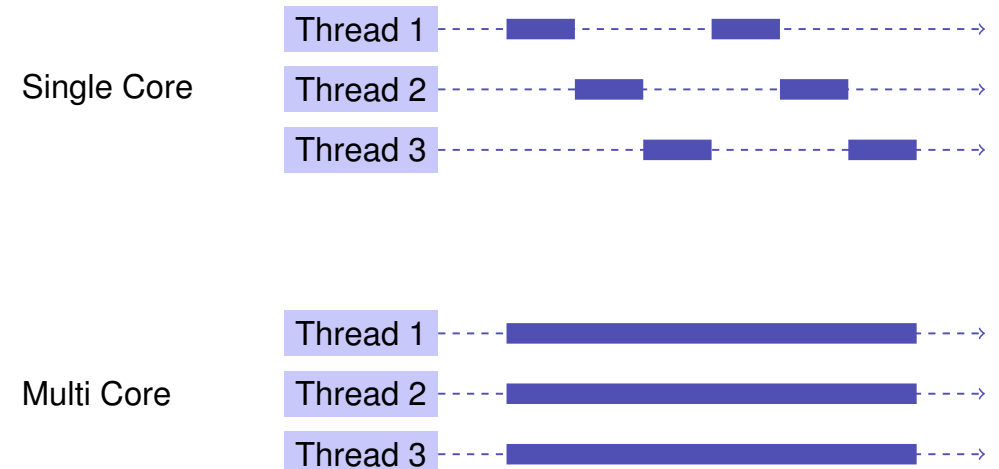
864

865

Why Multithreading?

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

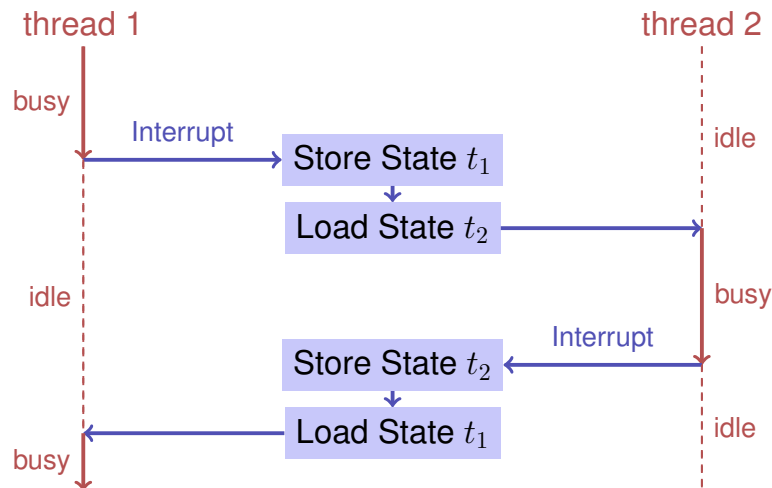
Multithreading conceptually



866

867

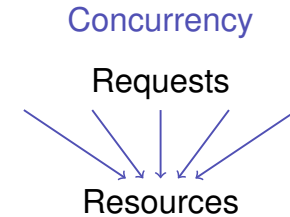
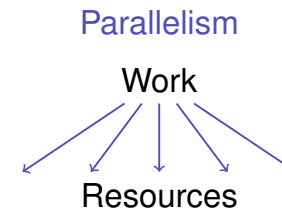
Thread switch on one core (Preemption)



868

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.



869

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.

870

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.



871

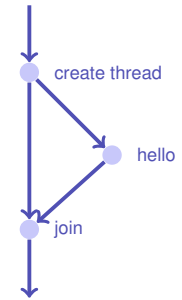
27.4 C++ Threads

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;
}
```



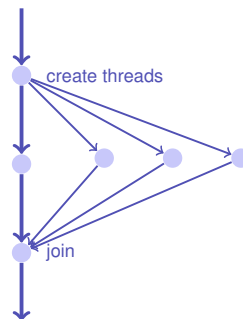
872

873

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
```

874

875

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
```

27.5 Scalability: Amdahl and Gustafson

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.

876

877

Scalability

In parallel Programming:

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

878

879

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 $S_p > p$

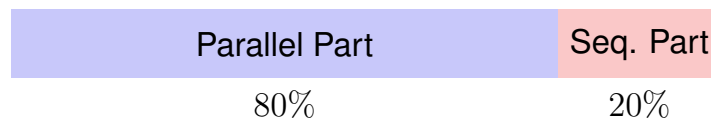
Efficiency: $E_p = S_p/p$

880

881

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 ($T_1 = W$):

$$T_1 = W_s + W_p$$

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

882

883

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 λ : $W_s = \lambda W$,
 $W_p = (1 - \lambda)W$:

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

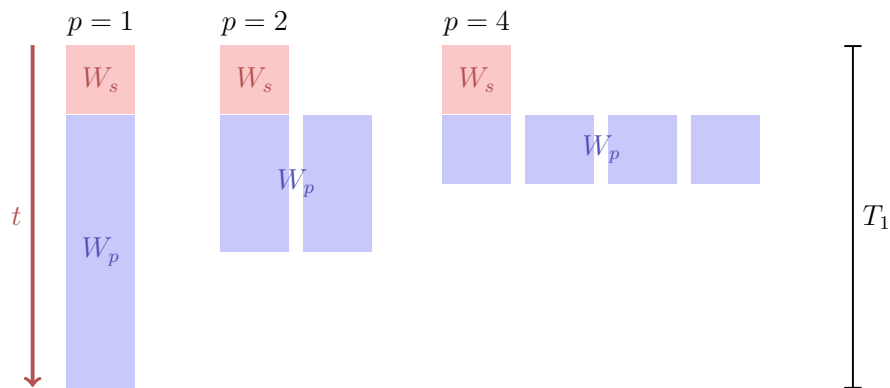
Thus

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

884

885

Illustration Amdahl's Law



886

887

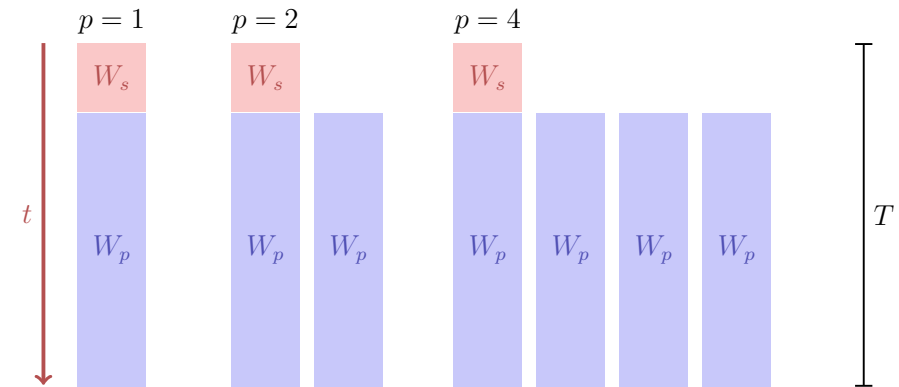
Amdahl's Law is bad news

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



888

889

Gustafson's Law

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

$$W_s + W_p = T$$

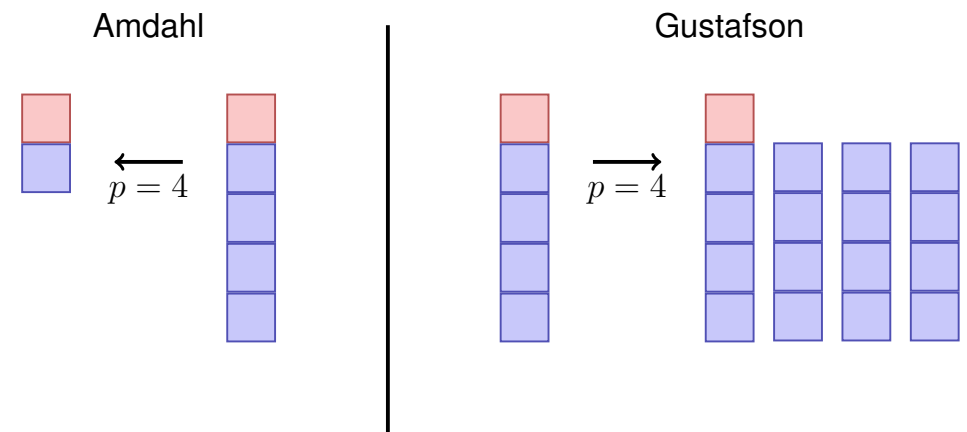
Work that can be executed by p processors in time T :

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

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



890

891

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.

892

27.6 Task- and Data-Parallelism

893

Parallel Programming Paradigms

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

894

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;
```

895

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);
}
```

896

Work Partitioning and Scheduling

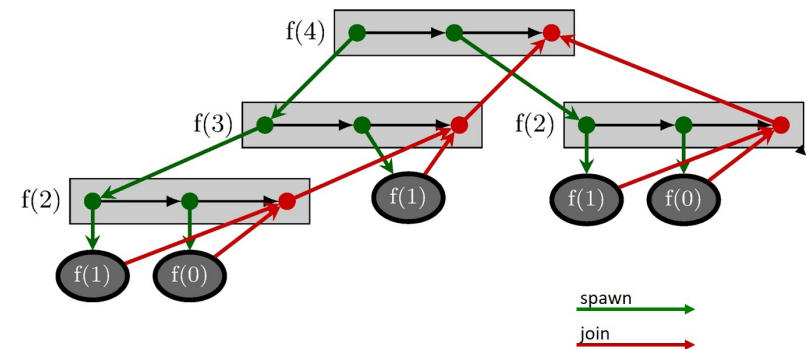
- Partitioning of the work into parallel task (programmer or system)
 - One task provides a unit of work
 - Granularity?
- Scheduling (Runtime System)
 - Assignment of tasks to processors
 - Goal: full resource usage with little overhead

897

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  $x + y$ ;
```

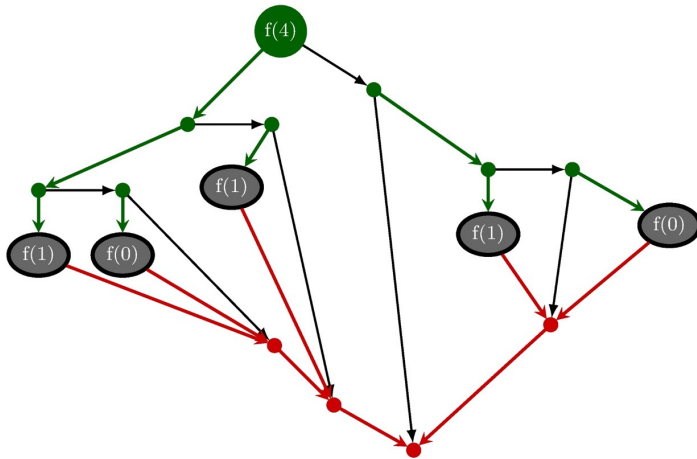
P-Fib Task Graph



898

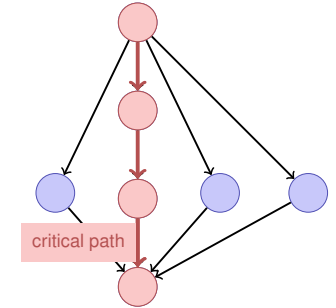
899

P-Fib Task Graph



Question

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

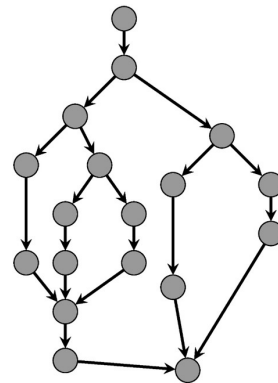


900

901

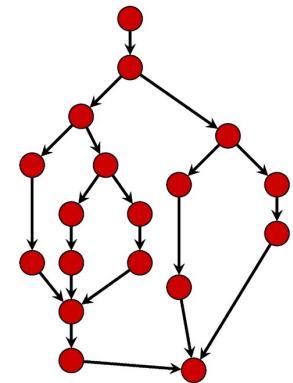
Performance Model

- 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



902

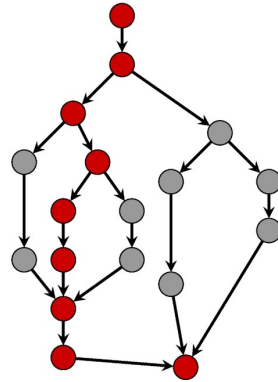
903

Performance Model

- T_∞ : *span*: critical path, execution time on ∞ processors. Longest path from root to sink.
- T_1/T_∞ : *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 available tasks.

Theorem

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

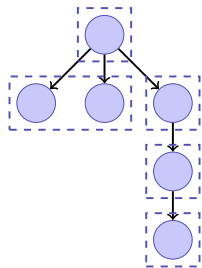
$$T_p \leq T_1/p + T_\infty$$

904

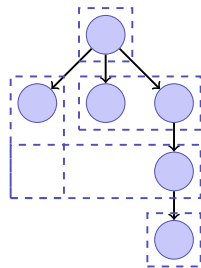
905

Beispiel

Assume $p = 2$.



$$T_p = 5$$



$$T_p = 4$$

906

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_∞ .

907

Consequence

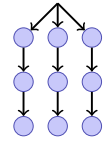
if $p \ll T_1/T_\infty$, i.e. $T_\infty \ll T_1/p$, then $T_p \approx T_1/p$.

Example Fibonacci

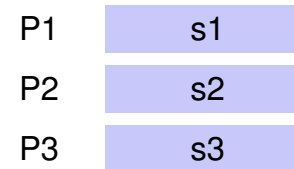
$T_1(n)/T_\infty(n) = \Theta(\phi^n/n)$. For moderate sizes of n we can use a lot of processors yielding linear speedup.

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.

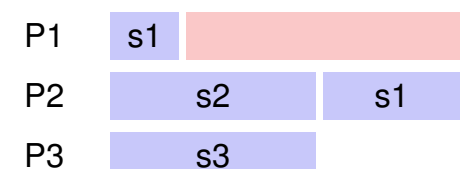


Exclusive utilization:



Execution Time: 3 Units

Foreign thread disturbing:



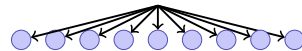
Execution Time: 5 Units

908

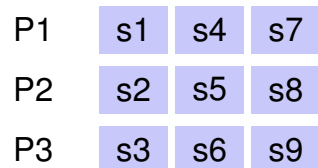
909

Granularity: how many tasks?

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

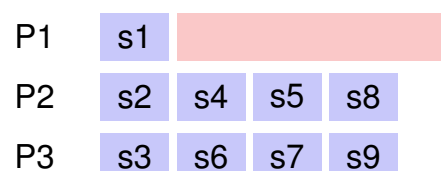


Exclusive utilization:



Execution Time: $3 + \varepsilon$ Units

Foreign thread disturbing:

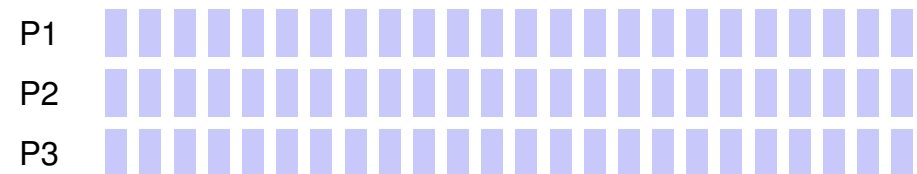


Execution Time: 4 Units. Full utilization.

910

Granularity: how many tasks?

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



Execution time: dominiert vom Overhead.

911

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)

