Introduction to Parallel and Distributed Programming

prof. Ing. Pavel Tvrdík CSc.

Department of Computer Systems
Faculty of Information Technology
Czech Technical University in Prague

Parallel and Distributed Programming (MIE-PDP)
Summer Semester 2020/21, Lecture 01
(Version Timestamp: 10. 2. 2021 20:36)

https://courses.fit.cvut.cz/MIE-PDP

Lecture Outline

1. Types of parallel computers
2. Models of Parallel Systems
3. Computational model
4. Programming models
5. Complexity Measures of Sequential Algorithms
6. Parallel time $T(n, p)$
7. Parallel speedup $S(n, p)$
8. Parallel cost $C(n, p)$
9. Parallel efficiency $E(n, p)$
10. Parallel scalability

List of lectures of MIE-PDP

1. Introduction to parallel and distributed computing
2. Introduction to OpenMP
3. Parallel algorithms for state space search
4. Performance tuning in OpenMP
5. Parallel algorithms in OpenMP
6. Parallel sorting in OpenMP
7. Introduction into MPI
8. Interconnection networks of parallel computers
9. Collective communication algorithms
10. Parallel algorithms in MPI I
11. Parallel algorithms in OpenMP/MPI II
12. MapReduce frameworks (principles, Hadoop, Spark)

Follow-up master courses:

- MIE-DSV (Distributed Systems and Computing), MIE-PAP (Parallel Computer Architectures), MIE-PRC (CUDA programming).

Parallel/distributed computers

Definition 1

A **parallel/distributed computer** is a collection of interconnected **computing nodes** that communicate and collaborate with the aim to solve faster large and high-performance computational tasks.

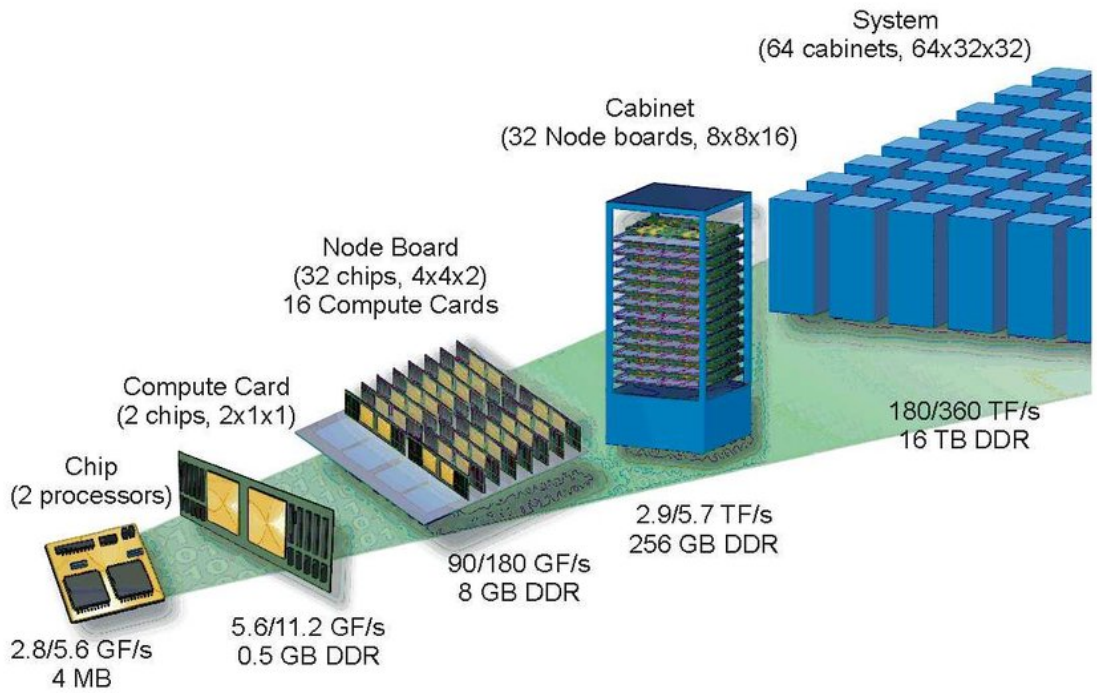
IBM Summit: 2nd most high-performance supercomputer on the planet (148 petaflops LINPACK, 2.4M CPU cores)

Source: https://en.wikipedia.org/wiki/Summit_(supercomputer)

Hierarchy of parallelism

- **Multithreaded cores**: one core can execute several threads simultaneously.
- **Multicore CPUs**: several (2, 4, 8, 16, ...) cores within 1 CPU.
- **GPUs**: thousands of smaller synchronized cores (e.g., NVidia).
- **Computing SMP** (Symmetric multiprocessors) **nodes**: several multicore CPUs with accelerators or not with (virtual) shared memory.
- **Computing clusters**: clusters (mostly Linux driven) of hundreds to tens-of-thousands of computing SMP nodes.
- **Tightly-coupled massively parallel supercomputers**: tens-of-thousands to hundreds-of-thousands computing SMP nodes interconnected via a proprietary interconnects with the distributed memory model.
- **Cloud computing** infrastructure, **data centers**, ...

An example of a parallelism hierarchy: IBM BlueGene/Q Sequoia



Source: `http://arstechnica.com/information-technology/2012/06/with-16-petaflops-and-1-6m-cores-doe-supercomputer-is-worlds-fastes`

Internet sources on parallel/high-performance computing

www.top500.org

- Updated every 6 months.
- All Top500 systems are Linux-based.
- The total performance of all 500 systems is 2.4 Eflop/s.
- World #1: Fugaku Fujitsu, Arm8 Volta (USA), 442 Pflop/s **HPL**, 7.6M cores.
- Europe #1: Juwels, Bull, 44 Pflop/s LinPack, 445K AMD EPYC cores.
- Czech #1: # 460 Salomon (IT4I in Ostrava), SGI, 1,5 Pflop/s, Intel Xeon Phi, 77K cores.
- Exa $= 10^{18}$, Peta $= 10^{15} = 1000$ Tera.

An example of a HPC application: Weather forecast

- **Region** of size $3000 \times 3000 \times 11\ km^3$ for the time of **2 days**.
- The region is decomposed into **segments** (e.g., using the finite elements method) of size $0.1 \times 0.1 \times 0.1\ km^3 \quad \Rightarrow \quad \#$ of segments $\sim 10^{11}$.
- Model parameters (temperature, wind speed) are computed with the time step of 30 minutes. Hence 2 days require 96 steps.
- **New** values of segment parameters are computed from the **previous** parameters of neighboring segments.
- Assume that the computation of 1segment parameters takes 100 ops.
- Then 1 **iteration** $=$ **update of parameter values** of all parameters in the whole region requires about $10^{11} \times 100 \times 96 \doteq 10^{15}$ ops.
- Single-core computer with 2Gflop/s requires 5×10^5 sec $\doteq 6$ **days**.
- And we must compute **thousands** of such iterations.
- Moreover, we get the memory problem: data do not fit into the memory of a single-core computer and must be swapped to a disk!!!
- To get a realistic weather forecast model, massively parallel computation is **the only feasible solution**.

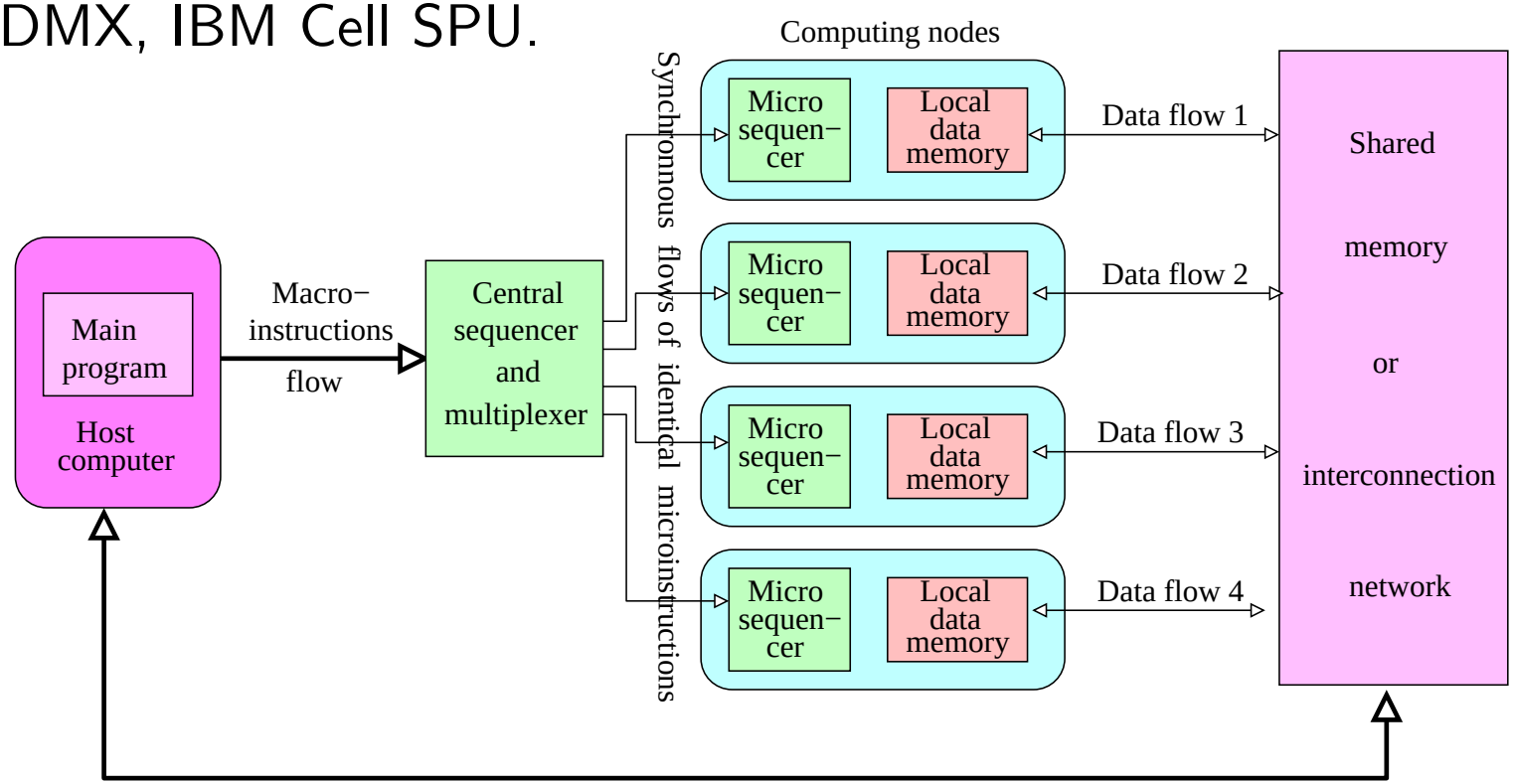
Models of Parallel Systems

- **Machine model**: HW, ISA, OS (skipped here)
- **Architecture model**: Data-and-instruction flows, memory organization, interconnection networks.
- **Computational model**: analytic model of the architecture for designing and evaluating parallel programs (RAM, PRAM, BSP, APRAM, ...)
- **Programming model**: semantics of parallel HLLs and environments, memory address space model,

Architecture models: Data and instruction flows

SIMD: Single Instruction Multiple Data

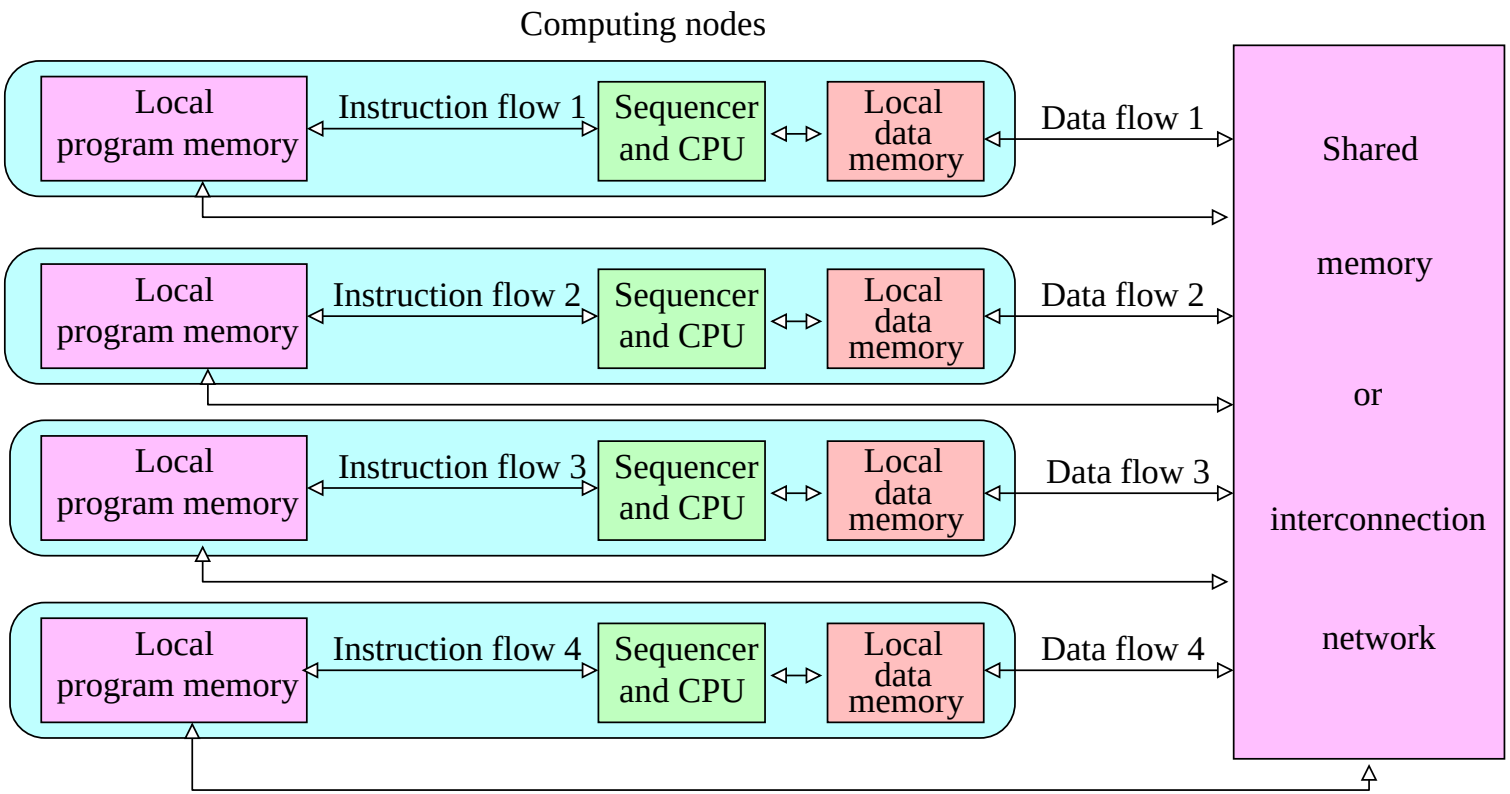
- Computing nodes have local data memories and can exchange data.
- All nodes receive synchronously the same stream of instructions.
- They can execute it or ignore it (based on local data).
- SIMDs today = **GPUs** and **vector extensions of CPU ISAs**: Intel MMX/SSEx/AVX, AMD 3DNow, SPARC VIS, Arm Neon, MPIS MDMX, IBM Cell SPU.



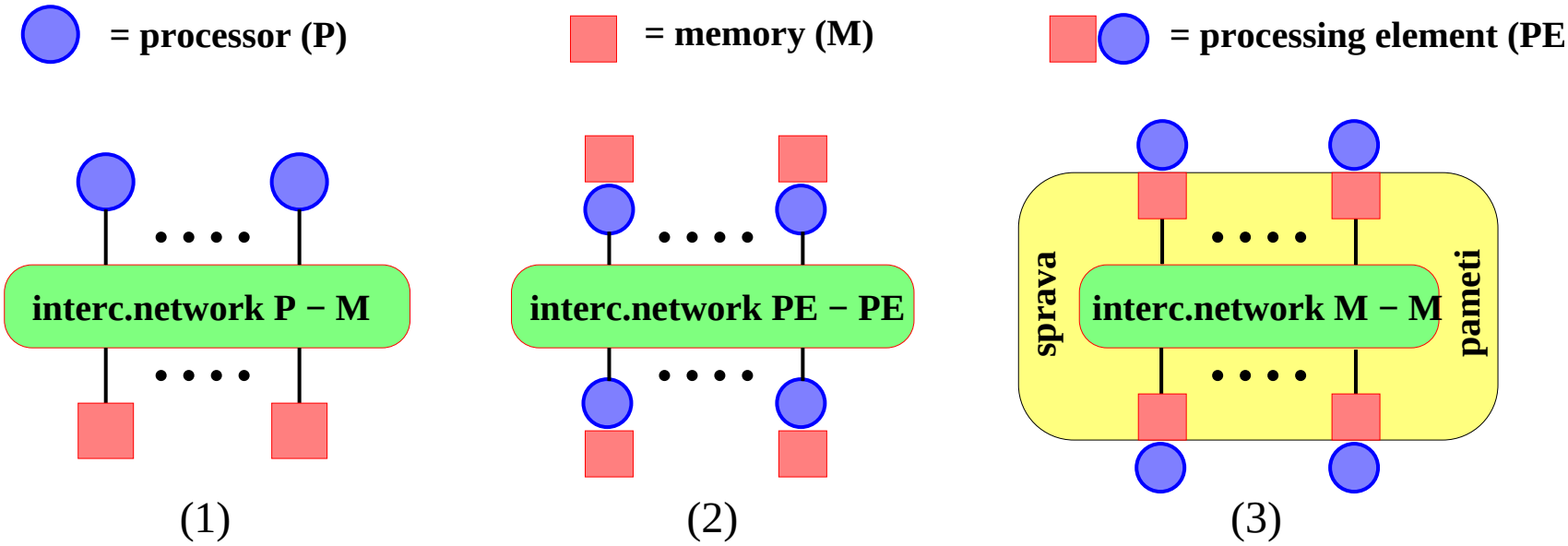
Architecture models: Data and instruction flows

MIMD: Multiple Instruction Multiple Data

- Each computing node is independent standalone computer with instruction and data memory.
- Individual nodes run asynchronously.
- All servers, clusters, massively parallel computers.



Architecture models: Memory organization



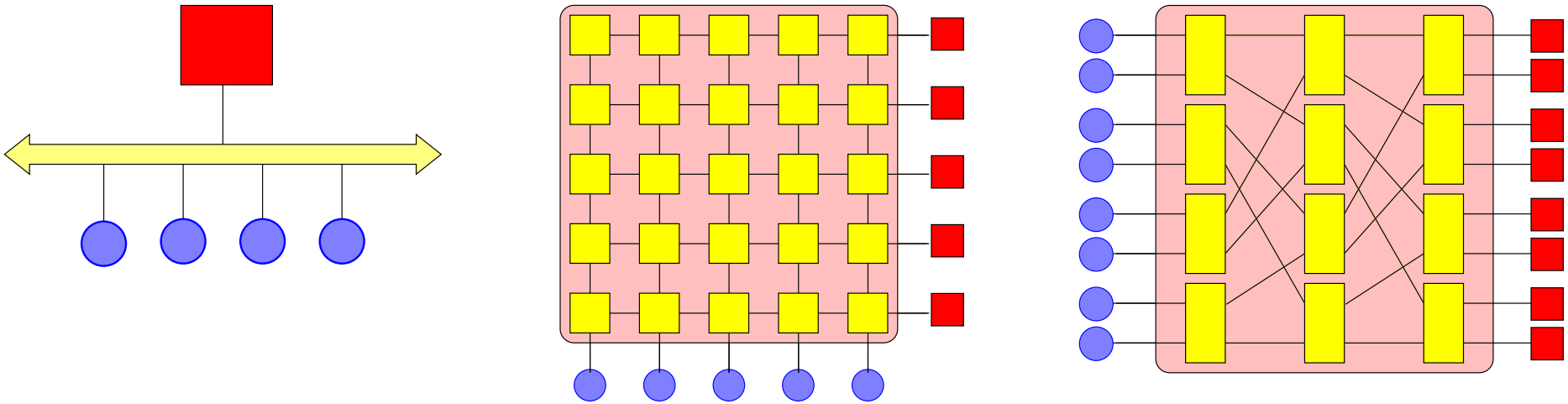
(a) parallel system with **shared memory** (SM) (= symmetric multiprocessors (SMPs), UMA)
- HW/SW communication = Read/Write in SM

(b) parallel system with **distributed memory** (= NUMA)
- HW/SW communication = Send/Receive

(c) parallel system with **virtual shared memory** (= distributed shared memory, CC-NUMA (Cache-Coherent))
- HW communication = Send/Receive, SW communication = R/W

Architecture models: Interconnection networks

Shared memory parallel systems



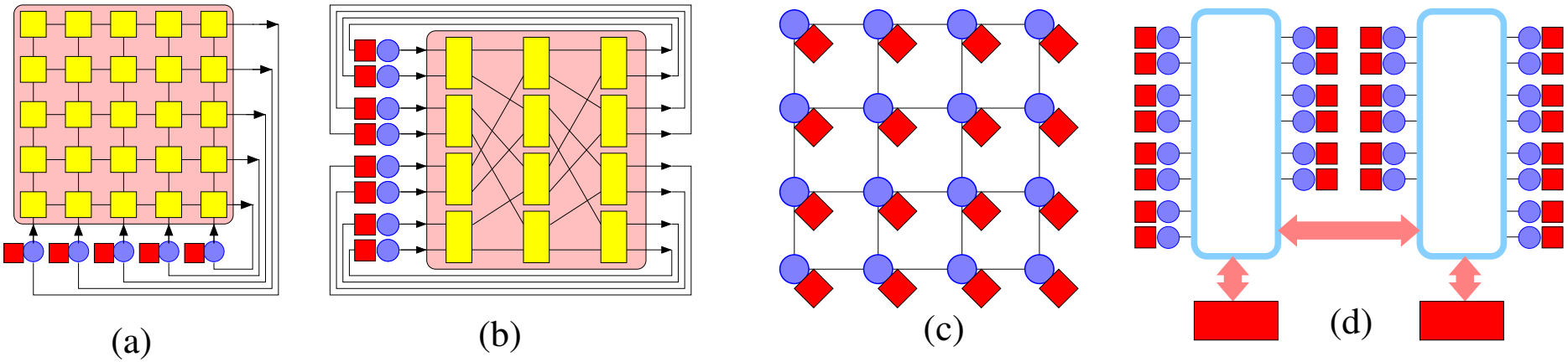
(a) shared bus (for only few Ps, not used today)

(b) (crossbar) switch,

(c) indirect multistage interconnection network (MIN)

Architecture models: Interconnection networks

Distributed memory parallel systems



(a) (crossbar) switch
(b) indirect multistage interconnection network (MIN)
(c) direct interconnection network
(d) hierarchical interconnection network (e.g., hierarchy of rings or switches)

Computational models: Parallel RAM (PRAM)

- A **set** of p RAM processors $P_1, P_2, \ldots, P_p$.
- Each P_i **knows** its index i and has its own **local registers/memory**.
- An array of n **shared memory** (SM) cells $M[1], M[2], \ldots, M[n]$.
- Each P_i can access **any** $M[j]$ in $O(1)$ **time**, unless access conflict.
- Handling of **access conflicts** must be **explicitly** defined (see below).
- **Input**: n items stored in n SM cells.
- **Output**: n' items stored in n' SM cells.
- PRAM processors execute **synchronously 3 types of operations** (one type at a time):
  - **Global Read** data from a SM cell (READ, `R`).
  - **Local computation:** (LOCAL, shortly `L`).
  - **Global Write** write data to a SM cell (WRITE, `W`).
- **Communication** = READ/WRITE of a shared memory cell.
- **Regular expressions:** For example, k iterations of a sequence READ, LOCAL, WRITE will be written as $\langle \mathtt{RLW} \rangle^k$.
- **Unit time** model: each operation `R`/`L`/`W` takes time 1.
- **Global time** model: `L` takes time 1 and `R`/`W` take constant time $d > 1$.

PRAM: Handling SM access conflicts

- **E**xclusive **R**ead **E**xclusive **W**rite (EREW-) PRAM: No two Ps are allowed to `R`/`W` the same SM cell simultaneously.
- **C**oncurrent **R**ead **E**xclusive **W**rite (CREW-) PRAM: Simultaneous `R`s of the same SM cell are allowed, but `W`s must be exclusive.
- **C**oncurrent **R**ead **C**oncurrent **W**rite (CRCW-) PRAM: Both simultaneous `R`/`W`s of the same memory cell are allowed.
  - Priority-CRCW-PRAM: The Ps are **assigned fixed** distinct priorities and the P with the **highest** priority is allowed to perform WRITE.
  - Arbitrary-CRCW-PRAM: One **randomly chosen** P is allowed to perform WRITE. The algorithm may make **no assumptions** about which P was chosen.
  - Common-CRCW-PRAM: All Ps are allowed to complete the WRITE operation iff all the values to be written are **equal**. Any algorithm for this model has to make sure that this condition is satisfied. If not, the machine state is undefined.

Computational models: Asynchronous PRAM (APRAM)

- The same operations **Global Read** (`R`), **Global Write** (`W`), and **Local** (`L`) as in PRAM.
- **Global time** model: operations `L` take time 1 and operations `R/W` take constant time $d > 1$.
- Ps work **asynchronously**, i.e., there is no central clock.
- Ps must be synchronized **explicitly** by the **barrier synchronization**: Each P stops at a logical point in the program until all processors arrive to this point. Then all Ps continue.
- **APRAM computation** = a sequence of **global phases** in which the Ps work asynchronously and which are separated by barrier synchronizations.

Two processors cannot access the same memory cell in the same global phase if at least one of them WRITEs into it.

Implementation of a barrier

1. **Central counter**: initiated to 0 and to the **incoming** phase, processes access it in mutual exclusion.
   1. A process arrives to a barrier, checks whether it is in the incoming phase and increments the counter.
   2. If the counter is $< p$, the process becomes **idle**.
   3. Otherwise, it sets the barrier to the **outgoing** phase and **activates** the idle processors.
   4. The **last activated** process sets the barrier into the **incoming** phase again.
   5. $B(p) = \Theta(dp)$
2. **Binary reduction tree**.
   1. Each process arrives to the barrier and checks, if it is in the **incoming** phase.
   2. Then it waits until the reduction in its subtree is completed.
   3. It then sends a signal to its parent and becomes **idle**.
   4. The root waits for the reduction form the both subtrees and switches the barrier to the **outgoing** phase.
   5. The processes activate in the reverse order.
   6. $B(p) = \Theta(d \log p)$

APRAM performance parameters

operation	time complexity
LOCAL	1
READ or WRITE	d
k consecutive READs or WRITEs	$d + k - 1$
barrier synchronization	$B(p)$

- d is a constant and $B(p)$ is a nondecreasing function of p.
- We assume that $2 \leq d \leq B(p) \leq dp$ and in practice: $B(p) = O(d \log p)$ or $B(p) = O(d \log p / \log d)$.
- **Consecutive** consecutive READs or WRITEs are pipelined on the shared memory bus and therefore k consecutive READs or WRITEs take time $d + k - 1$.

Programming models: Types of parallel constructs

- Instruction Level Parallelism (ILP)
- Data Parallelism
- Loop Parallelism
- Functional Parallelism
- Parallel Programming Patterns

Programming models: Instruction Level Parallelism (ILP)

- Multiple instructions can be executed in parallel at the same time if they are independent of each other.
- **Data dependency**:
  - **True data dependency** (RAW): one instruction needs data produced by the other instruction.
  - **Output dependency** (WAW): both instructions write to the same register.
  - **Anti-dependency** (WAR): one instruction must consume a register before it is overwritten by the other.
- **Superscalar** Ps extract data independent instructions from the instruction stream and schedule them dynamically to be executed on parallel functional units. The parallelism is **implicit**: the input is a sequential code.
- **VLIW** Ps use compilers to find static simultaneous scheduling of independent instructions.

Programming models: Data Parallelism

- Elements of a large data structure are distributed evenly among Ps that perform **synchronously** computation on its assigned part.
- The most common parallel data structures are **parallel arrays**.
- A typical construct is **array assignment** and **array arithmetic op**:

```
a[1:n]=b[0:n-1]+c[2:n+1].
```

- A **data-parallel programming language** is a HLL with data parallel constructs.
- The most significant example was **High-Performance Fortran** (HPF).
- These languages are not used nowadays, since more universal and flexible asynchronous SPMD model prevailed (see later).

Programming models: Loop Parallelism

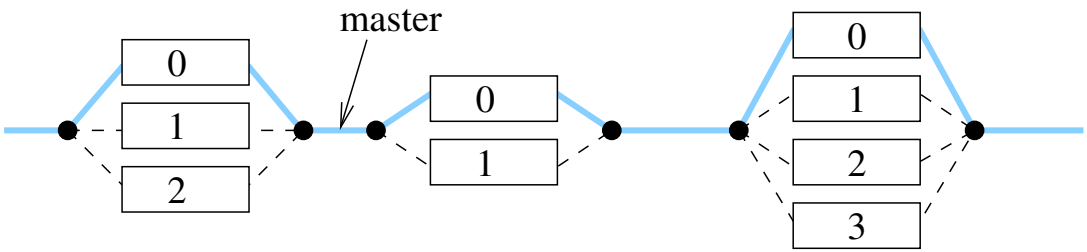
- Individual **data-independent cycle iterations** with distinct iterators can be executed simultaneously by individual Ps.
- A suitable **scheduling** is needed.
- This is another way of utilization of data parallelism.
- Data independency of iterations over a given iteration range must be guaranteed:
  - by the compiler (implicit loop parallelism) or
  - by the programmer (explicit loop parallelism).
- A very common construct to define parallel computation in OpenMP (the next Lecture).

Programming models: Functional Parallelism

- Also called **task parallelism**.
- The program splits into independent pieces of code, called **tasks** (modules, processes,...) that can be executed in parallel.
- Tasks form a **task graph** that defines data dependencies among them and **scheduling** (static or dynamic).
- These tasks can be structured blocks, functions, recursive calls to subsets of data, etc.
- Tasks can be sequential codes or parallelizable.
- Dynamic scheduling is often based on a **task pool** containing ready-to-execute tasks from where Ps that finished a previous task may grasp
- A very common tool to define parallel computation in OpenMP (the next lecture).

Programming models: Parallel Programming Patterns

- **Pipelining**: Threads participate in incremental ordered processing.
- **Fork-join**/**Parbegin-Parend**/**Cobegin-Coend**: see Lecture 2.



- **Single Program Multiple Data** (SPMD):
  - The most common programming pattern (OpenMP, MPI).
  - All threads execute the same static code asynchronously.
  - Synchronization must be done via explicit synchronizing operations.
  - Thread unique IDs are used in the code to differentiate the execution.
- **Master-Slave** or **Master-Worker**: Master assigns work, Slaves work and report results.
- **Server-Client**: Clients request work, the Server replies.
- **Producer-Consumer**: Producer threads provide tasks to a buffer for consumer threads.

Complexity Measures of Sequential Algorithms

- $T_A^K(n)$ = **time complexity/running time** of a seq. algorithm A solving problem K on input data set of size n.
- $SL^K(n)$ = the **lower bound** on the time complexity of any seq. alg. solving problem K (= the worst-case running time of any sequential algorithm solving K cannot be better).
- The **trivial lower bound** is given by the size of **input/output** data set n.
- $SU^K(n)$ = the **upper bound** on the time complexity to solve problem K (=the worst-case running time of the **fastest known** seq. alg. solving K.

Optimality of sequential algorithms

- Sequential algorithm A for solving problem K is **optimal** iff
$$T_A(n) = \Theta(SU^K(n)) = \Theta(SL^K(n)).$$
- Example: Let K_1 = **comparison-based sorting**. Then
$$SL^{K_1}(n) = \Omega(n \log n)$$
(the minimal depth of a binary tree with $n! = O(n^n)$ leaves. Hence: MergeSort, HeapSort with $T_{sort}(n) = O(n \log n)$ **are optimal**.
- Sequential algorithm A for solving K is **the best known** iff
$$T_A(n) = \Theta(SU^K(n)) = \omega(SL^K(n)).$$
- Example: Let K_2 = **matrix-matrix multiplication** $A_{n,n} \times B_{n,n}$.
Then there exist **no optimal** algorithms, because:
$SL^{K_2}(n) = \Omega(n^2)$ = the trivial lower bound.
$SU^{K_2}(n) = O(n^p)$, $2 < p < 3$, (Strassen $p = 2.81$, Coppersmith-Winograd $p = 2.376$).

Parallel running time $T(n, p)$

- **Parallel time complexity** of a parallel algorithm will depend not only on n, but also on **the number of cores/threads** p.
- We assume by default that

  $p =$ the # of processors $=$ the # of cores $=$ the # of threads .

Definition 2 (Parallel time)

$T(n, p) =$ the time elapsed from the beginning of a parallel computation until the last thread finishes the execution.

- $T(n, p)$ of parallel algorithms implemented a parallel computer, however, depends strongly on its **architecture** and **runtime**:
  - **communication overhead**: data exchanges or **work distribution** among cores.
  - **synchronization overhead**: respecting data dependencies, resolving access control to shared data, or thread handling costs.
  - the amount of **available parallelism** wrt the # of available cores/threads.

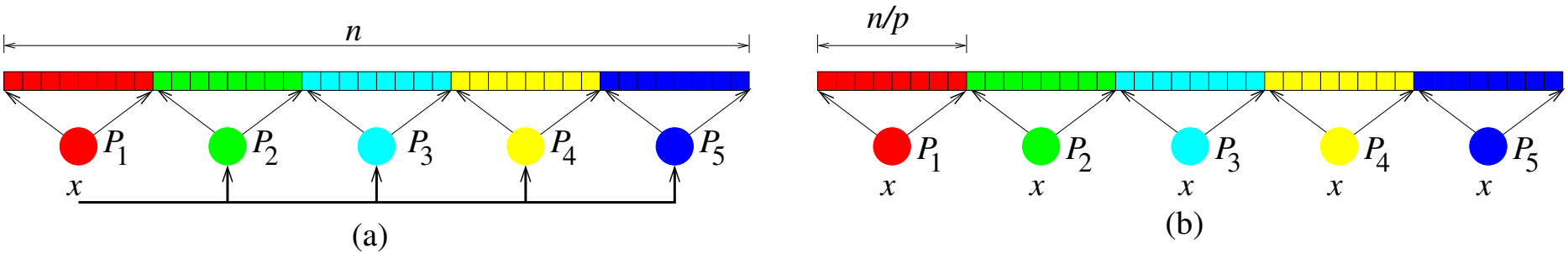
A trivial example of a parallel time analysis

Example 3

Parallel search of an item x in an unsorted shared input array A of n distinct items using p SMP cores. Since items in A are unique, at most 1 thread can report success.

We assume that any memory cell can be accessed only by 1 thread at a time.

$$T(n,p) = \underbrace{O(\log p)}_{\text{(a) broadcast } x} + \underbrace{O(n/p)}_{\text{(b) local search}} + \underbrace{O(1)}_{\text{(c) store the result}}$$



- The number of threads is a new degree of freedom.
- We should ask questions such as: Given n and a parallel computer, what is the **best choice of p to achieve the minimal time**?

Parallel speedup $S(n, p)$

Definition 4 (Parallel speedup)

$$S(n, p) = \frac{SU(n)}{T(n, p)} \leq p.$$

Parallel speedup is **linear** iff

$$S(n, p) = \Theta(p).$$

- Linear speedup is the ultimate goal of parallel programming:

  If p **increases** k times, we expect $T(n, p)$ to **decrease** also k times.

- Unfortunately, this is hard to guarantee in general.
- It depends on the measure of data independence of subtasks (parallelizability of their execution).

Superlinear speedup

- Exceptionally, we can experience even better than linear speedup.
- There are two typical situations in which this can happen:
  - A sequential algorithm needs more memory than the uniprocessor system can provide, whereas the cumulative memory of all cores of a parallel system is large enough to store all the data of the corresponding parallel algorithm. Therefore, during the parallel execution time-consuming disk swapping can be avoided. That is, a superlinear speedup is caused by specific HW characteristics that put sequential algorithms at a disadvantage.
  - The anomaly of the parallel search in a state spaces: see Lecture 3 Slides 20 and 21.

Parallel time lower bound $L^K(n,p)$

Definition 5 (Parallel time lower bound)

Given a problem K, the lower bound on the parallel time for p threads is

$$L^K(n,p) = \frac{SL^K(n)}{p}.$$

Example 6

The parallel lower bound on comparison-based sorting of n numbers with $p = n$ threads is

$$L(n,n) = \frac{\Omega(n \log n)}{n} = \Omega(\log n).$$

- This bound is purely theoretical, in practice always $p \ll n$.

Parallel cost $C(n, p)$

- Most parallel architectures allocate dedicated cores to the parallel computations **statically**.
- A parallel computation starts by creating a corresponding number of threads and they are used to perform the computation until the end, even if not all of them are utilized all the time.
- Some of them may be idle for some part of the computational time.
- Therefore, an important quality measure of a parallel algorithm is the **processor-time product**, also called the **parallel cost**.

Definition 7 (Parallel cost)

$$C(n, p) = p \times T(n, p).$$

Lemma 8

$$C(n, p) = \Omega(SU(n)).$$

Cost-optimal parallel algorithms

Definition 9 (Cost optimality)

A parallel algorithm has an optimal cost if

$$C(n, p) = O(SU(n)).$$

Corollary 10

It follows from Lemma 8 that a parallel cost is optimal iff

$$C(n, p) = \Theta(SU(n)).$$

Parallel efficiency $E(n,p)$

- Given a number of cores dedicated to a parallel computation, we may ask what is the **relative utilization** of these computational resources during the course of the parallel computation. This is called **parallel efficiency**.
- The communication and synchronization overhead will make the efficiency always less than 100%.

Definition 11 (Parallel efficiency)

$$E(n,p) = \frac{SU(n)}{C(n,p)} \leq 1.$$

Lemma 12

$E(n,p)$ is the speedup per core:

$$E(n,p) = \frac{SU(n)}{C(n,p)} = \frac{S(n,p) \times T(n,p)}{p \times T(n,p)} = \frac{S(n,p)}{p} \leq 1.$$

Parallel optimality

Definition 13 (Constant efficiency)

Given a constant $0 < E_0 < 1$, we say that a parallel algorithm has a **constant efficiency** if $E(n, p) \geq E_0$.

Lemma 14 (Parallel performance optimality)

It follows from the definitions that a parallel algorithm

is **cost-optimal**

$\Leftrightarrow$

it has a **linear speedup**

$\Leftrightarrow$

it has a **constant** efficiency.

An example: parallel binary reduction

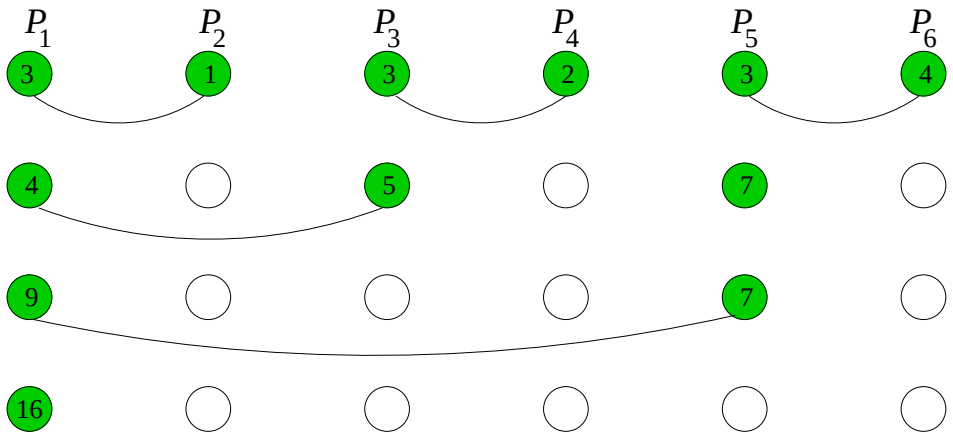
Example 15

(**Parallel** Σ). Compute the sum $\sum_{i=1}^{n} a_i$ of n input numbers $a_1, \ldots, a_n$ on a fully-connected parallel computer with $p = n$ cores $P_1, \ldots, P_n$.

Assumptions:

1. **Unit-time** computational model: both local arithmetic operations within a core and sending a number between two cores take unit time.
2. Initially, P_i stores a_i in its local register.

Algorithm `ParAdd`:



Performance analysis of `ParAdd`

$SL(n) = \Theta(n)$	$T(n, n) = \Theta(\log n)$	$C(n, n) = \Theta(n \log n)$
$SU(n) = \Theta(n)$	$S(n, n) = \Theta(n/\log n)$	$E(n, n) = \Theta(1/\log n)$

Discussion of the result:

- Adding n numbers with n cores is **not cost-optimal**.
- Intuitive explanation: **poor** use of **too many** cores, the number of cores doing useful work decreases **exponentially**.
- SOLUTION? To **better scale** p with n to keep **all cores busy most of the time**!!!!.
- The **scalability** issue:

What is the best relationship between p and n?

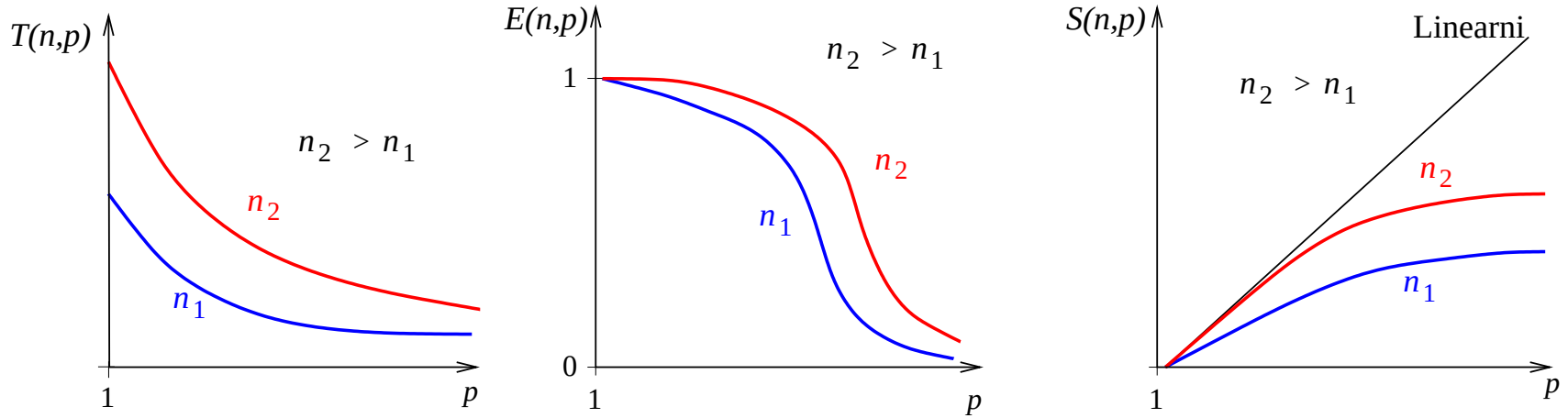
$$p = f(n) = ?$$

We only know that

$$p = \omega(1) \qquad \text{and} \qquad p = o(n).$$

Parallel scalability

- Consider a parallel algorithm A for fixed n and p growing from 1.
- Typical dependence of $T(n,p), E(n,p), S(n,p)$ on p is on the Figures.



- If $p = 1$, the efficiency is the best, but the time ($SU_A(n)$) is the worst.
- If p grows with n fixed, the time decreases, but after some limit, the time does not improve $\Rightarrow$ the efficiency decreases.

Definition 16 (Parallel scalability)

Scalability = the property of a parallel algorithm to keep the parallel optimality (see Slide 36) if both p and n grow or shrink.

Amdahl's law for saturation of parallelizability

- Every sequential algorithm A with time $T_A(n)$ on data of size n consists **proportionally** of
  - an **inherently sequential fraction** $0 < f_s < 1$ being that will be executed always by only 1 thread and
  - remaining **parallelizable fraction** $1 - f_s$.
- Let A with fixed n be executed in parallel using p threads.
- Then the speedup of parallel execution of A with p threads will be

$$S(n,p) = \frac{T_A(n)}{f_s \cdot T_A(n) + \frac{1-f_s}{p} \cdot T_A(n)} = \frac{1}{f_s + \frac{1-f_s}{p}} \leq \frac{1}{f_s}$$

- No matter how many threads are used, the speedup is bounded by $\frac{1}{f_s}$.
- For example, if $f_s = 10\%$, then $S(n,p) \leq 10$ for any p.
- After certain value of p, adding further processors makes no sense, since there is **not enough parallel work to do** – see the graphs on the previous Slide.

Gustafson's law

- Amdahl's law says that a fixed size problem provides a limited amount of parallelism and therefore it puts a limit on a reasonable number of parallel threads to execute it.
- **Gustafson's law** says that with growing p we should scale up the problem size n correspondingly.
- Then the inherently sequential part will take always **constant time** t_{seq} independently of p (I/O operations, initialization) whereas the inherently parallel part $t_{\mathrm{par}}(n,p)$ will **speedup linearly** with the number of processors in time.
- Then

$$S(n,p) = \frac{t_{\mathrm{seq}} + t_{\mathrm{par}}(n,1)}{t_{\mathrm{seq}} + t_{\mathrm{par}}(n,p)}$$

.

Gustafson's law

- Assume that the parallel part is perfectly parallelizable. Then

$$t_{\mathrm{par}}(n,1) = SU(n) - t_{\mathrm{seq}} \text{ and } t_{\mathrm{par}}(n,p) = \frac{SU(n)-t_{\mathrm{seq}}}{p}.$$

- Then

$$S(n,p) = \frac{t_{\mathrm{seq}} + SU(n) - t_{\mathrm{seq}}}{t_{\mathrm{seq}} + \frac{SU(n)-t_{\mathrm{seq}}}{p}} = \frac{\frac{t_{\mathrm{seq}}}{SU(n)-t_{\mathrm{seq}}} + 1}{\frac{t_{\mathrm{seq}}}{SU(n)-t_{\mathrm{seq}}} + \frac{1}{p}}.$$

- Therefore,

$$\lim_{n\to\infty} S(n,p) = p$$

for any monotonically increasing function $SU(n)$ (e.g., linear, polynomial).

Parallel scalability summary

Parallel Scalability:

- ... is the potential of a parallel computer to be enlarged in order to accommodate a problem size growth.
- ... expresses that larger problems can be solved in the same time as smaller problems if sufficient p is utilized.

In practice, there are two types of scalability:

- **Strong scalability** measures the capability of a parallel algorithm for fixed n to achieve linear speedup with increasing p. (Amdahl's law puts strong limits to this.)
- Alternatively: Strong scalability is the measure of efficiency decrease if p increases while n is fixed.
- **Weak scalability** defines how the parallel time varies with p for fixed n/p.
- Alternatively: Weak scalability is the measure of growth of n such that a fixed efficiency is preserved when p increases.

Isoefficiency functions ψ_1, ψ_2

Definition 17

Given constant $0 < E_0 < 1$, then isoefficient function

- ψ_1 is the **asymptotically minimal function** such that $\forall n_p = \Omega(\psi_1(p)) : E(n_p, p) \geq E_0$.
- ψ_2 is the **asymptotically maximal function** such that $\forall p_n = O(\psi_2(n)) : E(n, p_n) \geq E_0$.

Note 18

- Amdahl's law is a simple argument why $p = \omega(\psi_2(n))$ threads saturate the parallelism of a problem for fixed n.
- Gustafson's law is a simple argument that if the problem size grows as $n = \Omega(\psi_1(p))$, the efficiency will be sufficiently good.