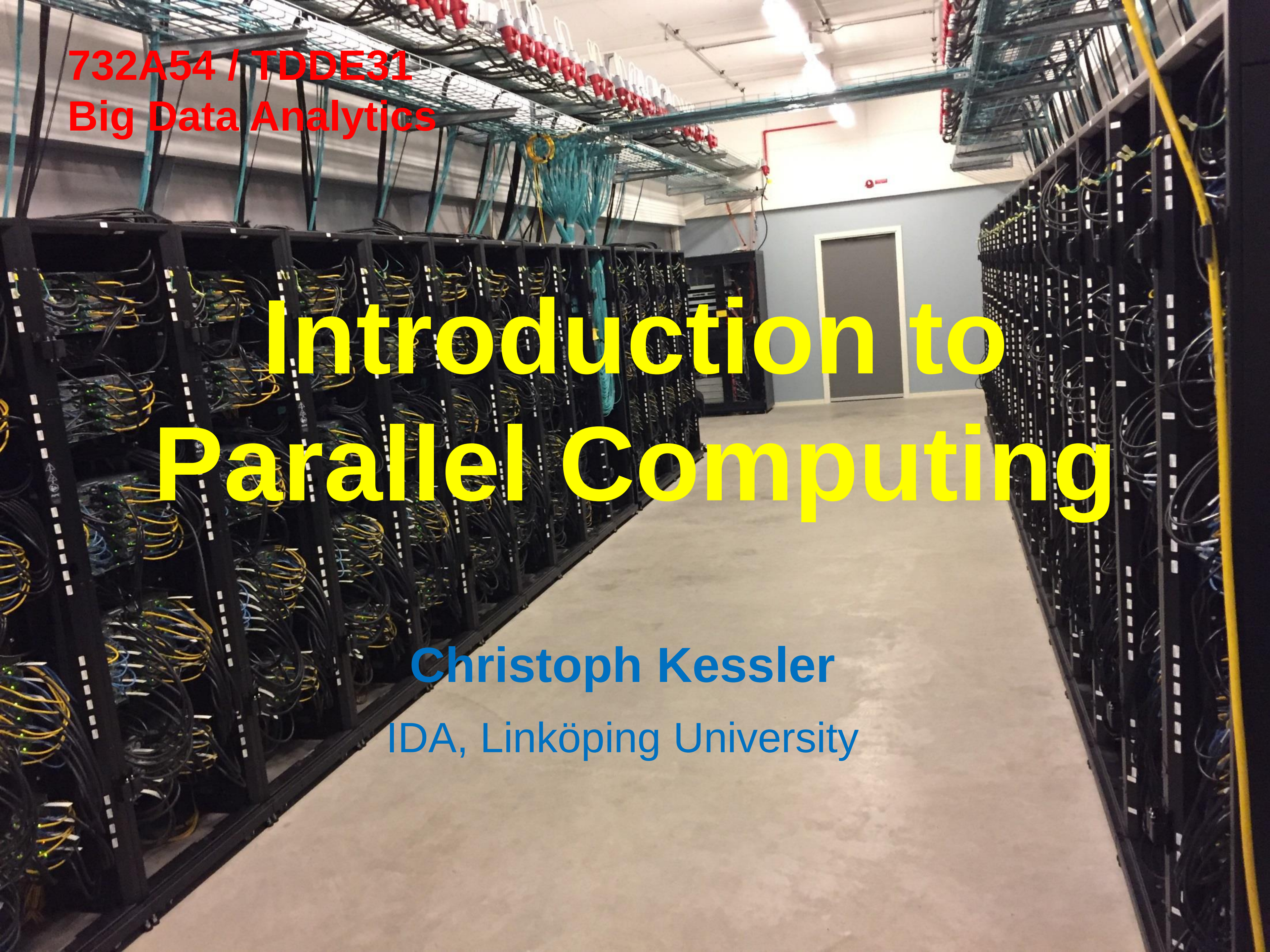


732A54 / TDDE31
Big Data Analytics

Introduction to Parallel Computing

Christoph Kessler

IDA, Linköping University

Traditional Use of Parallel Computing: Large-Scale HPC Applications

- **High Performance Computing (HPC)**
  - Much computational work
    (in FLOPs, floatingpoint operations)
  - Often, large data sets
  - E.g. climate simulations, particle physics, engineering, sequence matching or proteine docking in bioinformatics, …
- Single-CPU computers and even today's multicore processors cannot provide such massive computation power
- Aggregate LOTS of computers → **Clusters**
  - Need scalable parallel algorithms
  - Need exploit multiple levels of parallelism

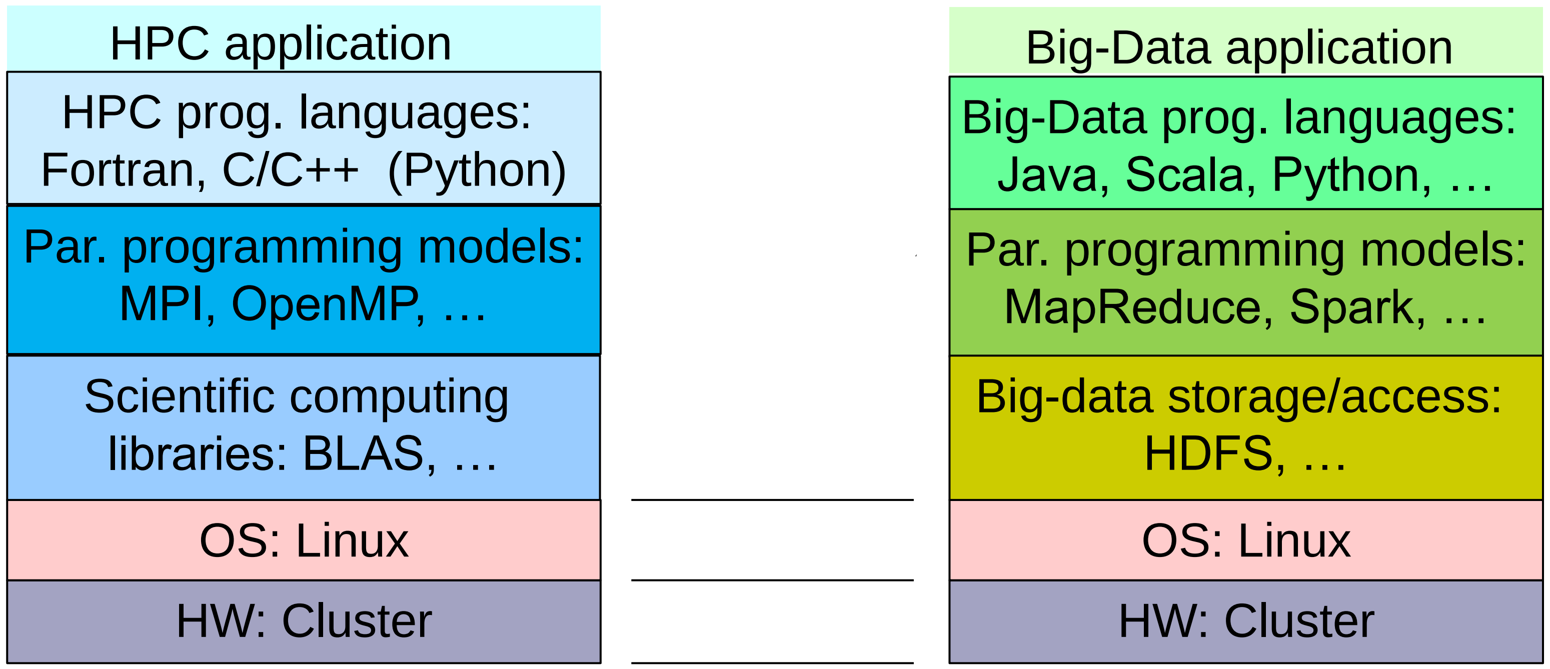
More Recent Use of Parallel Computing: Big-Data Analytics Applications

- **Big Data Analytics**
  - Data access intensive (disk I/O, memory accesses)
    - Typically, very large data sets (GB … TB … PB … EB …)
  - Also some computational work for combining/aggregating data
  - E.g. data center applications, business analytics, click stream analysis, scientific data analysis, machine learning, …
  - Soft real-time requirements on interactive querys
- Single-CPU and multicore processors cannot provide such massive computation power and I/O bandwidth+capacity
- Aggregate LOTS of computers → **Clusters**
  - Need scalable parallel algorithms
  - Need exploit multiple levels of parallelism
  - Fault tolerance

NSC Tetralith

HPC vs Big-Data Computing

- Both need **parallel computing**
- **Same kind of hardware** – Clusters of (multicore) servers
- Same OS family (Linux)
- **Different programming models**, languages, and tools



HPC application
HPC prog. languages: Fortran, C/C++ (Python)
Par. programming models: MPI, OpenMP, …
Scientific computing libraries: BLAS, …
OS: Linux
HW: Cluster
Big-Data application
Big-Data prog. languages: Java, Scala, Python, …
Par. programming models: MapReduce, Spark, …
Big-data storage/access: HDFS, …
OS: Linux
HW: Cluster

→ Let us start with the common basis: Parallel computer architecture

Parallel Computer

A parallel computer is a computer consisting of

+ two or more processors
  that can cooperate and communicate
  to solve a large problem faster,

+ one or more memory modules,

+ an interconnection network
  that connects processors with each other
  and/or with the memory modules.

Multiprocessor: tightly connected processors, e.g. shared memory

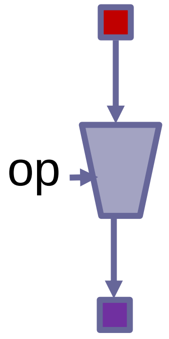
Multicomputer: more loosely connected, e.g. distributed memory

Parallel Computer Architecture Concepts

Classification of parallel computer architectures:

- by control structure
  - SISD, SIMD, MIMD
- by memory organization
  - in particular, Distributed memory vs. Shared memory
- by interconnection network topology

Classification by Control Structure [Flynn'72]



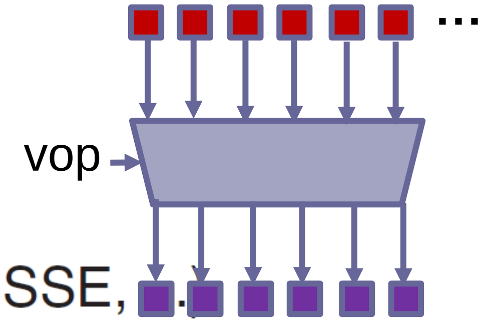
SISD single instruction stream, single data stream

+ sequential. OK where performance is not an issue.

SIMD single instruction stream, multiple data streams

Common clock, common program memory, common program counter.

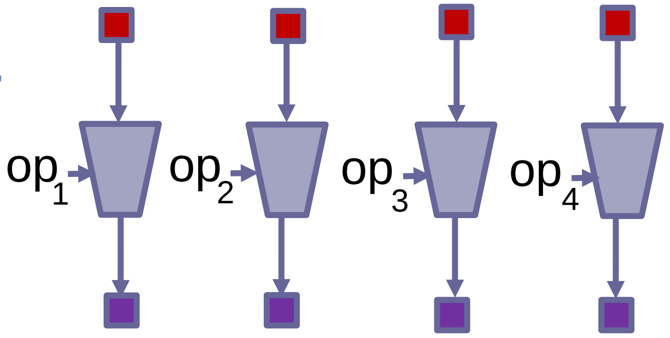
+ VLIW processors
+ traditional vector processors
+ traditional array computers
+ SIMD instructions on wide data words (e.g. Altivec, SSE, ...)



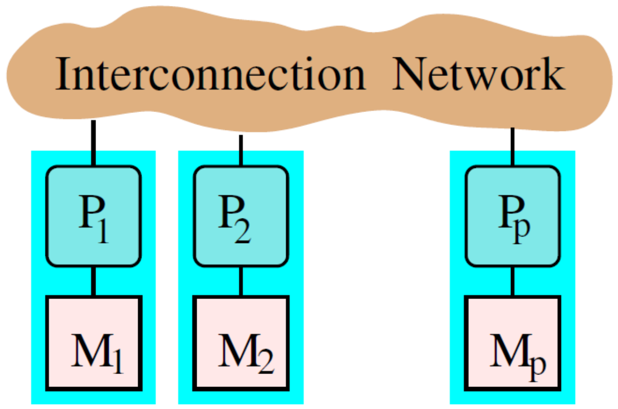
MIMD multiple instruction streams, multiple data streams

Each processor has its own program counter.

Hybrid forms

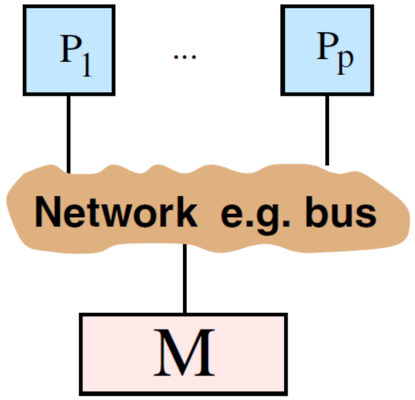


Classification by Memory Organization



Distributed memory system

e.g. (traditional) HPC cluster



Shared memory system

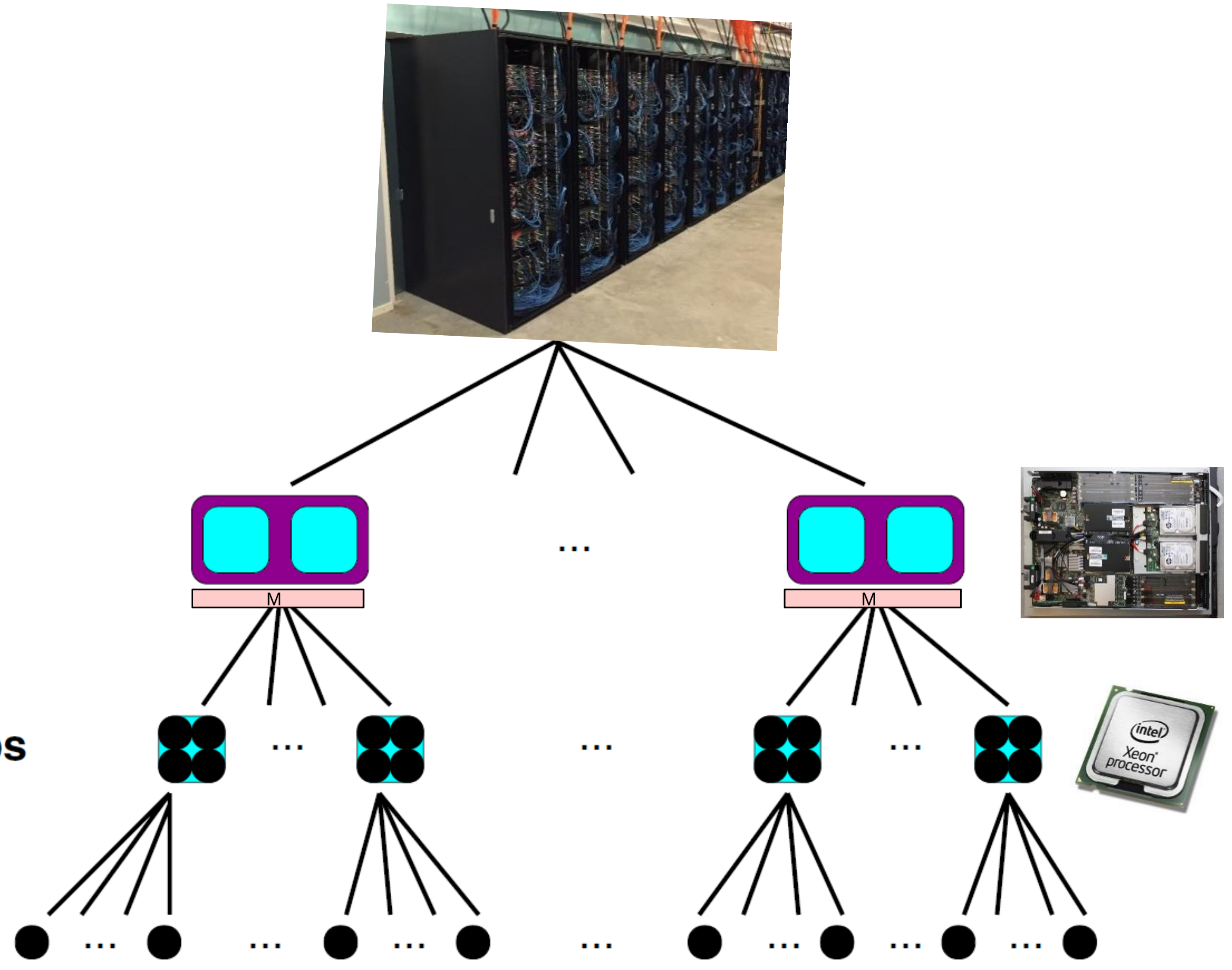
e.g. multiprocessor (SMP) or computer
with a standard multicore CPU

Most common today in HPC and Data centers:

Hybrid Memory System

- Cluster (distributed memory)
  of hundreds, thousands of
  shared-memory servers
  each containing one or several multi-core CPUs

Hybrid (Distributed + Shared) Memory



Interconnection Networks (1)

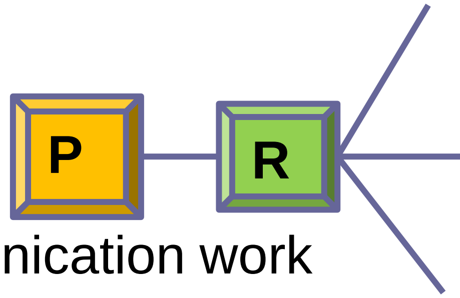
- **Network**
  = physical interconnection medium (wires, switches)
  + communication protocol

  (a) connecting cluster nodes with each other (DMS)

  (b) connecting processors with memory modules (SMS)

Classification

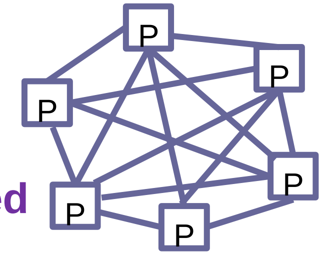
- Direct / static interconnection networks
  - connecting nodes directly to each other
  - Hardware routers (communication coprocessors) can be used to offload processors from most communication work



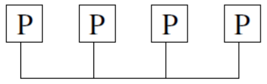
- Switched / dynamic interconnection networks

Interconnection Networks (2): Simple Topologies

fully connected

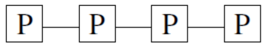


bus

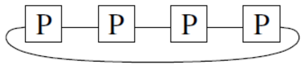


1 wire – bus saturation with many processors
e.g. Ethernet

linear array

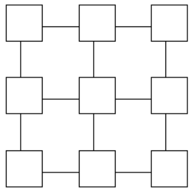


ring

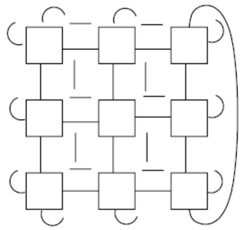


e.g. Token Ring

2D grid

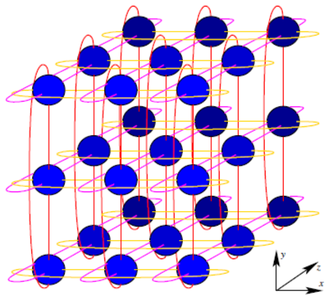


torus:

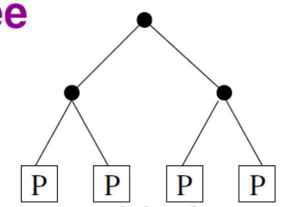


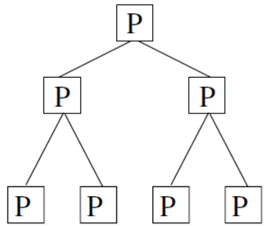
3D grid

3D torus



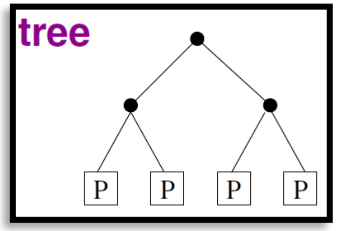
tree



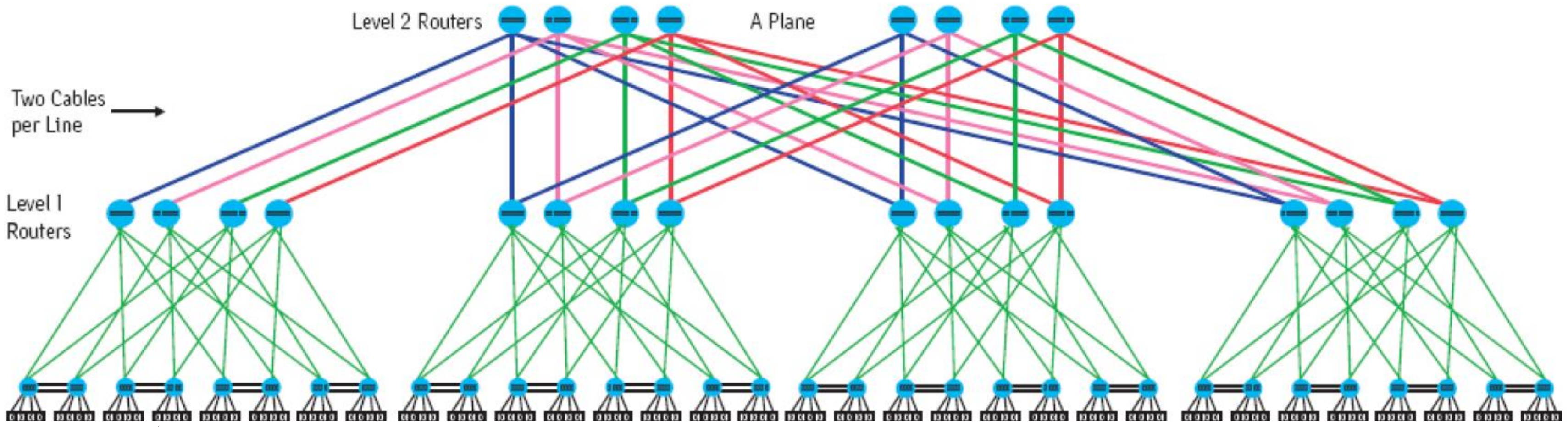


root processor
is bottleneck

Interconnection Networks (3): Fat-Tree Network



- Tree network extended for higher bandwidth (more switches, more links) closer to the root
  - avoids bandwidth bottleneck



- Example: Infiniband network (www.mellanox.com)

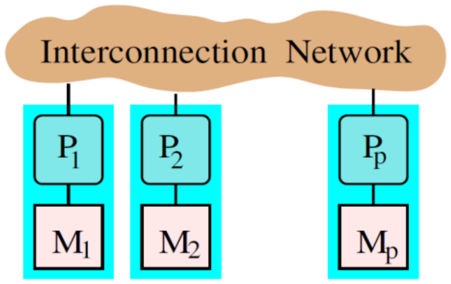
More about Interconnection Networks

- Hypercube, Crossbar, Butterfly, Hybrid networks… → TDDC78

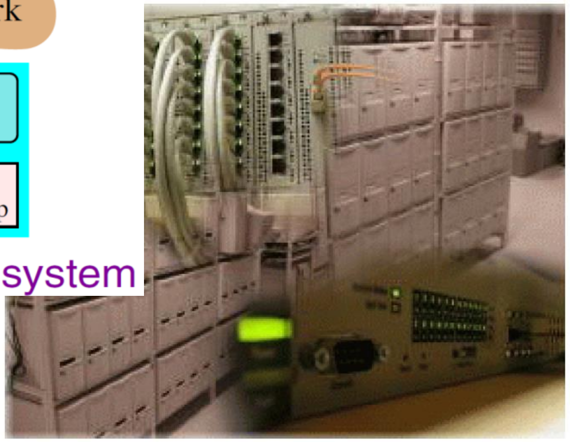
- Switching and routing algorithms

- **Discussion of interconnection network properties**
  - Cost (#switches, #lines)
  - Scalability
    (asymptotically, cost grows not much faster than #nodes)
  - Node degree
  - Longest path (→ latency)
  - Accumulated bandwidth
  - Fault tolerance (worst-case impact of node or switch failure)
  - …

Example: Beowulf-class PC Clusters



Distributed memory system



Characteristics:

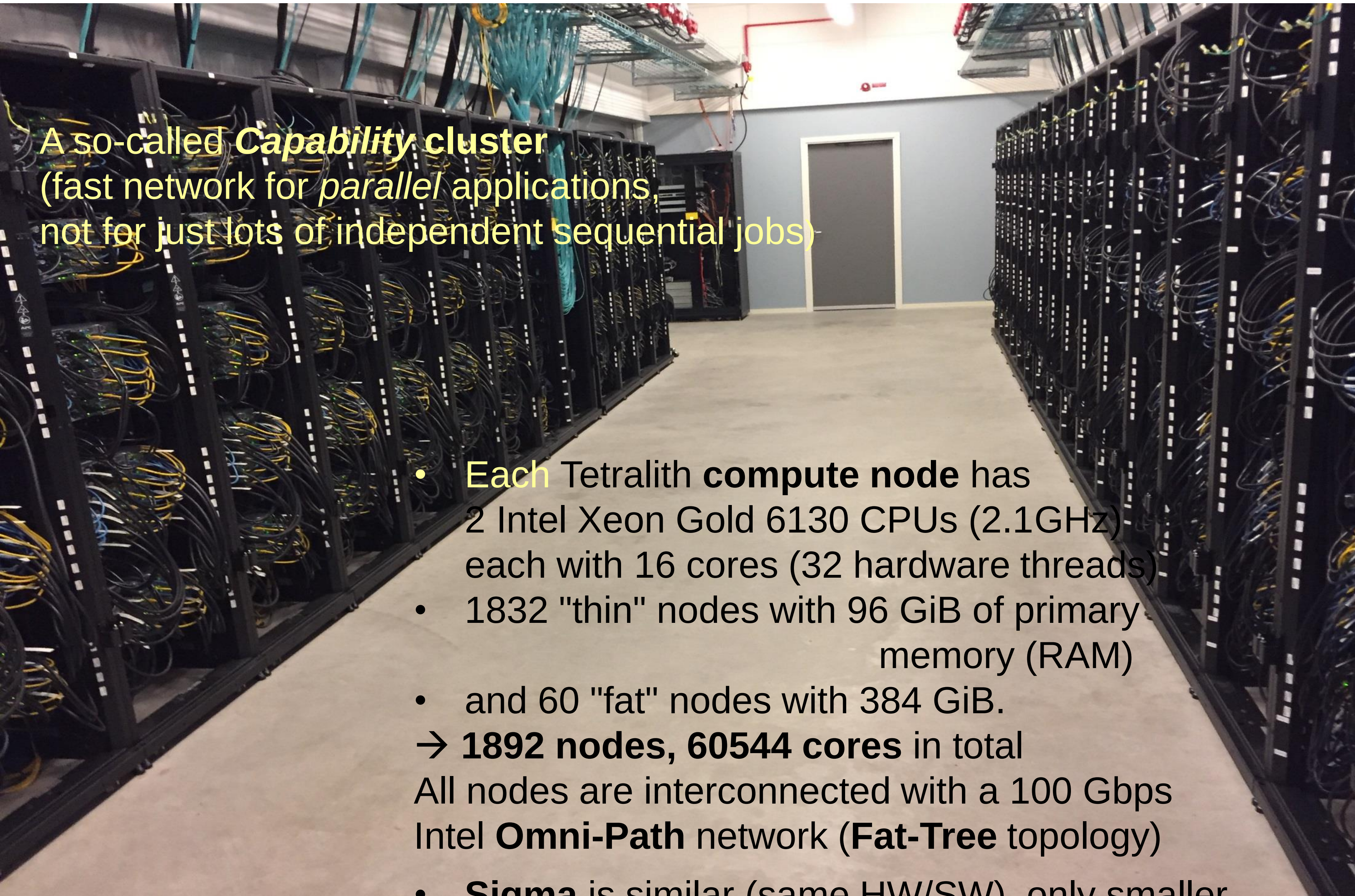
- off-the-shelf (PC) nodes
  with off-the-shelf CPUs
  (Xeon, Opteron, …)
- commodity interconnect
  G-Ethernet, Myrinet, Infiniband, SCI
- Open Source Unix
  Linux, BSD
- Message passing computing
  MPI, PVM

Advantages:

+ best price-performance ratio
+ low entry-level cost
+ vendor independent
+ scalable
+ rapid technology tracking

T. Sterling: The scientific workstation of the future may be a pile of PCs.
Communications of the ACM **39**(9), Sep. 1996

Example: Tetralith (NSC, 2018/2019)



A so-called ***Capability* cluster**
(fast network for *parallel* applications,
not for just lots of independent sequential jobs)

- Each Tetralith **compute node** has 2 Intel Xeon Gold 6130 CPUs (2.1GHz) each with 16 cores (32 hardware threads)
- 1832 "thin" nodes with 96 GiB of primary memory (RAM)
- and 60 "fat" nodes with 384 GiB.

→ **1892 nodes, 60544 cores** in total

All nodes are interconnected with a 100 Gbps Intel **Omni-Path** network (**Fat-Tree** topology)

- **Sigma** is similar (same HW/SW), only smaller

The Challenge

- **Today, basically *all* computers are parallel computers!**
  - Single-thread performance stagnating
  - Dozens of cores and hundreds of HW threads available per server
  - May even be heterogeneous (core types, accelerators)
  - Data locality matters
  - Large clusters for HPC and Data centers, require message passing
- Utilizing more than one CPU core requires thread-level parallelism
- One of the biggest *software* challenges: **Exploiting parallelism**
  - Need LOTS of (mostly, independent) tasks to keep cores/HW threads busy and overlap waiting times (cache misses, I/O accesses)
  - All application areas, not only traditional HPC
    - General-purpose, data mining, graphics, games, embedded, DSP, …
  - Affects HW/SW system architecture, programming languages, algorithms, data structures …
  - Parallel programming is more error-prone (deadlocks, races, further sources of inefficiencies)
    - And thus more expensive and time-consuming

Can’t the compiler fix it for us?

- **Automatic parallelization?**
  - at compile time:
    - Requires static analysis – not effective for pointer-based languages
      - inherently limited – missing runtime information
    - needs programmer hints / rewriting ...
    - ok only for few benign special cases:
      - loop vectorization
      - extraction of instruction-level parallelism
  - at run time (e.g. speculative multithreading)
    - High overheads, not scalable

Insight

- Design of efficient / scalable parallel algorithms is, *in general*, a creative task that is not automatizable
- But some good recipes exist …
  - Parallel algorithmic design patterns $\rightarrow$

The remaining solution …

- **Manual parallelization!**
  - using a parallel programming language / framework,
    - e.g. MPI message passing interface for distributed memory;
    - Pthreads, OpenMP, TBB, … for shared-memory
  - Generally harder, more error-prone than sequential programming,
    - requires special programming expertise to exploit the HW resources effectively
  - Promising approach:
    Domain-specific languages/frameworks,
    - Restricted set of predefined constructs
      doing most of the low-level stuff under the hood
    - e.g. MapReduce, Spark, … for big-data computing

Parallel Programming Model

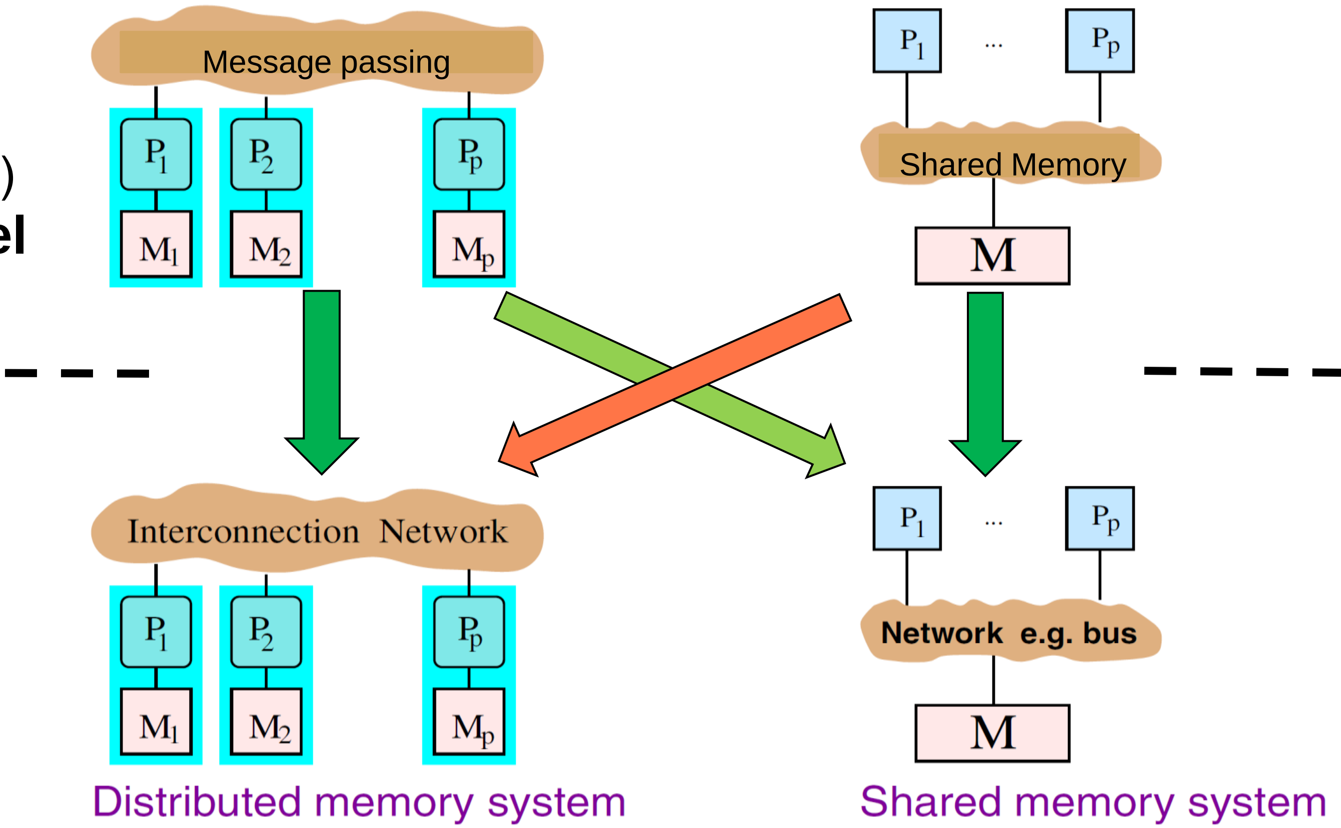
- System-software-enabled **programmer's view** of the underlying hardware
- **Abstracts** from details of the underlying architecture, e.g. network topology
- Focuses on **a few characteristic properties**, e.g. memory model

→ **Portability** of algorithms/programs across a family of parallel architectures

Programmer's view of the underlying system (Lang. constructs, API, …)
→ **Programming model**

Mapping(s) performed by programming toolchain (compiler, runtime system, library, OS, …)

Underlying parallel computer **architecture**



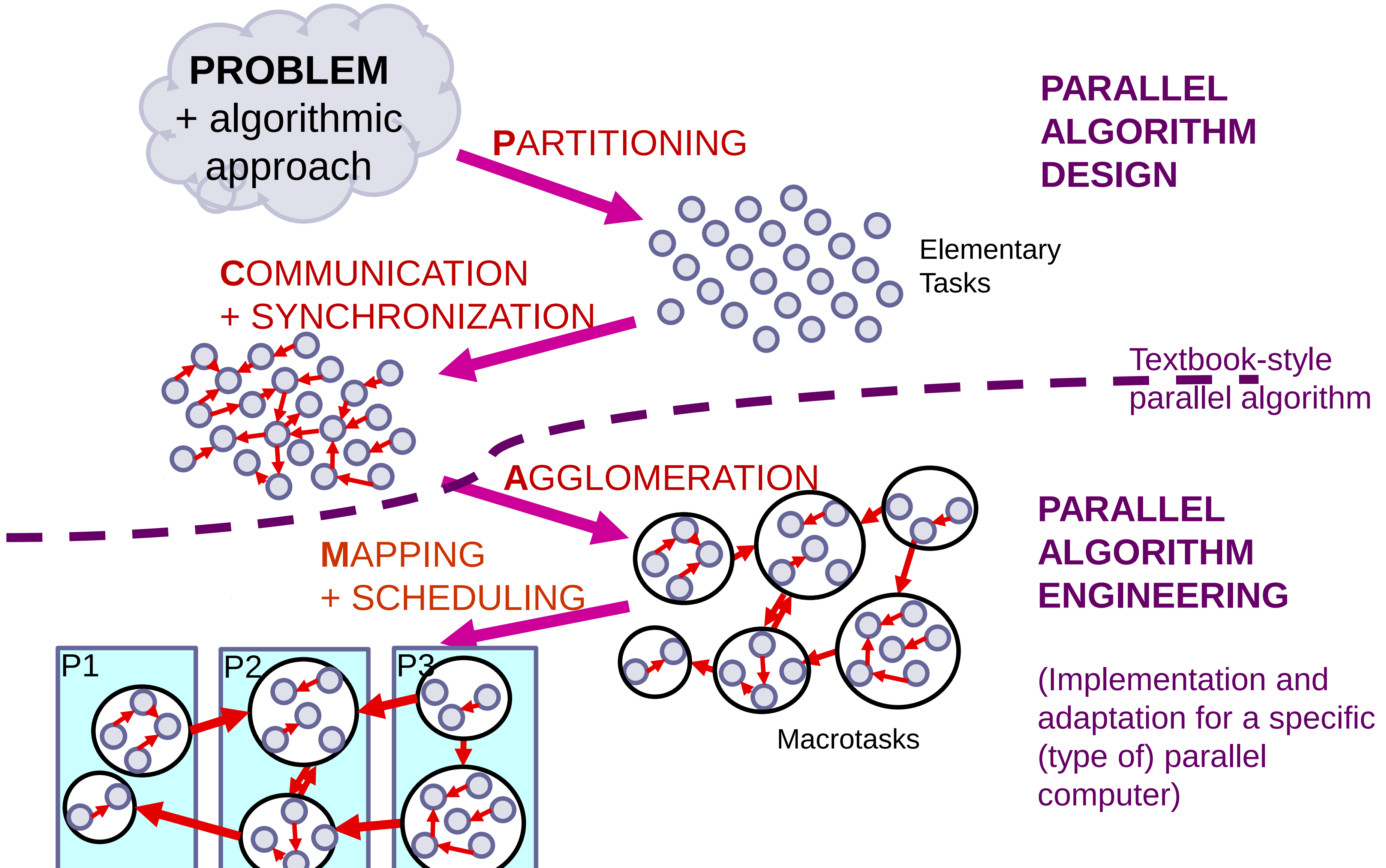
LINKÖPING UNIVERSITY

Design and Analysis of Parallel Algorithms

Introduction

Christoph Kessler, IDA,
Linköpings universitet.

Foster's Generic Method for the Design of Parallel Programs ("PCAM")


PROBLEM
+ algorithmic approach
PARTITIONING
PARALLEL ALGORITHM DESIGN
Elementary Tasks
COMMUNICATION + SYNCHRONIZATION
Textbook-style parallel algorithm
AGGLOMERATION
PARALLEL ALGORITHM ENGINEERING
MAPPING + SCHEDULING
P1
P2
P3
Macrotasks
(Implementation and adaptation for a specific (type of) parallel computer)

Parallel Computation Model
= Programming Model + Cost Model

+ abstract from hardware and technology

+ specify basic operations, when applicable

+ specify how data can be stored

$\rightarrow$ analyze algorithms before implementation independent of a particular parallel computer $\quad \rightarrow \; T = f(n, p, \ldots)$

$\rightarrow$ focus on most characteristic (w.r.t. influence on exec. time) features of a broader class of parallel machines

Programming model

- shared memory / message passing,
- degree of synchronous execution

Cost model

- key parameters
- cost functions for basic operations
- constraints

LINKÖPING UNIVERSITY

Parallel Cost Models

A Quantitative Basis for the Design of Parallel Algorithms

Background reading:
C. Kessler, *Design and Analysis of Parallel Algorithms*, Chapter 2.
Compendium TDDC78/TDDD56, (c) 2019.
https://www.ida.liu.se/~TDDC78/handouts login: parallel
(For internal use in my courses only – please do not share publically)

Cost Model

Cost model: should

+ explain available observations
+ predict future behaviour
+ abstract from unimportant details $\rightarrow$ generalization

Simplifications to reduce model complexity:

- use idealized multicomputer model

  ignore hardware details: memory hierarchies, network topology, ...
- use scale analysis

  drop insignificant effects
- use empirical studies

  calibrate simple models with empirical data

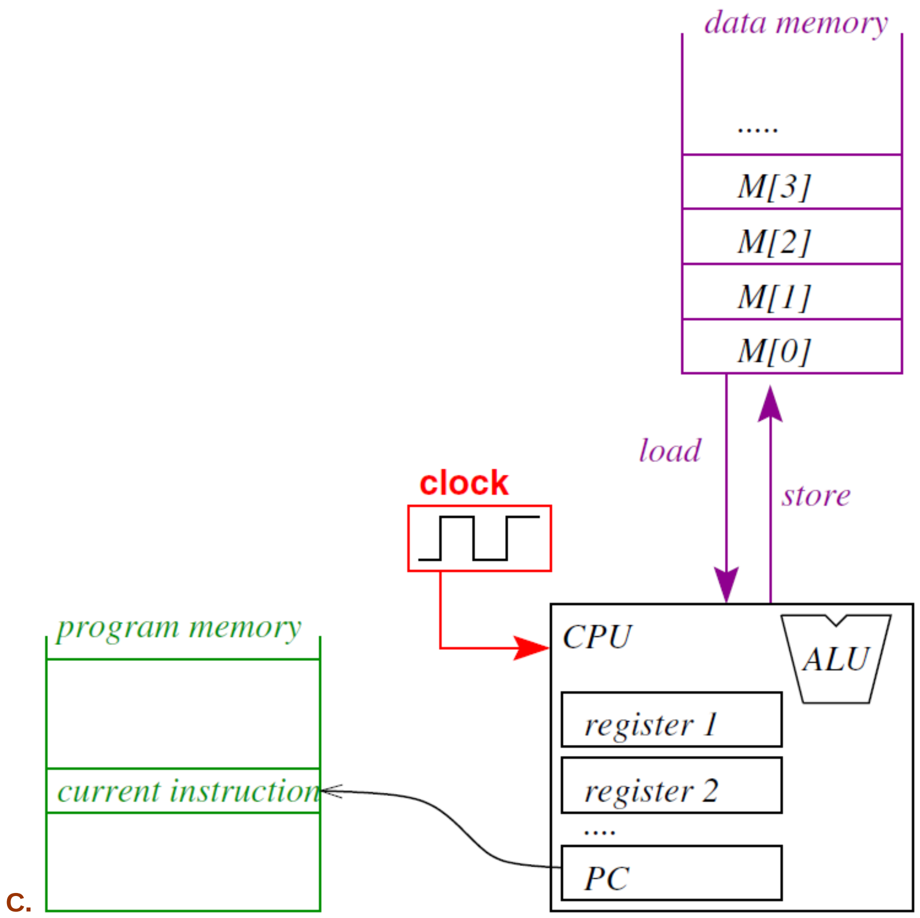
  rather than developing more complex models

How to analyze *sequential* algorithms:

The RAM (von Neumann) model for sequential computing

RAM (Random Access Machine)

programming and cost model for the analysis of sequential algorithms



Basic operations (instructions):
- Arithmetic (add, mul, …) on registers
- Load
- Store
- Branch

Simplifying assumptions for time analysis:
- All of these take 1 time unit
- Serial composition adds time costs
  T(op1;op2) = T(op1)+T(op2)

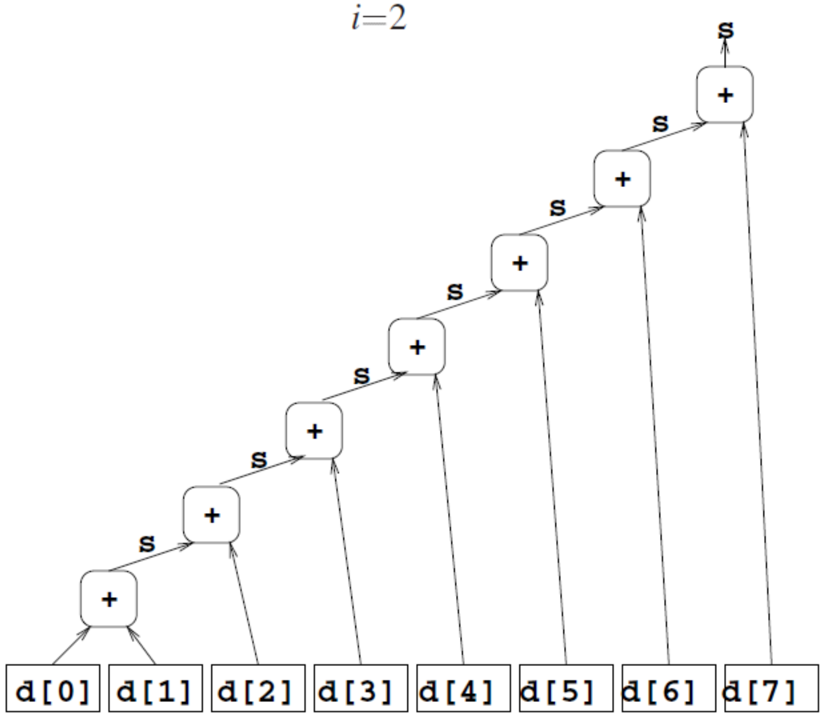
Analysis of sequential algorithms: RAM model (Random Access Machine)

Algorithm analysis: Counting instructions

Example: Computing the global sum of N elements

```
s = d[0]
for (i=1; i<N; i++)
     s = s + d[i]
```

$$t = t_{load} + t_{store} + \sum_{i=2}^{N} (2t_{load} + t_{add} + t_{store} + t_{branch}) = 5N - 3 \in \Theta(N)$$



← *Data flow graph*, showing dependences (precedence constraints) between operations

C. → arithmetic circuit model, directed acyclic graph (DAG) model

The PRAM Model – a Parallel RAM

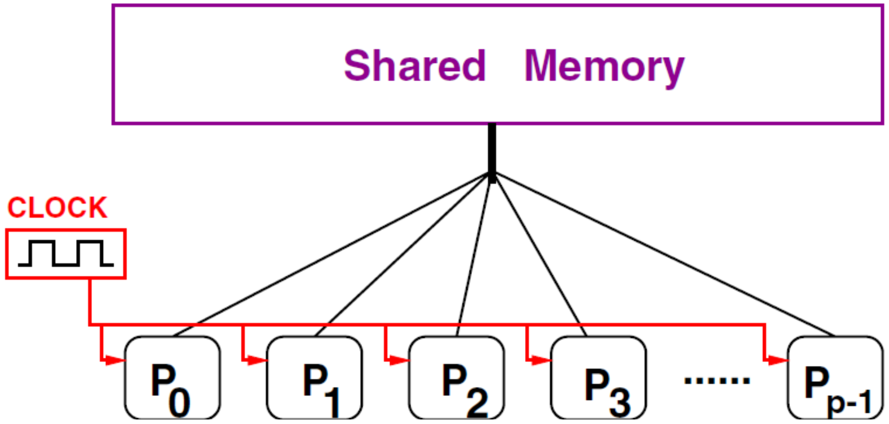
Parallel Random Access Machine [Fortune/Wyllie'78]

p processors

- MIMD
- common clock signal
- arithm./jump: 1 clock cycle

shared memory

- uniform memory access time
- latency: 1 clock cycle (!)
- concurrent memory accesses
- sequential consistency



PRAM variants → TDDD56, TDDC78

Remark

PRAM model is very idealized,
extremely simplifying / abstracting from real parallel architectures:

unbounded number of processors:

abstracts from scheduling overhead

local operations cost 1 unit of time

every processor has unit time memory access
to any shared memory location:

abstracts from communication time, bandwidth limitation,
memory latency, memory hierarchy, and locality

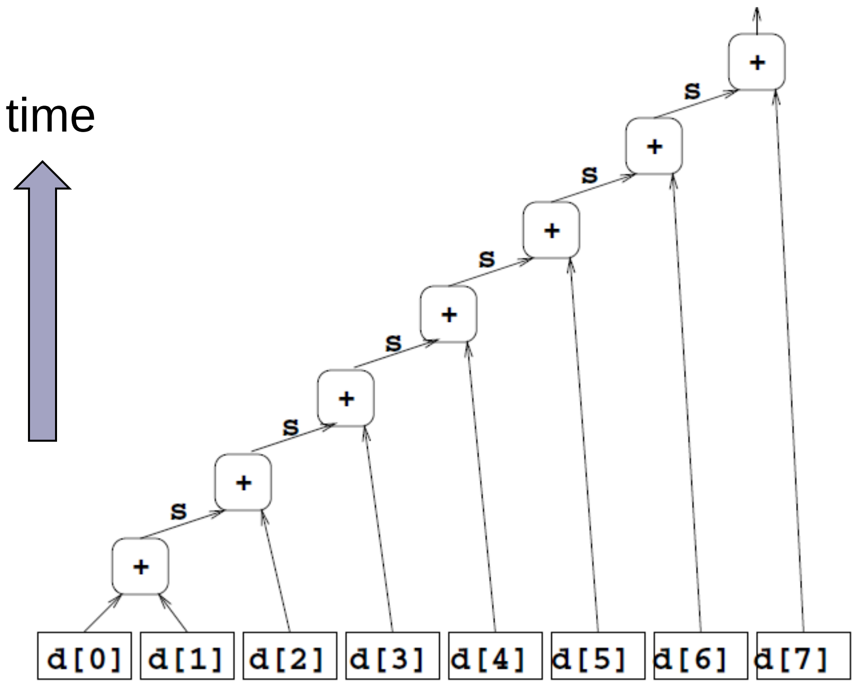
$\rightarrow$ focus on pure, fine-grained parallelism

The PRAM cost model has only 1 machine-specific parameter:
the number of processors

→ Good for **early analysis** of parallel algorithm designs:
A parallel algorithm that does not scale under the PRAM model
does not scale well anywhere else!

A first parallel sum algorithm …

Keep the sequential sum algorithm's structure / data flow graph.
Giving each processor one task (load, add) does not help much
- All *n* loads could be done in parallel, but
- Processor *i* needs to wait for partial result from processor *i*-1, for *i*=1,…,*n*-1



← *Data flow graph*,
showing dependences
(precedence constraints)
between operations

→ Still O(*n*) time steps!

Divide&Conquer Parallel Sum Algorithm in the PRAM / Circuit (DAG) cost model

Given n numbers $x_0, x_1, \ldots, x_{n-1}$ stored in an array.

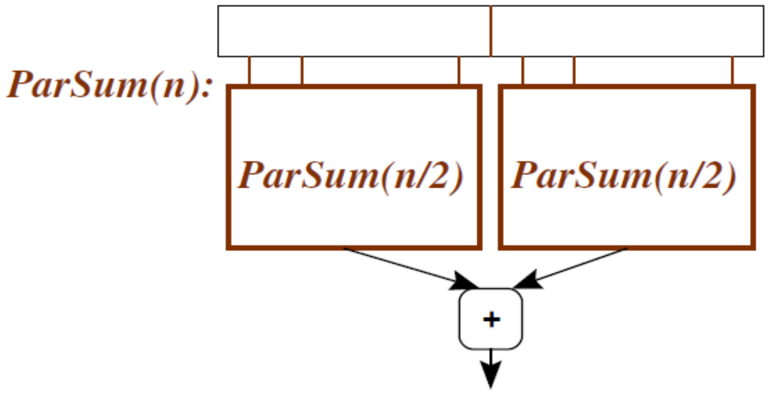
The global sum $\sum_{i=0}^{n-1} x_i$ can be computed in $\lceil \log_2 n \rceil$ time steps on an EREW PRAM with n processors.

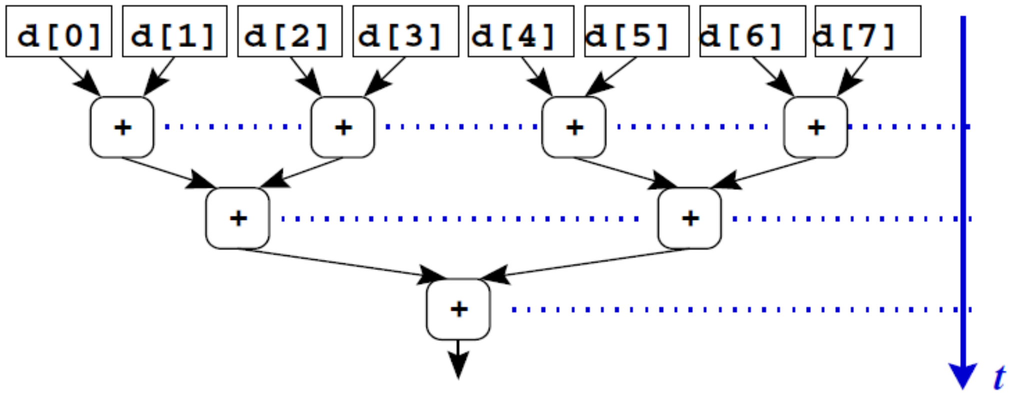
Divide&Conquer Parallel Sum Algorithm in the PRAM / Circuit (DAG) cost model

Given n numbers $x_0, x_1, ..., x_{n-1}$ stored in an array.

The global sum $\sum_{i=0}^{n-1} x_i$ can be computed in $\lceil \log_2 n \rceil$ time steps on an EREW PRAM with n processors.

Parallel algorithmic paradigm used: Parallel Divide-and-Conquer





Divide phase: trivial, time $O(1)$
Recursive calls: parallel time $T(n/2)$
with base case: load operation, time $O(1)$
Combine phase: addition, time $O(1)$

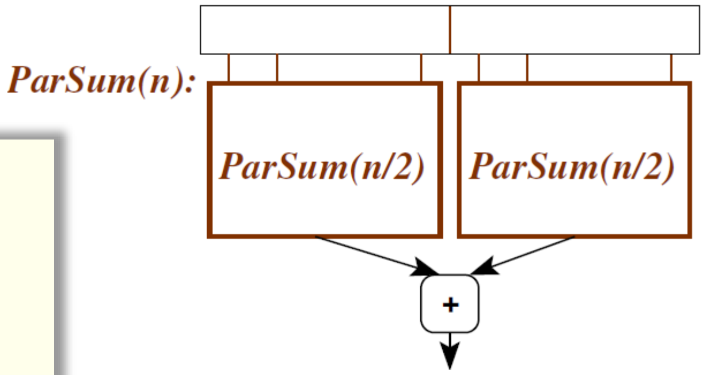
Recurrence equation for parallel execution time:

$$\begin{cases} T(n) = T(n/2) + O(1) \\ T(1) = O(1) \end{cases}$$

Use induction or the master theorem [Cormen+'90 Ch.4] $\rightarrow T(n) \in O(\log n)$

Recursive formulation of DC parallel sum algorithm in some programming model

Implementation e.g. in **Cilk**: (shared memory)



```
cilk int parsum ( int *d, int from, int to )
{
    int mid, sumleft, sumright;
    if (from == to)  return d[from];     // base case
    else {
        mid = (from + to) / 2;
        sumleft = spawn parsum ( d, from, mid );
        sumright = parsum( d, mid+1, to );
        sync;
        return sumleft + sumright;
    }
}
```

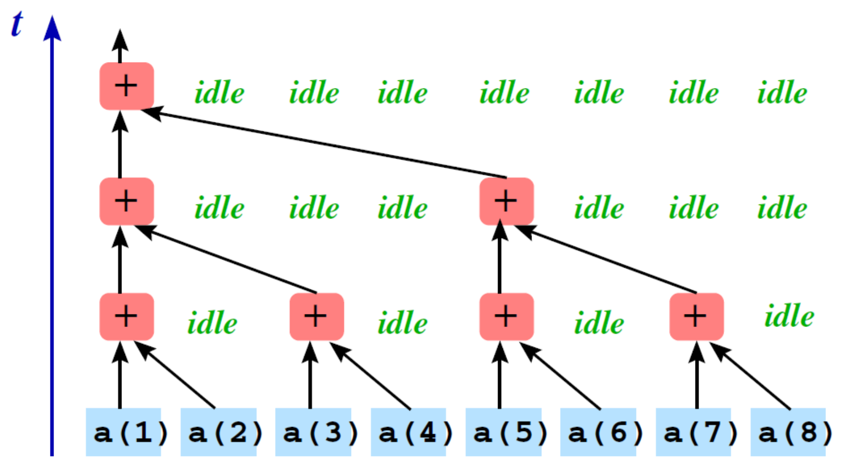
Fork-Join execution style:
single task starts,
tasks spawn child tasks for independent subtasks, and synchronize with them

```
// The main program:

main()
{
  …
  parsum ( data, 0, n-1 );
  …
}
```

Circuit / DAG model

- Independent of how the parallel computation is expressed, the resulting (unfolded) task graph looks the same.



- **Task graph** is a directed acyclic graph (DAG) G=(*V*,*E*)
  - Set *V* of vertices: elementary tasks (taking time 1 resp. *O*(1) each)
  - Set *E* of directed edges: dependences (partial order on tasks) (v_1,v_2) in $E \rightarrow v_1$ must be finished before v_2 can start
- **Critical path** = longest path from an entry to an exit node
  - Length of critical path is a lower bound for parallel time complexity
- **Parallel time** can be longer if number of processors is limited
  → *schedule tasks* to processors such that dependences are preserved
  (by programmer (SPMD execution) or run-time system (fork-join exec.))

For a fixed number of processors … ?

- Usually, *p* << *n*
- Requires scheduling the work to *p* processors

(A) manually, at algorithm design time:

- Requires **algorithm engineering**
- E.g. stop the parallel divide-and-conquer e.g. at subproblem size *n/p* and switch to sequential divide-and-conquer (= task agglomeration)

  For parallel sum:

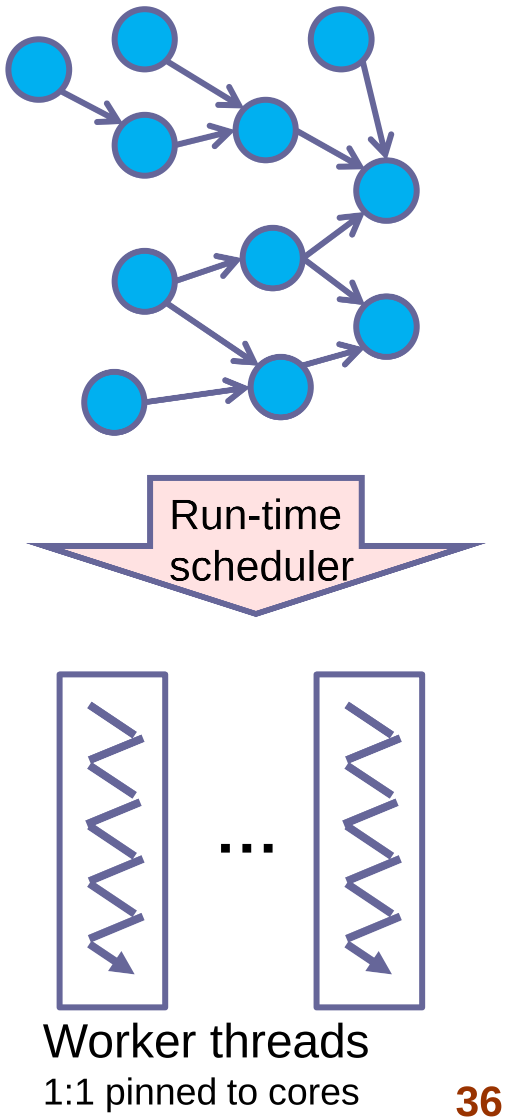
  - Step 0. Partition the array of *n* elements in *p* slices of *n/p* elements each (= domain decomposition)
  - Step 1. Each processor calculates a local sum for one slice, using the sequential sum algorithm,
    resulting in *p* partial sums (intermediate values)
  - Step 2. The *p* processors run the parallel algorithm
    to sum up the intermediate values to the global sum.

For a fixed number of processors … ?

- Usually, *p* << *n*
- Requires scheduling the work to *p* processors

(B) automatically, at run time:

- Requires a **task-based runtime system** with dynamic scheduler
  - Each newly created task is dispatched at runtime to an available worker processor.
  - Load balancing (→ runtime overhead)
    - Central task queue where idle workers fetch next task to execute
    - Local task queues + Work stealing – idle workers steal a task from some other processor



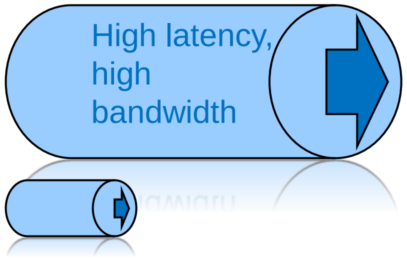
Analysis of Parallel Algorithms

Christoph Kessler, IDA,
Linköpings universitet.

Analysis of Parallel Algorithms

Performance metrics of parallel programs

- **Parallel execution time**
  - Counted from the start time of the earliest task to the finishing time of the latest task
- **Work** – the total number of performed elementary operations
- **Cost** – the product of parallel execution time and #processors
- **Speed-up**
  - the factor by how much faster we can solve a problem with *p* processors than with 1 processor, usually in range (0…*p*)
- **Parallel efficiency** = Speed-up / #processors, usually in (0…1)
- **Throughput** = #operations finished per second
- **Scalability**
  - does speedup keep growing well also when #processors grows large?



Analysis of Parallel Algorithms

Asymptotic Analysis

- Estimation based on a cost model and algorithm idea (pseudocode operations)
- Discuss behavior for large problem sizes, large #processors

Empirical Analysis

- Implement in a concrete parallel programming language
- Measure time on a concrete parallel computer
  - Vary number of processors used, as far as possible
- More precise
- More work, and fixing bad designs at this stage is expensive

Parallel Time, Work, Cost

problem size n

processors p

time $t(p,n)$

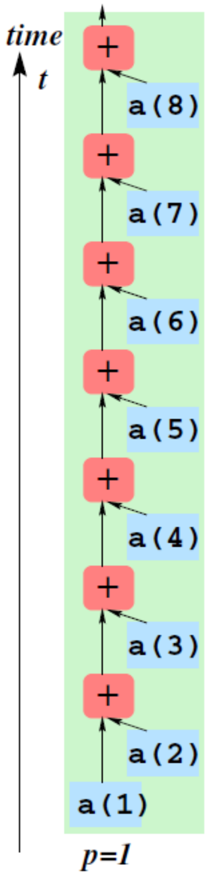
work $w(p,n)$

cost $c(p,n) = t \cdot p$

Example:
seq. sum algorithm

```
s = d[0]
for (i=1; i<N;i++)
      s = s + d[i]
```

$n-1$ additions
n loads
$O(n)$ other

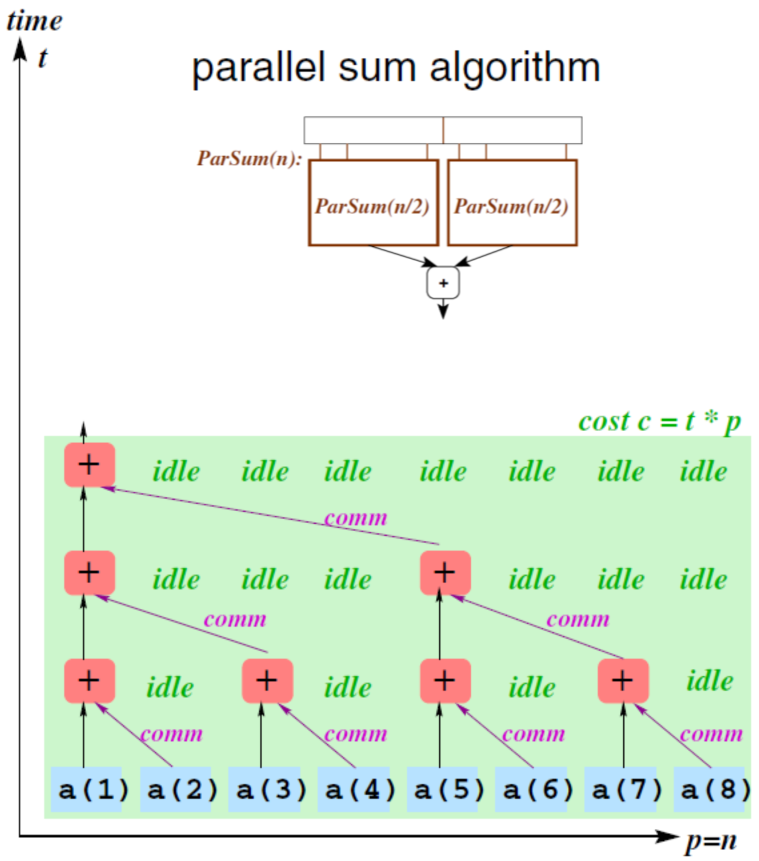


$$t(1,n) = t_{seq}(n) = O(n)$$

$$w(1,n) = O(n)$$

$$c(1,n) = t(1,n) \cdot 1$$

$$= O(n)$$



$$t(n,n) = O(\log n)$$

$$w(n,n) = O(n)$$

$$c(n,n) = O(n \log n)$$

par. sum alg. *not* cost-effective!

Speedup

Consider problem $\mathcal{P}$, parallel algorithm A for $\mathcal{P}$

T_s = time to execute the best serial algorithm for $\mathcal{P}$
on one processor of the parallel machine

$T(1)$ = time to execute parallel algorithm A on 1 processor

$T(p)$ = time to execute parallel algorithm A on p processors

Absolute speedup $S_{abs} = \frac{T_s}{T(p)}$

Relative speedup $S_{rel} = \frac{T(1)}{T(p)}$

$S_{abs} \leq S_{rel}$

Speedup $S(p)$ with p processors is usually in the range $(0 \ldots p)$

Amdahl's Law: Upper bound on Speedup

Consider execution (trace) of parallel algorithm A:

sequential part A^s where only 1 processor is active

parallel part A^p that can be sped up perfectly by p processors

$\rightarrow$ total work $w_A(n) = w_{A^s}(n) + w_{A^p}(n)$, time $T = T_{A^s} + \frac{T_{A^p}}{p}$,

Amdahl's Law

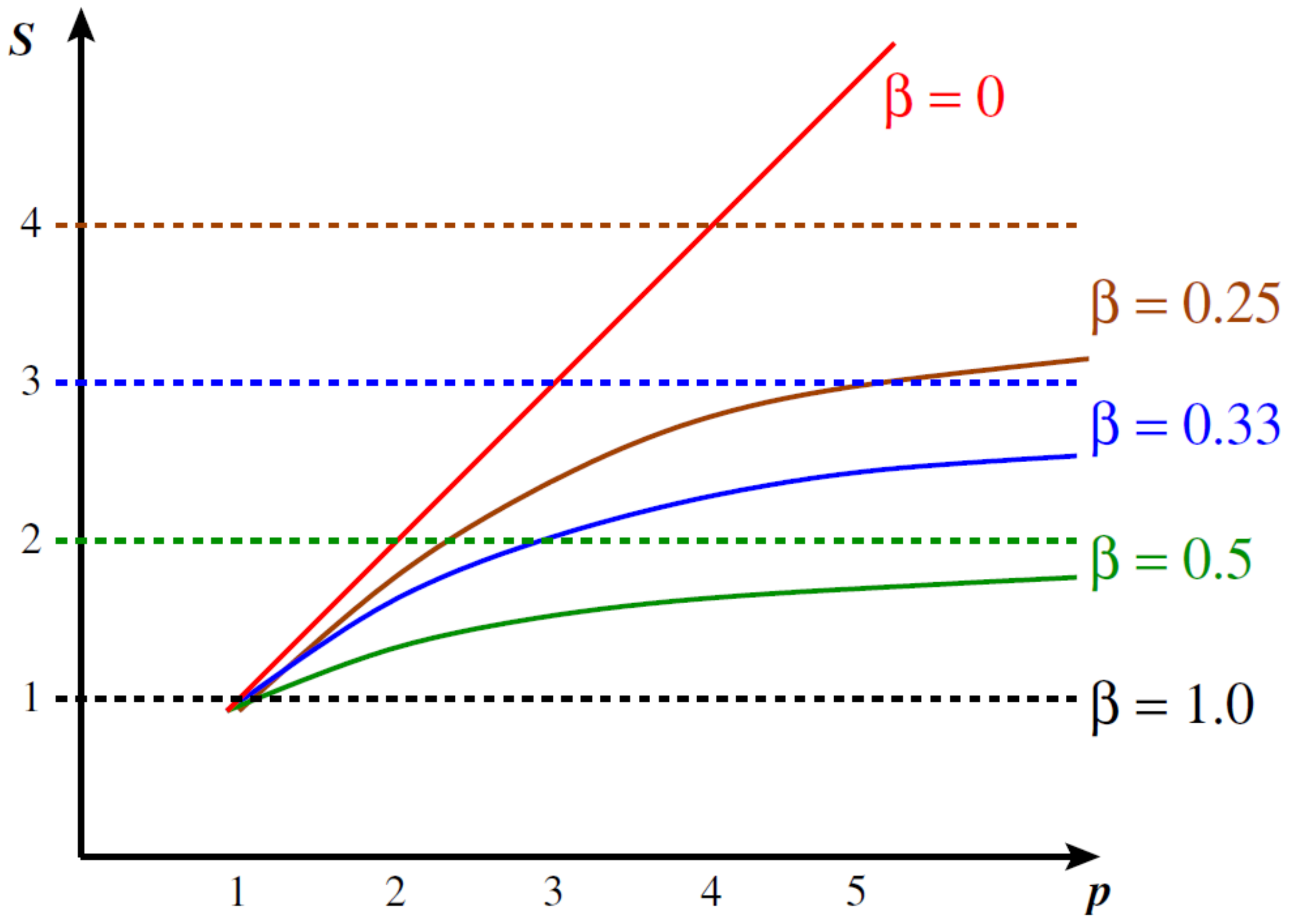
If the sequential part of A is a *fixed* fraction of the total work irrespective of the problem size n, that is, if there is a constant β with

$$\beta = \frac{w_{A^s}(n)}{w_A(n)} \leq 1$$

the relative speedup of A with p processors is limited by

$$\frac{p}{\beta p + (1-\beta)} < 1/\beta$$

Amdahl's Law



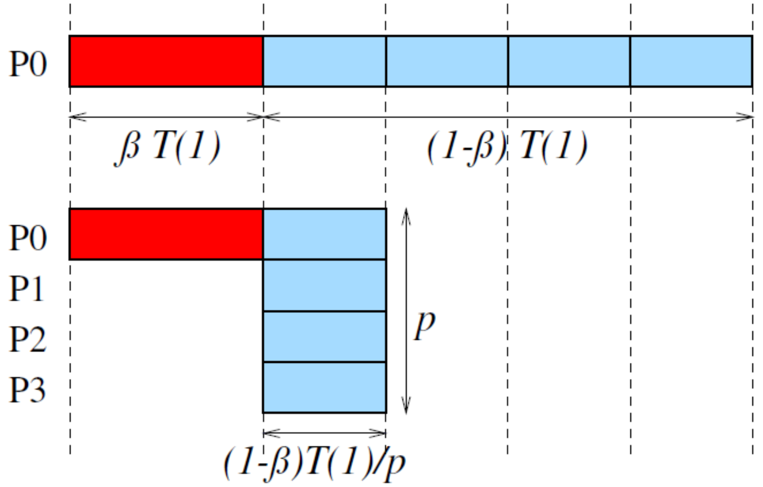
$$S(p) = \frac{p}{\beta p + (1-\beta)} < 1/\beta$$

Proof of Amdahl's Law

$$S_{rel} = \frac{T(1)}{T(p)} = \frac{T(1)}{T_{A^s} + T_{A^p}(p)}$$

Assume perfect parallelizability of the parallel part A^p,
that is, $T_{A^p}(p) = (1-\beta)T(p) = (1-\beta)T(1)/p$:

$$S_{rel} = \frac{T(1)}{\beta T(1) + (1-\beta)T(1)/p} = \frac{p}{\beta p + 1 - \beta} \leq 1/\beta$$



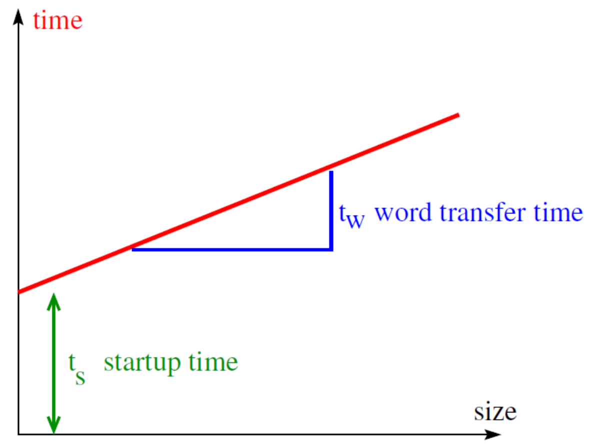
Towards More Realistic Cost Models

Modeling the cost of communication and data access

Christoph Kessler, IDA,
Linköpings universitet.

Modeling Communication Cost: Delay Model

Idealized multicomputer: point-to-point communication costs overhead t_{msg}.



Cost of communicating a larger block of n bytes:

time $t_{msg}(n) =$ sender overhead + latency + receiver overhead + n/bandwidth

$$=: \quad t_{startup} + n \cdot t_{transfer}$$

Assumption: network not overloaded; no conflicts occur at routing

$t_{startup}$ = startup time (time to send a 0-byte message)

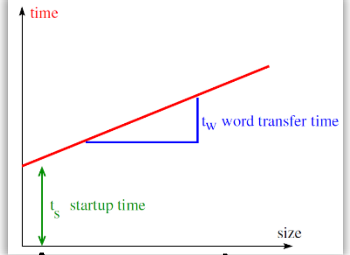
accounts for hardware and software overhead.

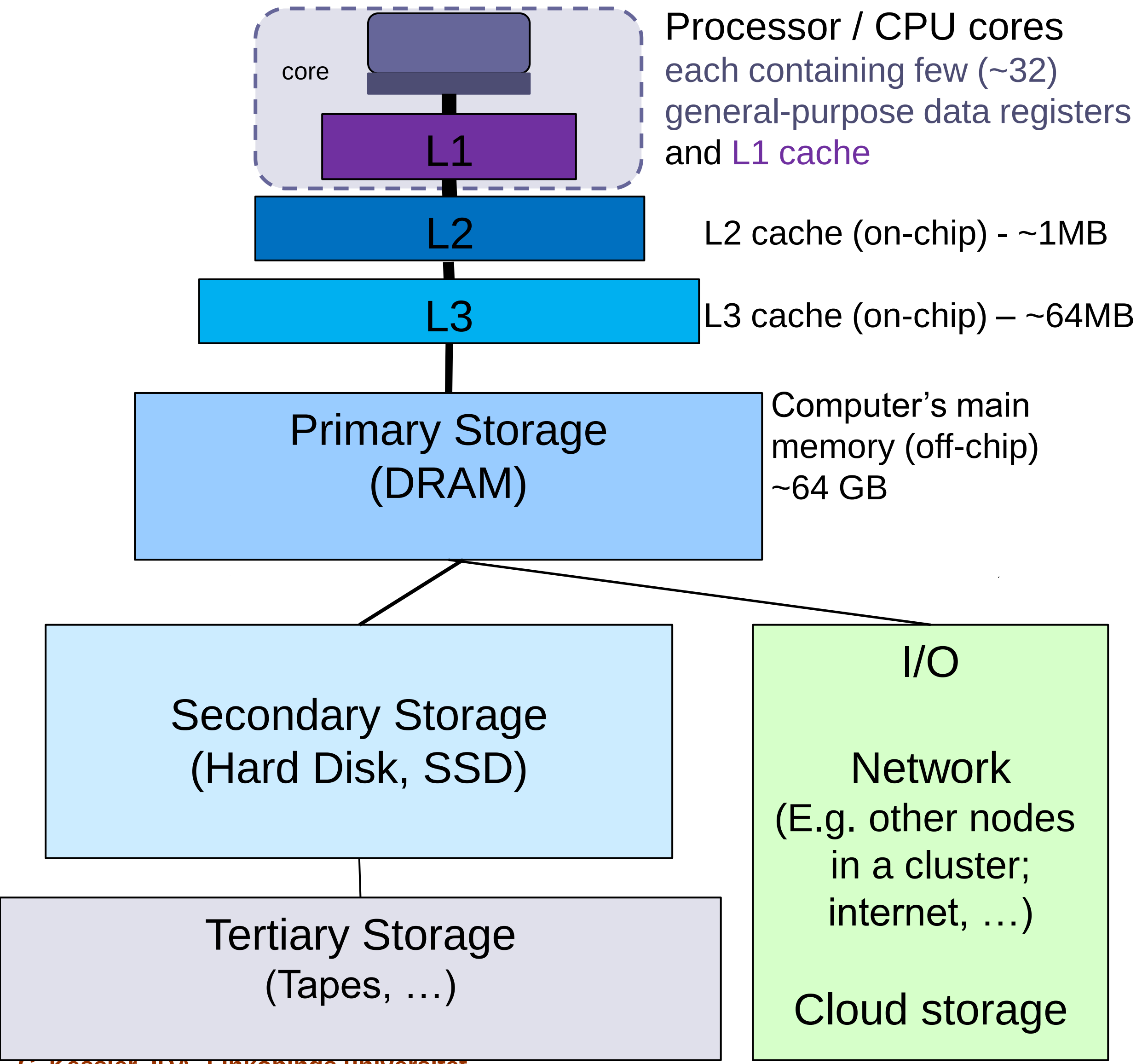
$t_{transfer}$ = transfer rate, send time per word sent.

depends on the network bandwidth.

Memory Hierarchy

And The Real Cost of Data Access



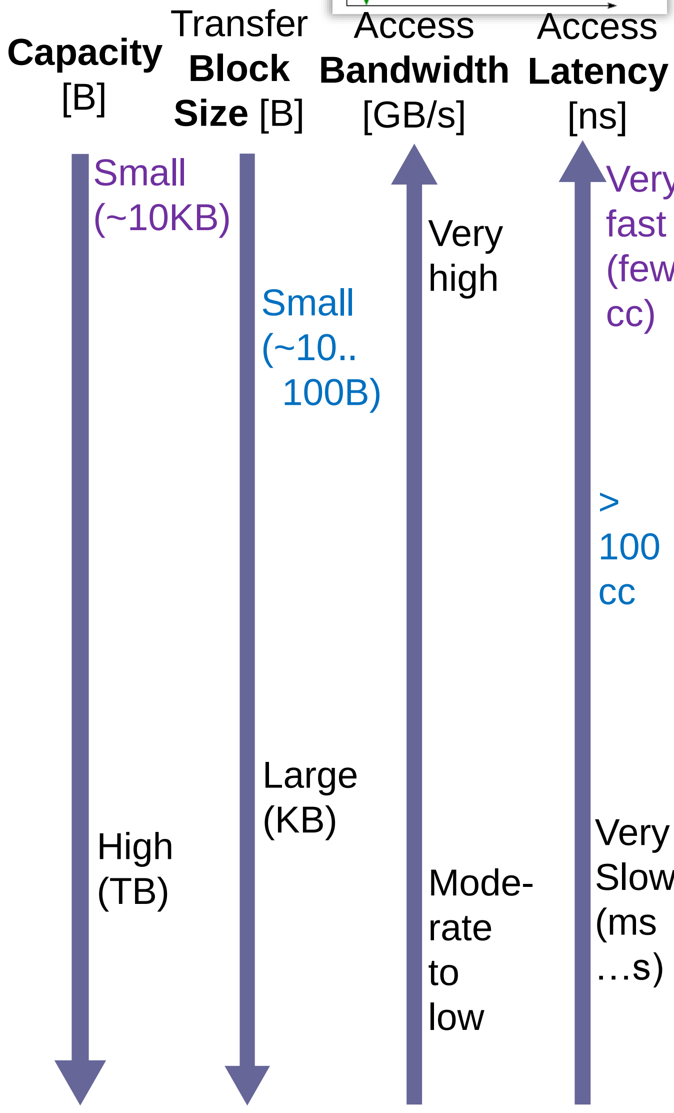

core
L1
L2
L3
Primary Storage (DRAM)
Secondary Storage (Hard Disk, SSD)
Tertiary Storage (Tapes, …)
I/O
Network (E.g. other nodes in a cluster; internet, …)
Cloud storage

Processor / CPU cores each containing few (~32) general-purpose data registers and L1 cache

L2 cache (on-chip) - ~1MB

L3 cache (on-chip) – ~64MB

Computer's main memory (off-chip) ~64 GB



Data Locality

- **Memory hierarchy rationale**: Try to amortize the high access cost of lower levels (DRAM, disk, …) by caching data in higher levels for faster subsequent accesses
  - **Cache miss** – stall the computation. fetch the block of data containing the accessed address from next lower level, then resume
  - More reuse of cached data (**cache hits**) → better performance
- **Working set** = the set of memory addresses accessed together in a period of computation
- **Data locality** = property of a computation: keeping the working set small during a computation
  - **Temporal locality** – re-access same data element multiple times within a short time interval
  - **Spatial locality** – re-access neighbored memory addresses multiple times within a short time interval
- High latency favors larger transfer block sizes (cache lines, memory pages, file blocks, messages) for amortization over many subsequent accesses

Memory-bound vs. CPU-bound computation

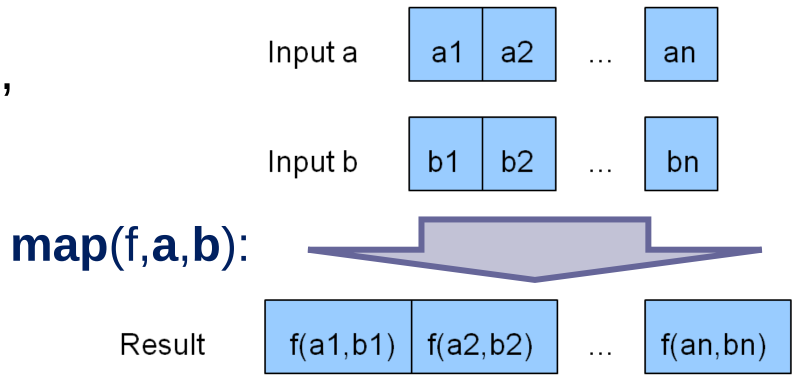
- **Arithmetic intensity** of a computation
  = #arithmetic instructions (computational work) executed
  per accessed element of data in memory (after cache miss)
- A computation is **CPU-bound**
  if its arithmetic intensity is >> 1.
  - The performance bottleneck is the CPU's arithmetic throughput
- A computation is **memory-access bound** otherwise.
  - The performance bottleneck is memory accesses,
    CPU is not fully utilized
- Examples:
  - Matrix-matrix-multiply (if properly implemented) is CPU-bound.
  - Array global sum is memory-bound on most architectures.

Some Parallel Algorithmic Design Patterns

Christoph Kessler, IDA,
Linköpings universitet.

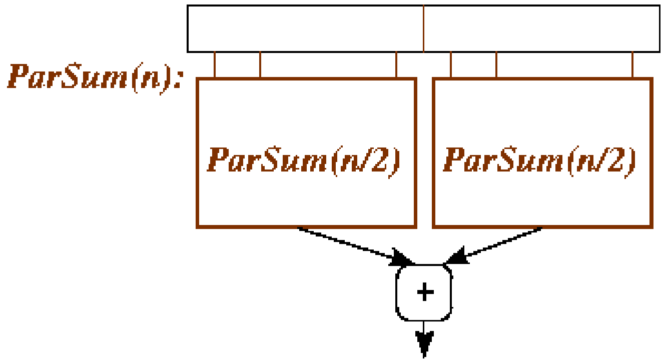
Data Parallelism

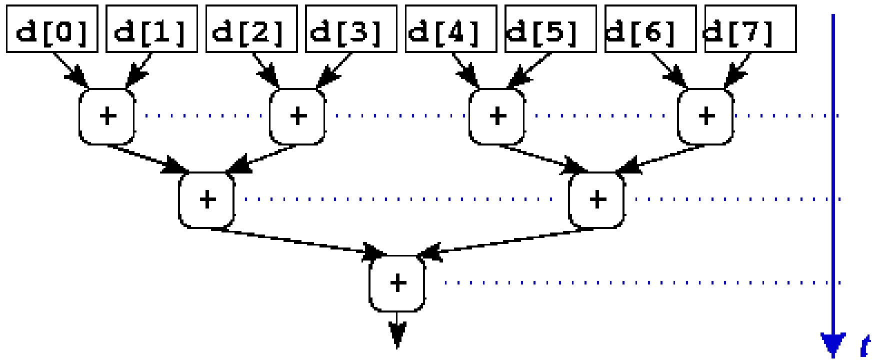
- **Given:**
  - One (or several) data containers $\boldsymbol{x}$, $\boldsymbol{z}$, … *with* n elements each, e.g. array(s) $\boldsymbol{x} = (x_1,...x_n)$, $\boldsymbol{z} = (z_1,\ldots,z_n)$, …
  - An operation *f* on individual elements of *x, z, …* (e.g. *incr, sqrt, mult*, ...)
- **Compute**: $\boldsymbol{y} = f(\boldsymbol{x}) = (f(x_1), ..., f(x_n))$
- **Parallelizability: Each data element defines a task**
  - Fine grained parallelism
  - Easily partitioned into independent tasks, fits very well on all parallel architectures
- **Notation** with higher-order function:
  - $y = \boldsymbol{map}\ (f,\ x)$



Data-parallel Reduction

- **Given:**
  - A data container ***x*** with *n* elements,
    e.g. array ***x*** = $(x_1,...x_n)$
  - A <u>binary, associative</u> operation *op* on individual elements of *x*
    (e.g. *add, max, bitwise-or*, ...)
- **Compute**: $y \;=\; OP_{i=1\ldots n}\,\mathbf{x} \;=\; x_1 \;op\; x_2 \;op\; \ldots \;op\; x_n$
- **Parallelizability:** Exploit *associativity* of op





- **Notation** with higher-order function*:*
  - *y* = ***reduce*** (*op*, ***x***)

MapReduce (pattern)

- A **Map** operation with operation *f*
  on one or several input data containers **x**, …,
  producing a temporary output data container **w**,
  directly followed by a **Reduce** with operation *g* on **w**
  producing result *y*
- y = **MapReduce** (*f*, *g*, **x**, …)

- Example:

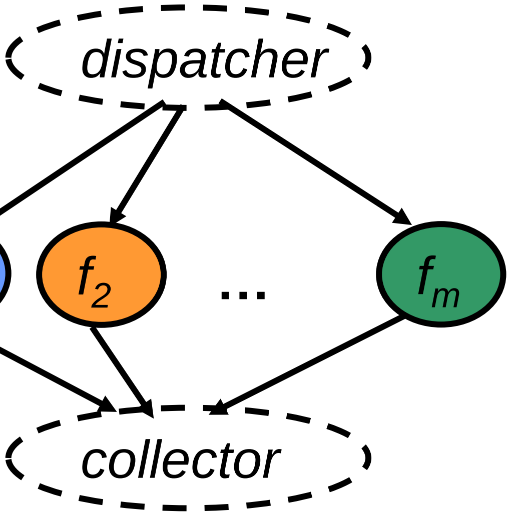
  Dot product of two vectors ***x***, ***z***: $y = \Sigma_i \; x_i * z_i$

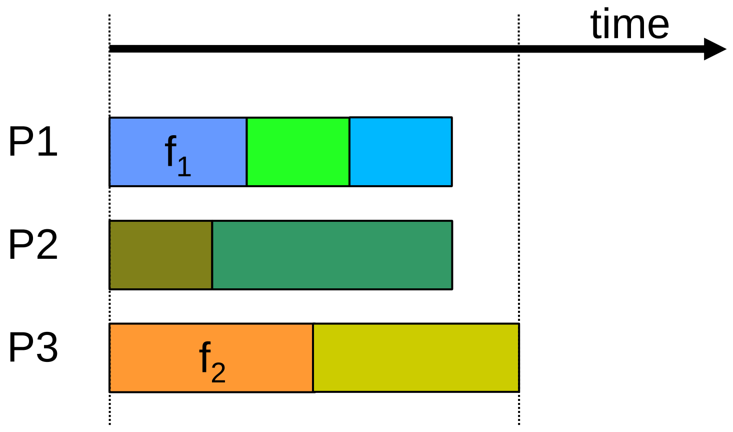
  f = scalar multiplication,

  g = scalar addition

Task Farming

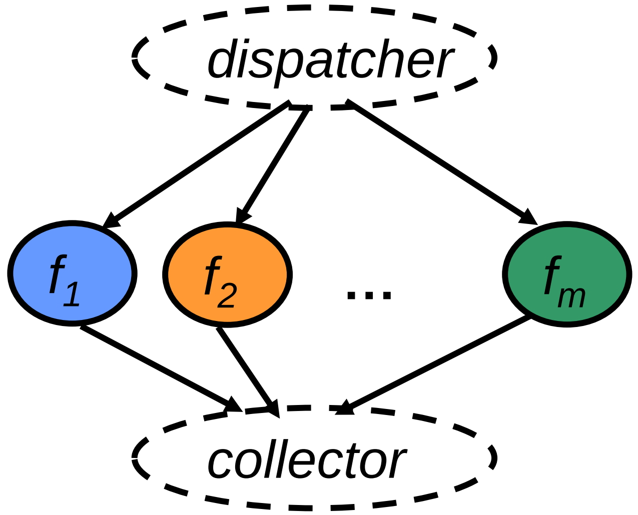
- **Independent subcomputations** $f_1, f_2, \ldots, f_m$ could be done in parallel and/or in arbitrary order, e.g.
  - independent loop iterations
  - independent function calls
- **Scheduling (mapping) problem**
  - *m* tasks onto *p* processors
  - static (before running) or dynamic
  - *Load balancing* is important: most loaded processor determines the parallel execution time
- **Notation** with higher-order function:
  - ***farm*** $(f_1, \ldots, f_m)\ (x_1, \ldots, x_n)$



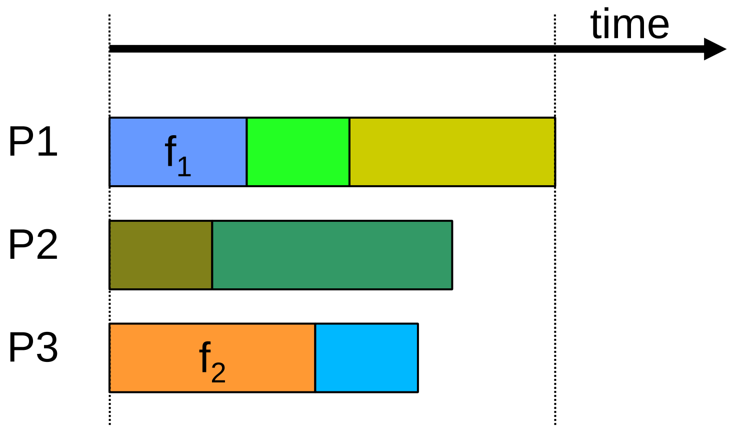


Task Farming

- **Independent subcomputations** $f_1, f_2, \ldots, f_m$ could be done in parallel and/or in arbitrary order, e.g.
  - independent loop iterations
  - independent function calls



- **Scheduling (mapping) problem**
  - *m* tasks onto *p* processors
  - static (before running) or dynamic
  - *Load balancing* is important: most loaded processor determines the parallel execution time



- **Notation** with higher-order function:
  - ***farm*** $(f_1, \ldots, f_m)$ $(x_1,\ldots,x_n)$

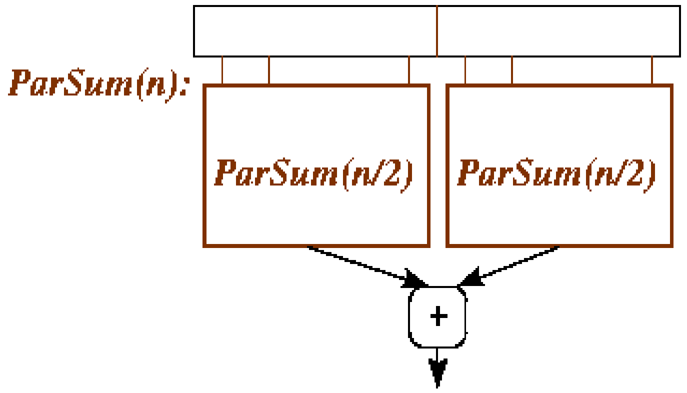
Parallel Divide-and-Conquer

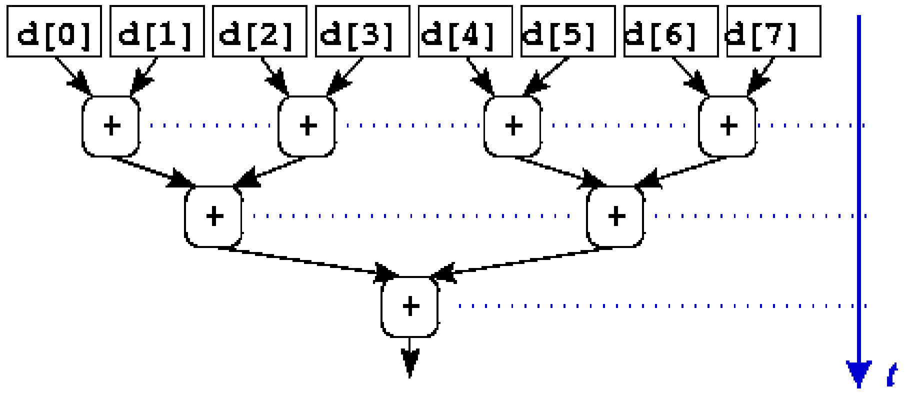
- **(Sequential) Divide-and-conquer:**
  - If given problem instance P is *trivial*, solve it *directly*. Otherwise:
  - *Divide*: Decompose problem instance P in one or several smaller independent instances of the same problem, $P_1, \ldots, P_k$
  - For each i: solve P_i by recursion.
  - *Combine* the solutions of the P_i into an overall solution for P

- **Parallel Divide-and-Conquer:**
  - Recursive calls can be done in parallel.
  - Parallelize, if possible, also the divide and combine phase.
  - Switch to sequential divide-and-conquer when enough parallel tasks have been created.

- **Notation** with higher-order function:
  - *solution* = ***DC*** (*divide, combine, istrivial, solvedirectly, n, P*)

Example: Parallel Divide-and-Conquer





Example: **Parallel Sum** over integer-array *x*

Exploit associativity:

$$Sum(x_1,...,x_n) \; = \; Sum(x_1,...x_{n/2}) + Sum(x_{n/2+1},...,x_n)$$

Divide: trivial, split array *x* in place

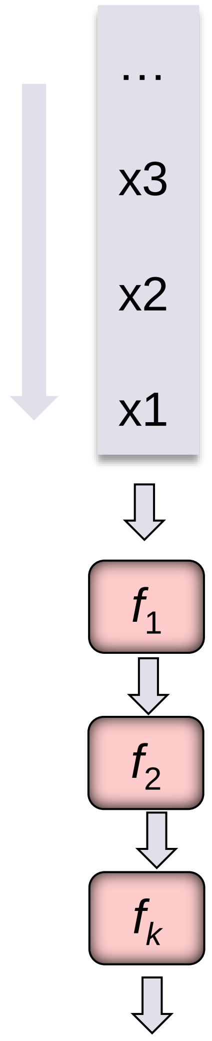
Combine is just an addition.

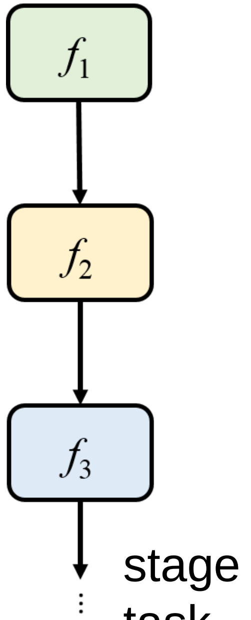
$$y \; = \; \mathbf{DC} \; (\; split, \;\; add, \;\; nIsSmall, \;\; addFewInSeq, \;\; n, \;\; x \;\;)$$

→ Data parallel reductions are an important special case of DC.

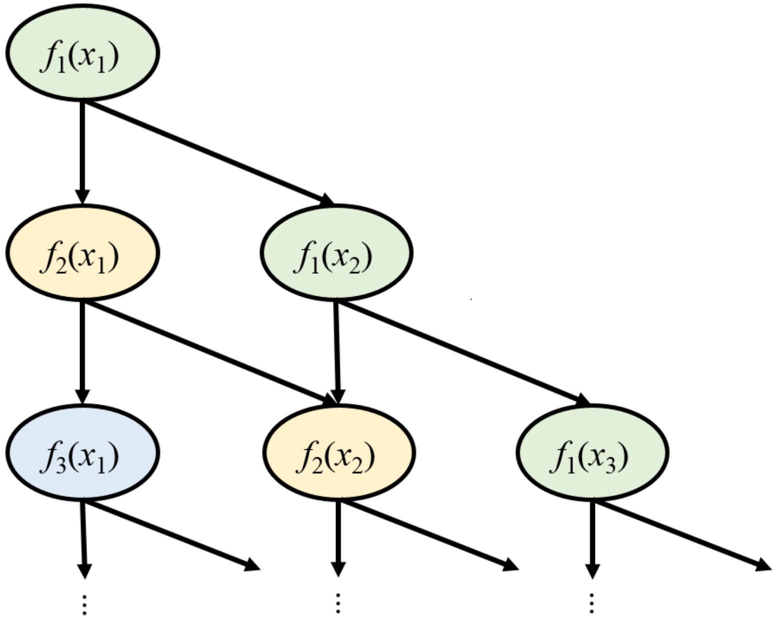
Pipelining

- applies a sequence of dependent computations/tasks (f_1, f_2, ..., f_k) elementwise to data sequence $\boldsymbol{x} = (x_1,x_2,x_3,\dots,x_n)$
  - For fixed x_j, must compute $f_i(x_j)$ before $f_{i+1}(x_j)$
  - … and $f_i(x_j)$ before $f_i(x_{j+1})$ if the tasks f_i have a *run-time state*





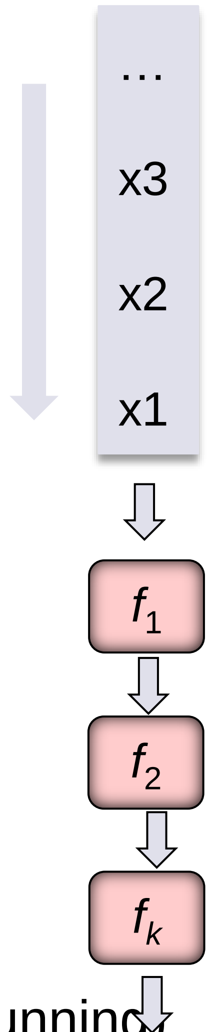
stage
task
dependence
graph



pipeline
task instance
dependence
graph

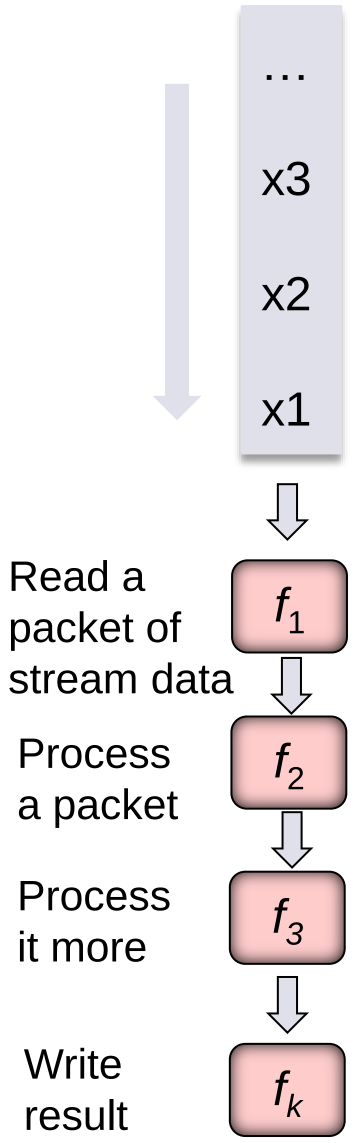
Pipelining

- applies a sequence of dependent computations/tasks ($f_1, f_2, \ldots, f_k$) elementwise to data sequence $\boldsymbol{x} = (x_1,x_2,x_3,\ldots,x_n)$
  - For fixed x_j, must compute $f_i(x_j)$ before $f_{i+1}(x_j)$
  - … and $f_i(x_j)$ before $f_i(x_{j+1})$ if the tasks f_i have a *run-time state*
- **Parallelizability:** Overlap execution of all f_i for k subsequent x_j
  - *time=1: compute* $f_1(x_1)$
  - *time=2: compute* $f_1(x_2)$ *and* $f_2(x_1)$
  - *time=3: compute* $f_1(x_3)$ *and* $f_2(x_2)$ *and* $f_3(x_1)$
  - ...
  - Total time: $O\,(\,(n+k)\max_i(\,time(f_i)\,))$ with k processors
  - Still, requires good mapping of the tasks f_i to the processors for even load balancing – often, static mapping (done before running)
- **Notation** with higher-order function:
  - $(y_1,\ldots,y_n)$ = ***pipe*** $(\,(f_1, \ldots, f_k), (x_1,\ldots,x_n)\,)$



Streaming

- **Streaming** applies pipelining to processing of large (possibly, infinite) data *streams* from or to memory, network or devices, usually partitioned in fixed-sized data packets,
  - in order to **overlap** the processing of each packet of data **in time** with access of *subsequent* units of data and/or processing of preceding packets of data.
- Examples
  - Video streaming from network to display
  - Surveillance camera, face recognition
  - Network data processing e.g. deep packet inspection



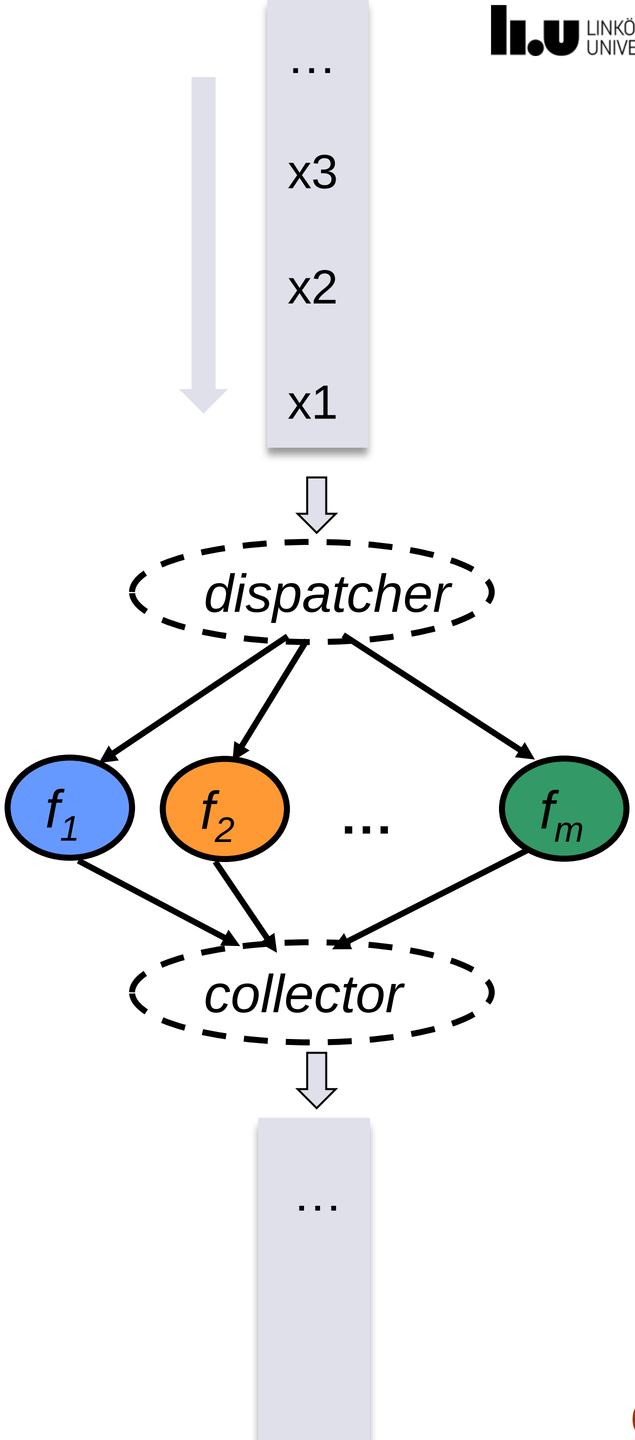
Stream Farming

Combining streaming and task farming patterns

- Independent streaming subcomputations *$f_1, f_2, ..., f_m$* on each data packet

- Speed up the pipeline by parallel processing of subsequent data packets

- In most cases, the original order of packets must be kept after processing

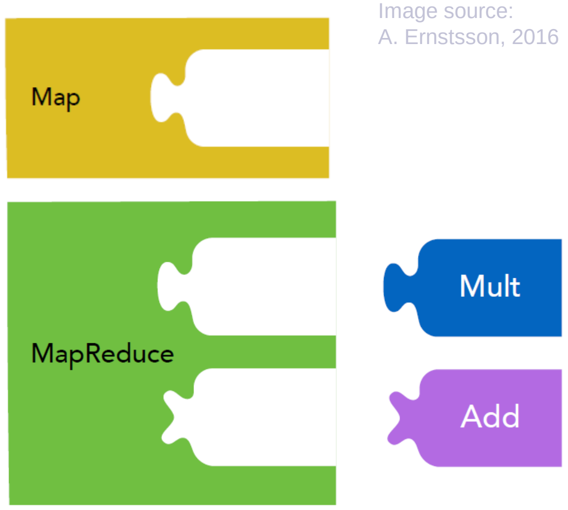


(Algorithmic) Skeletons

- ***Skeletons*** are reusable, parameterizable SW components with well defined semantics for which efficient parallel implementations may be available.
- Inspired by higher-order functions in functional programming
- One or very few skeletons per parallel algorithmic paradigm
  - map, farm, DC, reduce, pipe, scan ...
- Parameterised in user code
  - Customization by instantiating a skeleton template in a user-provided function
- Composition of skeleton instances in program code normally by sequencing+data flow
  - e.g. *squaresum(x)* can be defined by

```
{
    tmp = map( sqr, x );
    return reduce( add, tmp );
}
```

Image source: A. Ernstsson, 2016



For frequent combinations, may define advanced skeletons, e.g.:

```
{
   mapreduce( sqr, add, x )
}
```

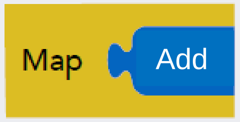
SkePU [Enmyren, K. 2010]

- Skeleton programming library for heterogeneous multicore systems, based on C++
- Example: Vector addition in SkePU-2 [Ernstsson 2016]

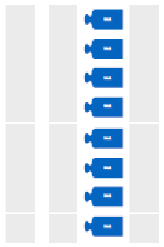
Image source:
A. Ernstsson, 2016

```
int add(int a, int b)
{
    return a + b;
}
```

```
auto vec_add = Map<2>(add);
```



```
vec_add(result, v1, v2);
```



SkePU [Enmyren, K. 2010]

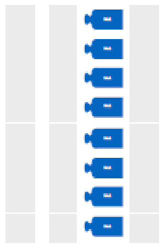
- Skeleton programming library for heterogeneous multicore systems, based on C++
- Example: Vector sum in SkePU-2 [Ernstsson 2016]

Image source: A. Ernstsson, 2016

```
int add(int a, int b)
{
    return a + b;
}
```

```
auto vec_sum = Reduce(add);
```

```
vec_sum(result, v1, v2);
```



High-Level Parallel Programming with Skeletons

- **Skeletons** (constructs) *implement* (parallel) **algorithmic design patterns**

☺ Abstraction, hiding complexity (parallelism and low-level programming)

😐 Enforces structuring, restricted set of constructs

☺ Parallelization for free

☺ Easier to analyze and transform

☹ Requires complete understanding and rewriting of a computation

☹ Available skeleton set does not always fit

☹ May lose some efficiency compared to manual parallelization

- Idea developed in HPC (mostly in Europe) since the late 1980s.
- Many (esp., academic) frameworks exist, mostly as libraries
- Industry (also beyond HPC domain) has adopted skeletons
  - map, reduce, scan in many modern parallel programming APIs
    - e.g., Intel *Threading Building Blocks* (*TBB*): par. for, par. reduce, pipe
    - NVIDIA *Thrust*
  - Google *MapReduce* (for distributed data mining applications)

Further Reading

C. Kessler: ***Design and Analysis of Parallel Algorithms – An Introduction***. Compendium for TDDC78 and TDDD56, Edition Spring 2019. PDF, 123 pages.
http://www.ida.liu.se/~TDDC78/handouts (login: parallel, password see whiteboard)

- Chapter 2 on analysis of parallel algorithms as background reading

On PRAM model and Design and Analysis of Parallel Algorithms

- J. Keller, C. Kessler, J. Träff: ***Practical PRAM Programming.*** Wiley Interscience, New York, 2001.
- J. JaJa: ***An introduction to parallel algorithms.*** Addison-Wesley, 1992.
- D. Cormen, C. Leiserson, R. Rivest: ***Introduction to Algorithms,*** Chapter 30. MIT press, 1989, or a later edition.
- H. Jordan, G. Alaghband: ***Fundamentals of Parallel Processing.*** Prentice Hall, 2003.
- A. Grama, G. Karypis, V. Kumar, A. Gupta: ***Introduction to Parallel Computing***, 2nd Edition. Addison-Wesley, 2003.

On skeleton programming, see e.g. our publications on SkePU:

- http://www.ida.liu.se/labs/pelab/skepu

Questions for Reflection

- Model the overall cost of a streaming computation with a very large number *N* of input data elements on a *single* processor
  (a) if implemented as a loop over the data elements
      running on an ordinary memory hierarchy
      with hardware caches (see above)
  (b) if overlapping computation for a data packet
      with transfer/access of the next data packet
      (b1) if the computation is CPU-bound
      (b2) if the computation is memory-bound
- Which property of streaming computations makes it possible to overlap computation with data transfer?
- Can each dataparallel computation be streamed?
- What are the performance advantages and disadvantages of large vs. small packet sizes in streaming?
- Why should servers in datacenters running I/O-intensive tasks (such as disk/DB accesses) get many more tasks to run than they have cores?
- How would you extend the skeleton programming approach for computations that operate on secondary storage (file/DB accesses)?