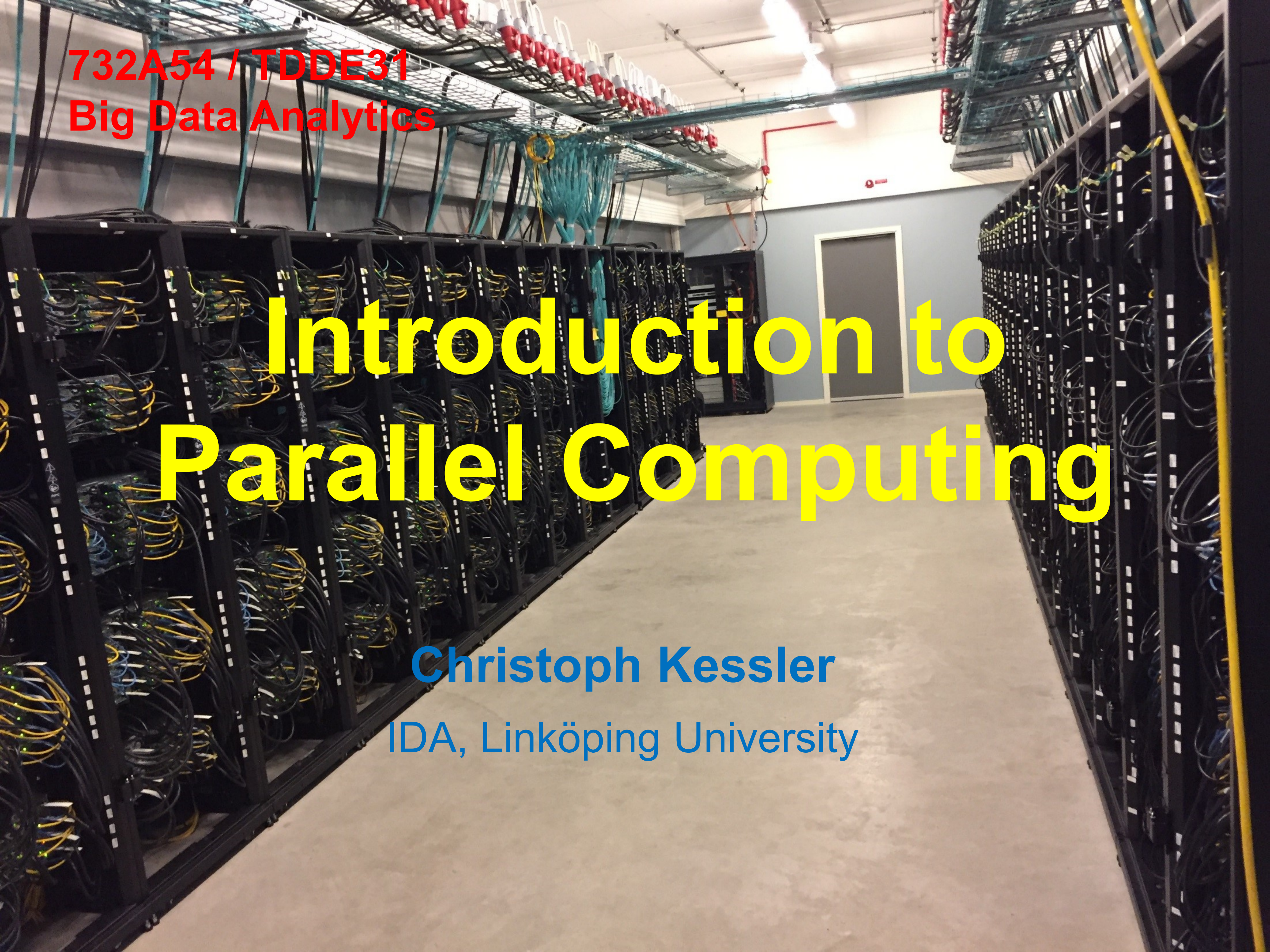


732A54 / TDDE31
Big Data Analytics

Introduction to Parallel Computing

Christoph Kessler

IDA, Linköping University

5 Lectures

- Lectures 1-2: *Introduction* to parallel computing
  - Parallel architectural concepts
  - Parallel algorithms design and analysis
  - Parallel algorithmic patterns and skeleton programming
- Lecture 3: MapReduce
- Lecture 4: Spark
- Lecture 5: Cluster management systems.
  Selected exercises (exam training).

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

NSC Tetralith

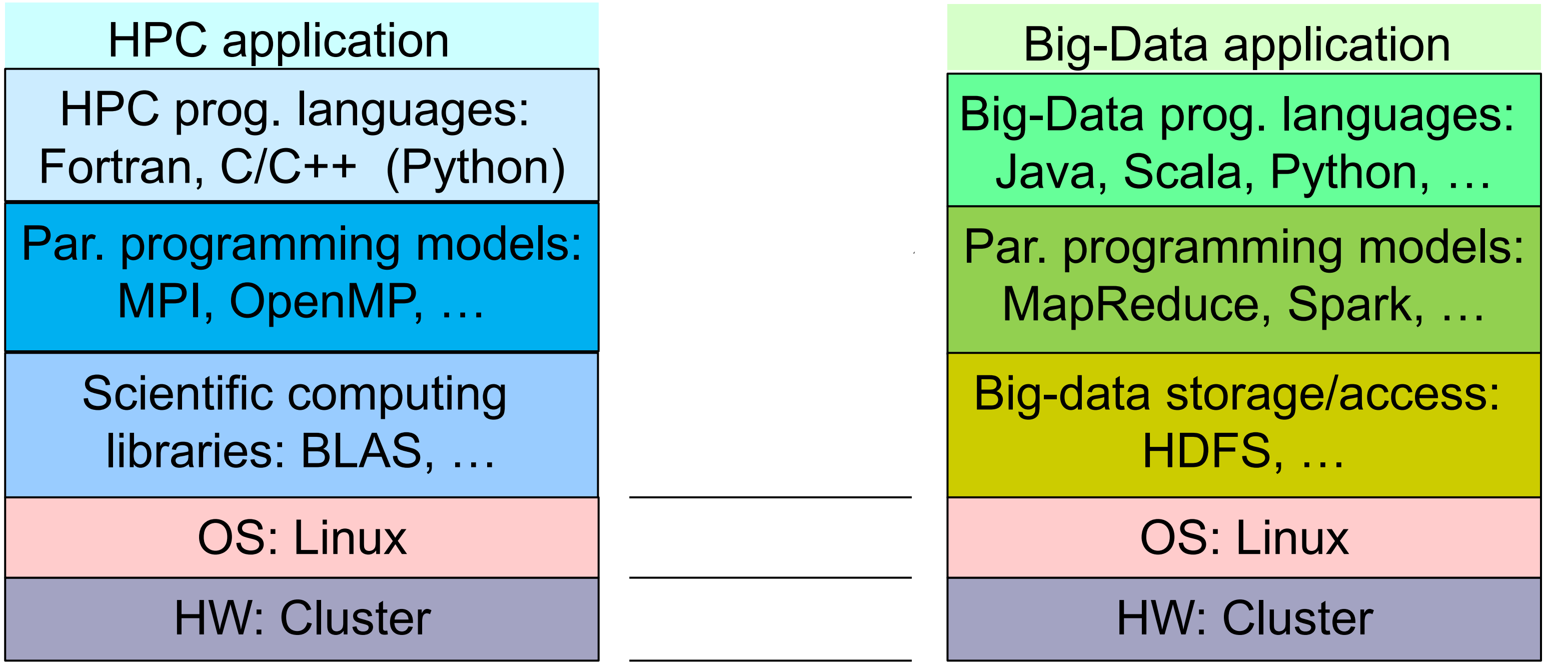
More Recent Use of Parallel Computing: Big-Data Analytics Applications

- **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,

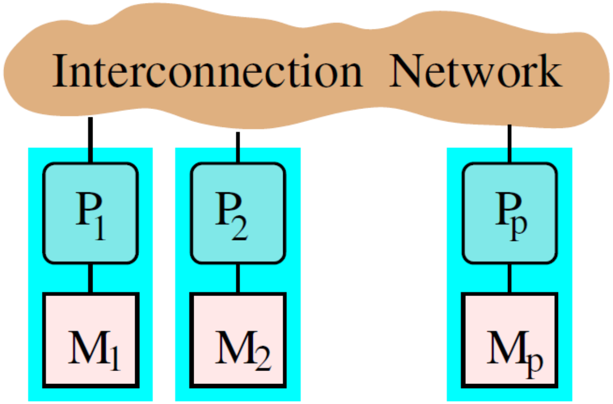
+ one or more memory modules,

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

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

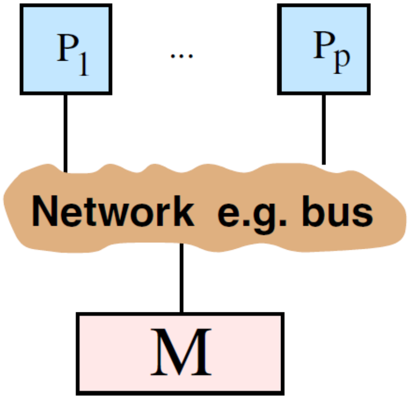
Multicomputer: more loosely connected, e.g. distributed memory

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

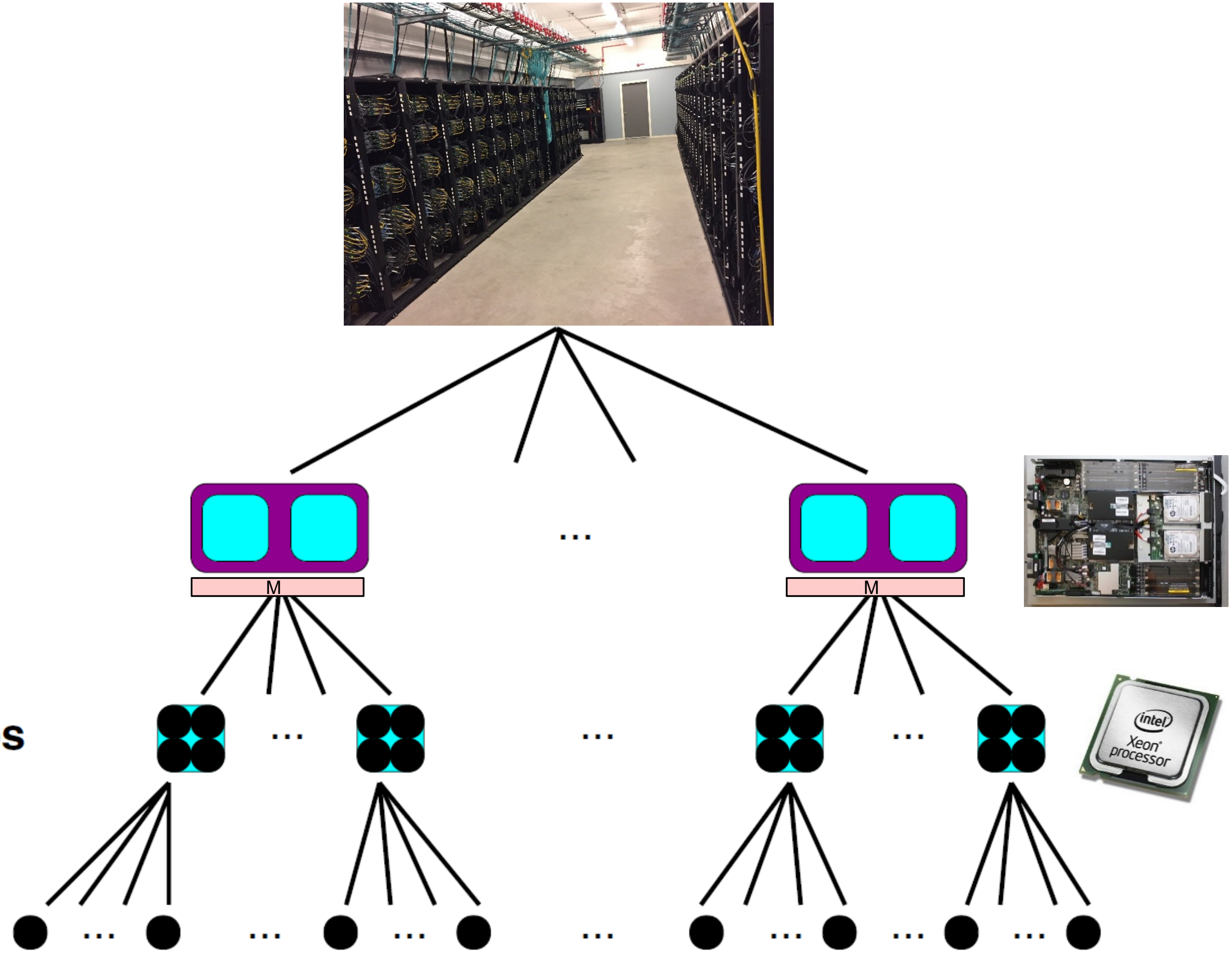
Hybrid (Distributed + Shared) Memory

System

Nodes

Processor chips

Cores



Interconnection Networks (1)

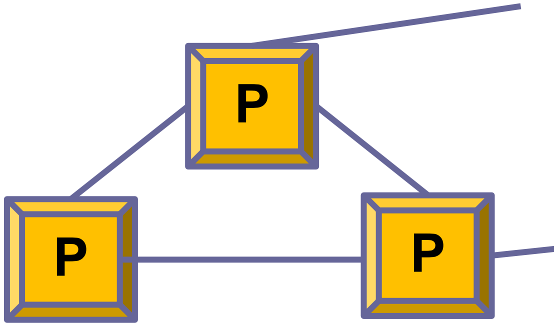
- **Network**
  = physical interconnection medium (wires, switches)
  + communication protocol

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

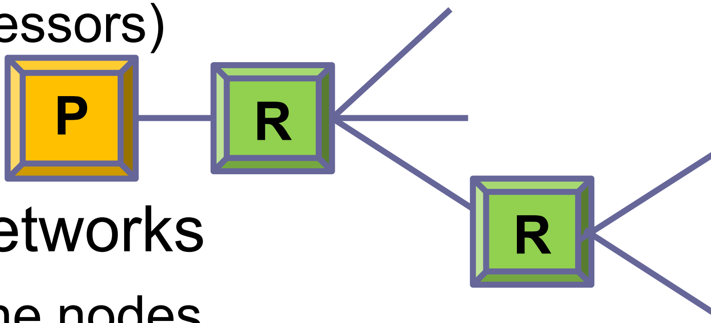
  (b) connecting processors with memory modules (SMS)

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

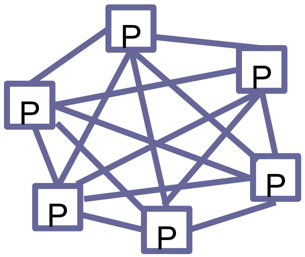


- Switched / dynamic interconnection networks
  - Graphs of routers (switches) connecting the nodes



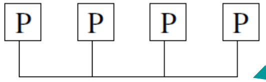
Interconnection Networks (2): Simple Topologies

fully connected



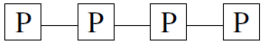
one server node

bus

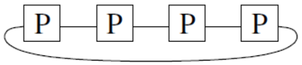


1 wire – bus saturation with many processors
e.g. Ethernet

linear array

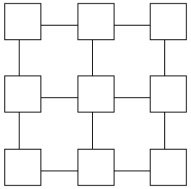


ring

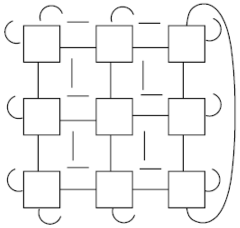


e.g. Token Ring

2D grid

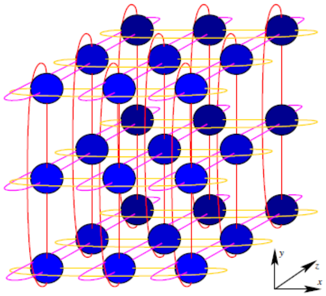


torus:

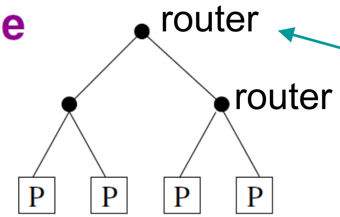


3D grid

3D torus



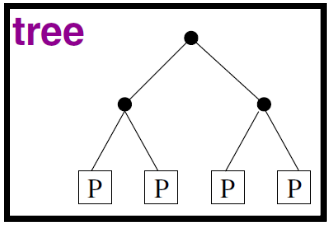
tree



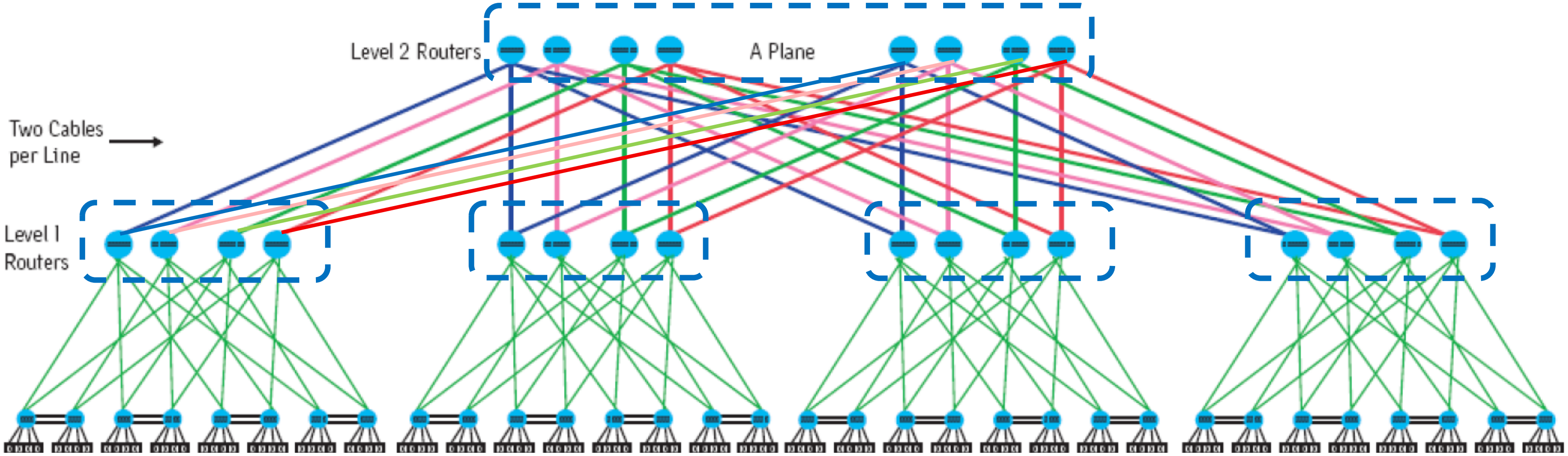
Server nodes in leaves only

All communication paths sender-receiver go via at most 2 $\log_2 N$ hops (with N nodes), but the root (and other upper levels) in the tree network is the communication bottleneck

Interconnection Networks (3): Fat-Tree Network



- Switching network extended for higher communication bandwidth in the layers closer to the root (more switches, more links)
  - avoids bandwidth bottleneck
  - still logarithmic length of longest communication distance



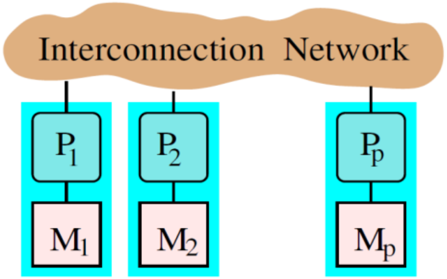
- Example: Infiniband network, Omnipath network

More about Interconnection Networks

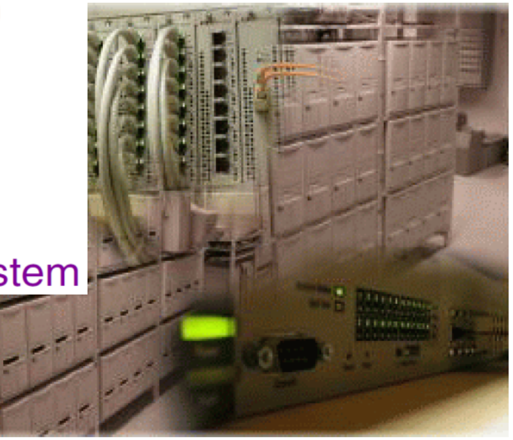
- Hypercube, Crossbar, Butterfly, Hybrid networks… → TDDE65
    - Switching and routing algorithms

- **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:

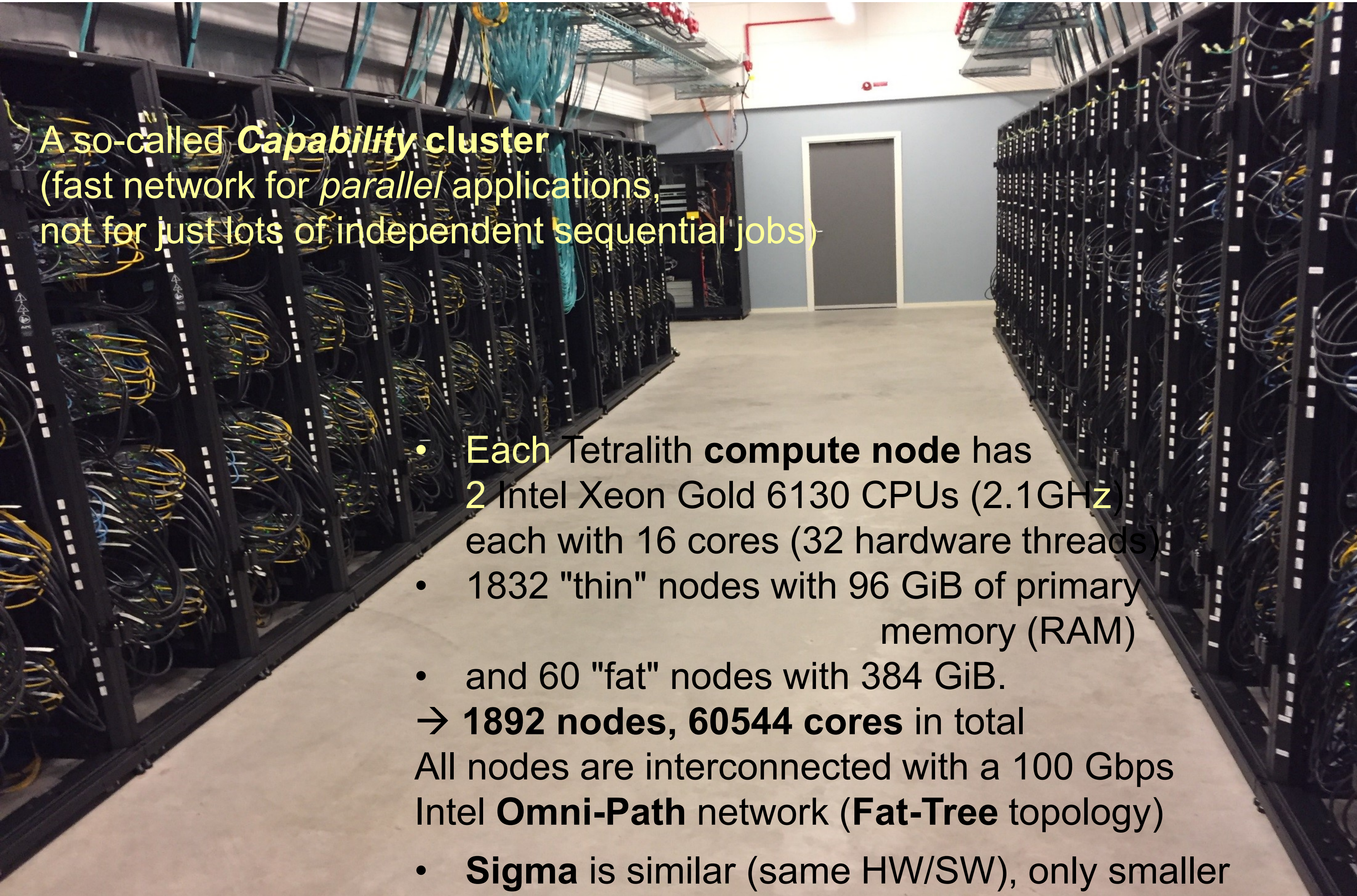
- off-the-shelf (PC) nodes with off-the-shelf CPUs (Xeon, Opteron, …)
- commodity interconnect G-Ethernet, Myrinet, Infiniband, SCI
- Open Source Unix Linux, BSD
- Message passing computing MPI, PVM

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* challenge**s: **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, data 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 →

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

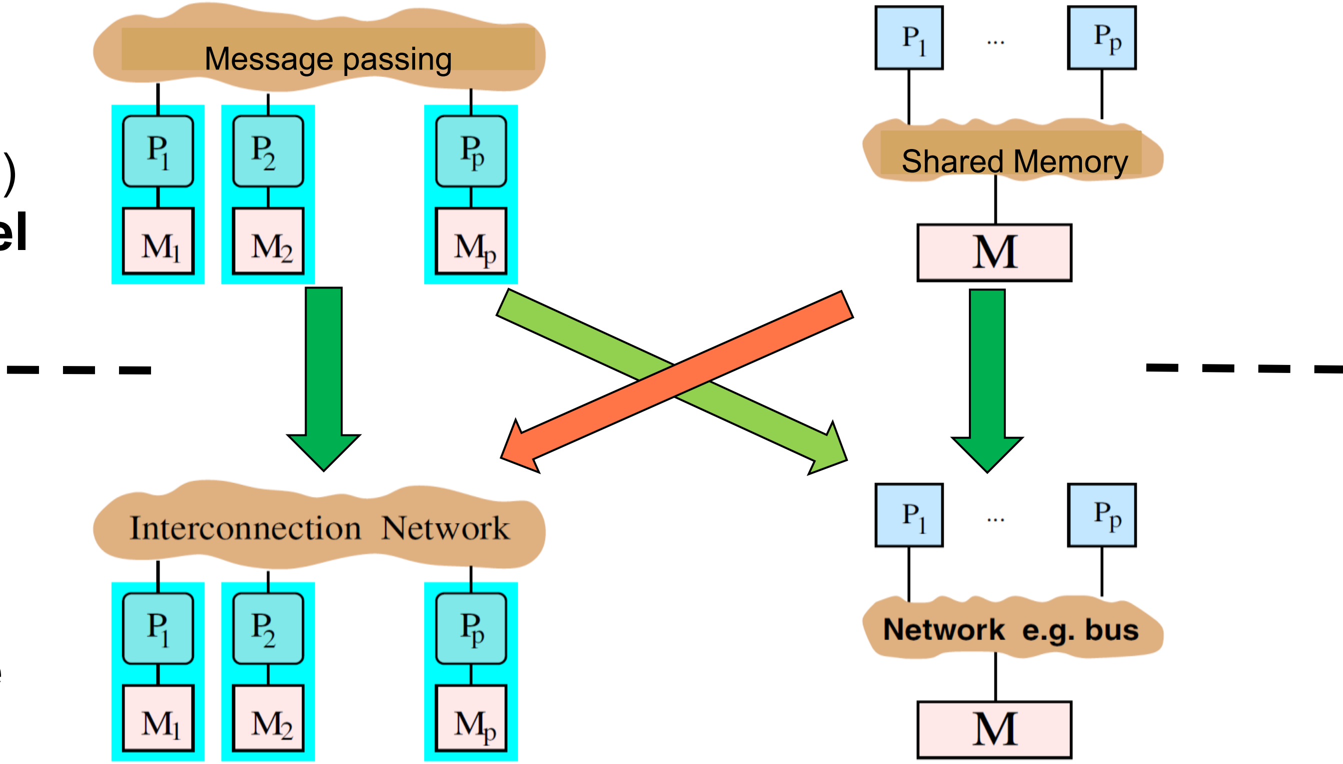
- System-software-enabled **programmer's view** of the underlying hardware
- **Abstracts** from details of the underlying architecture, e.g. network topology
- Focuses on **a few characteristic properties**, e.g. memory model

→ **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**



Distributed memory system

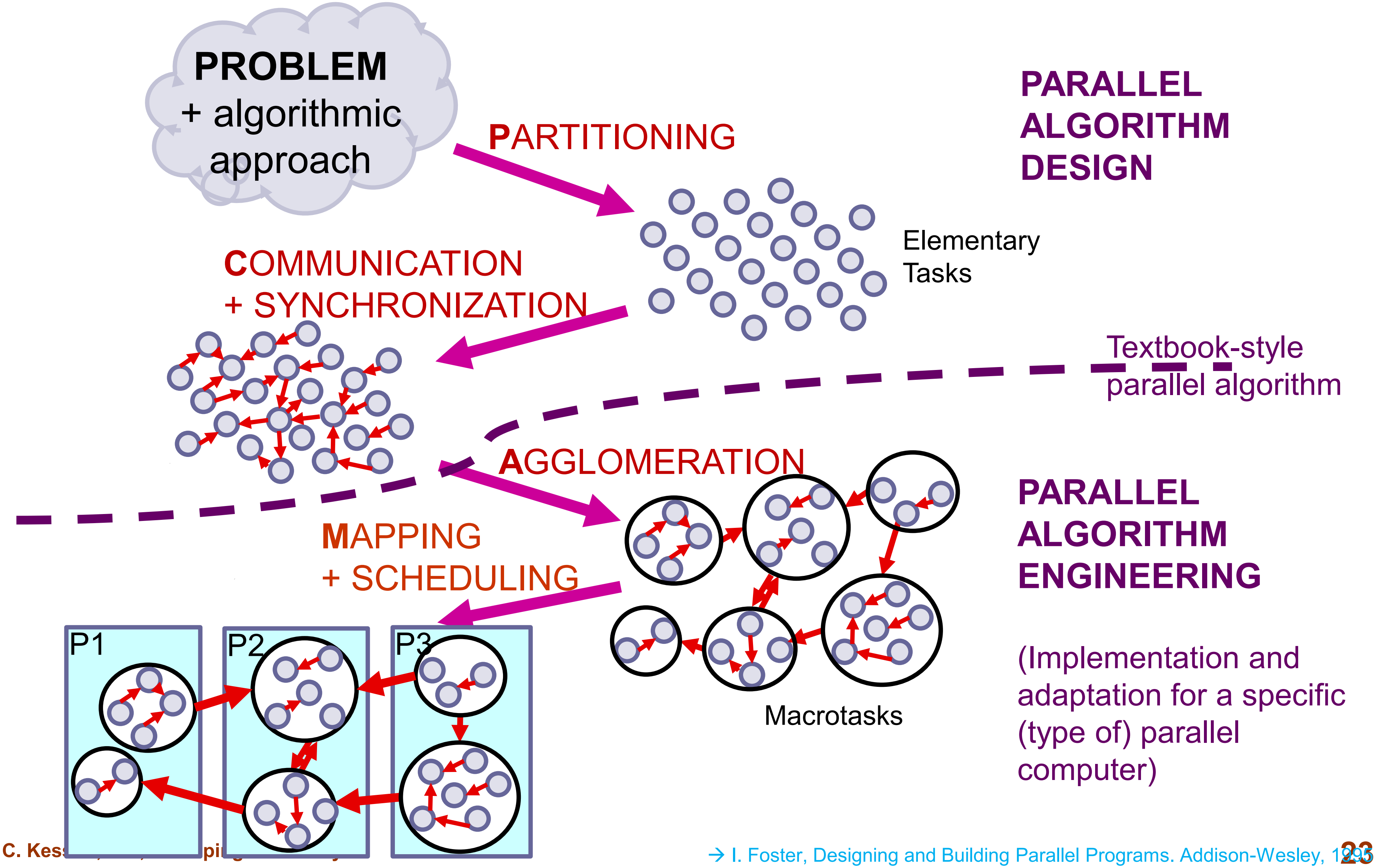
Shared memory system

Design and Analysis of Parallel Algorithms

Introduction

Christoph Kessler, IDA,
Linköpings universitet.

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



PARALLEL ALGORITHM DESIGN

PARALLEL ALGORITHM ENGINEERING

(Implementation and adaptation for a specific (type of) parallel computer)

→ I. Foster, Designing and Building Parallel Programs. Addison-Wesley, 1995.

Parallel Computation Model
= Programming Model + Cost Model

+ abstract from hardware and technology

+ specify basic operations, when applicable

+ specify how data can be stored

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

→ 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 TDDE65/TDDD56, (c) 2023.
https://www.ida.liu.se/~TDDE65/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 → 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

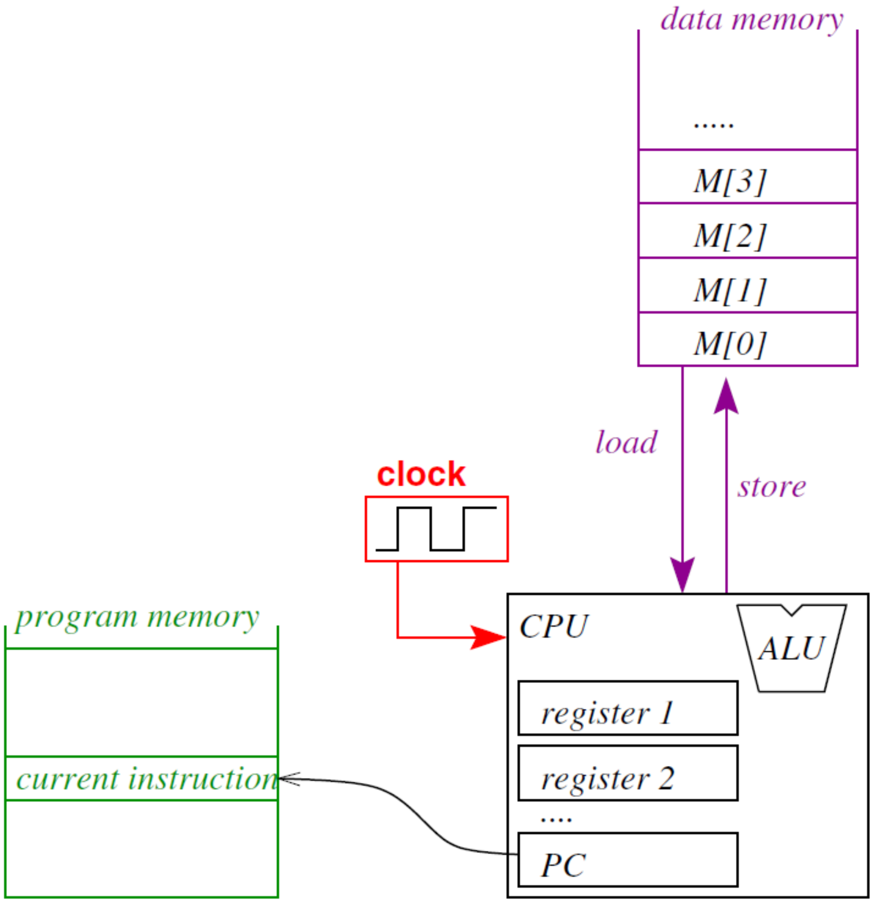
  rather than developing more complex models

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

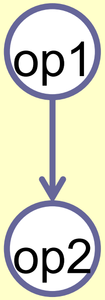


Basic operations (instructions):

- 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)



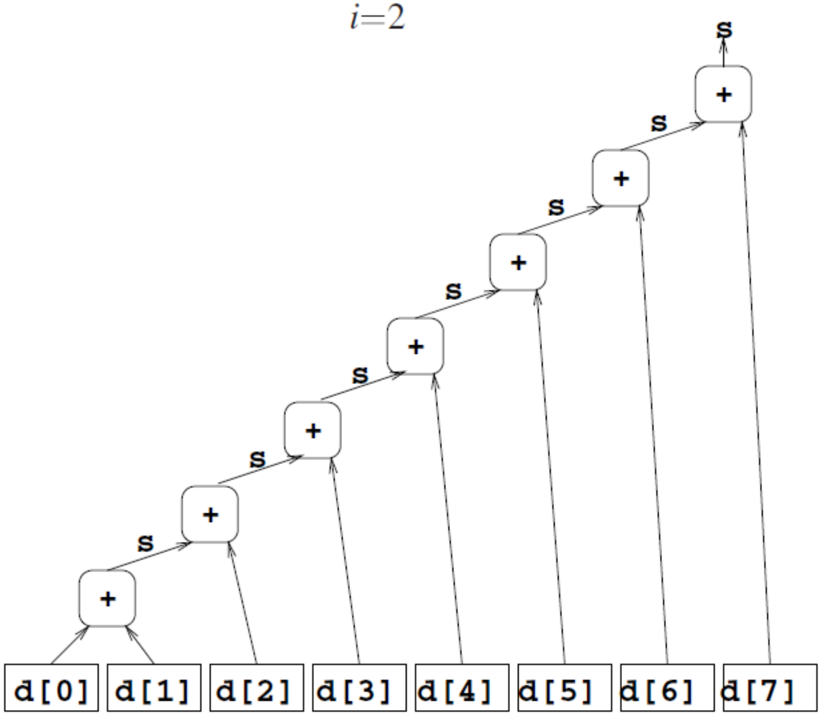
Analysis of sequential algorithms: RAM model (Random Access Machine)

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

Algorithm analysis: Counting instructions

Example: Computing the global sum of N elements

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



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

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

C.

The PRAM Model – a Parallel RAM

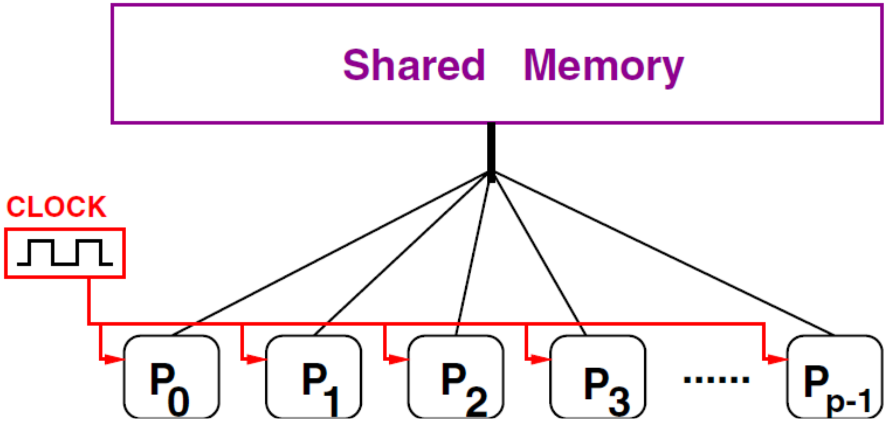
Parallel Random Access Machine [Fortune/Wyllie'78]

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

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

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

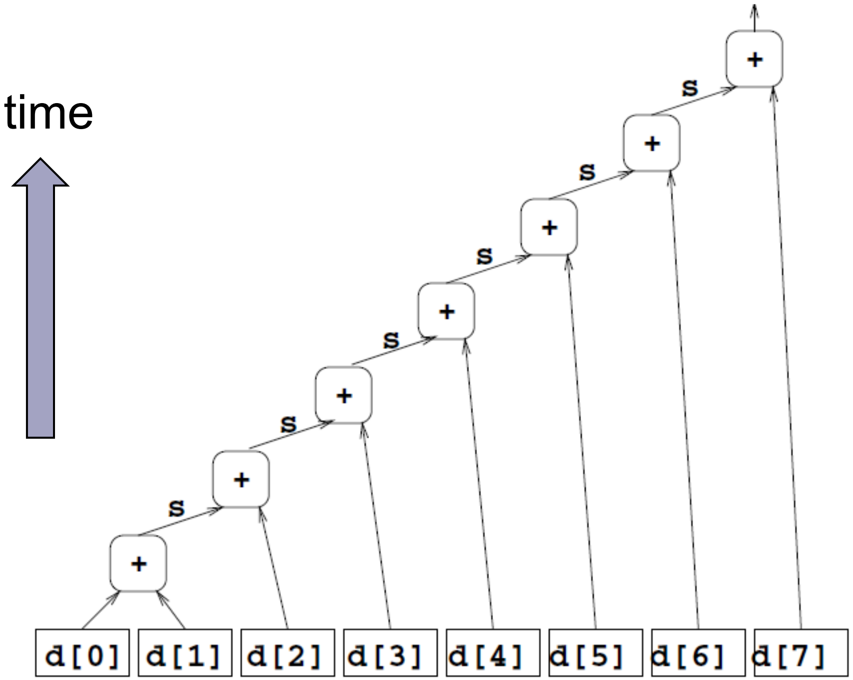
→ focus on pure, fine-grained parallelism

→ 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.

+ is *associative:*

$(x_1 + x_2) + x_3 = x_1 + (x_2 + x_3)$

Idea:

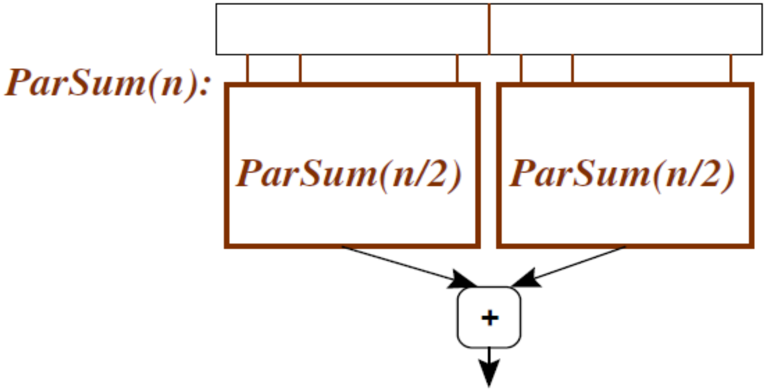
+ *associative* → $((x_1 + x_2) + x_3) + x_4 = (x_1 + x_2) + (x_3 + x_4)$

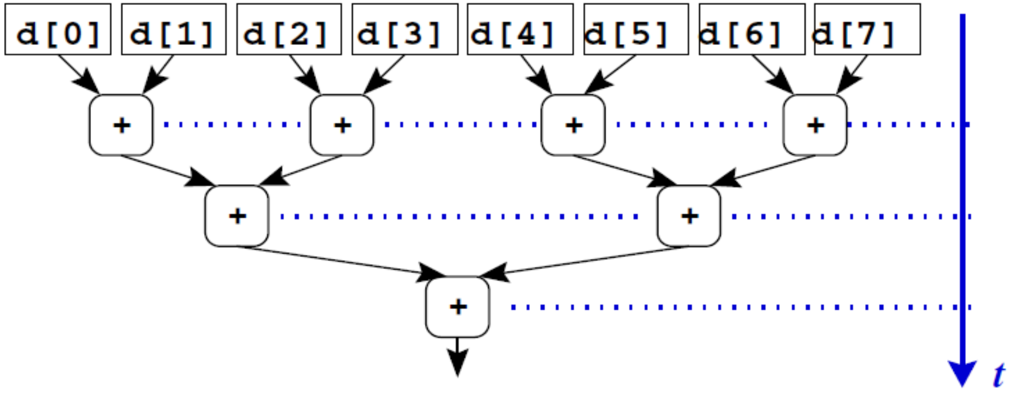
Divide&Conquer Parallel Sum Algorithm in the PRAM / Circuit (DAG) cost model

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.

Parallel algorithmic paradigm used: Parallel Divide-and-Conquer



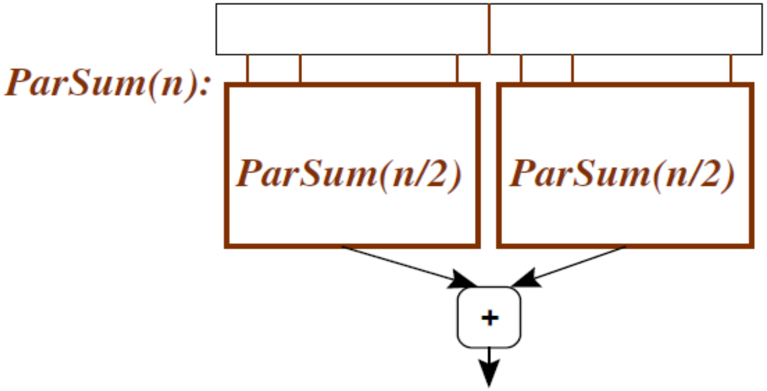


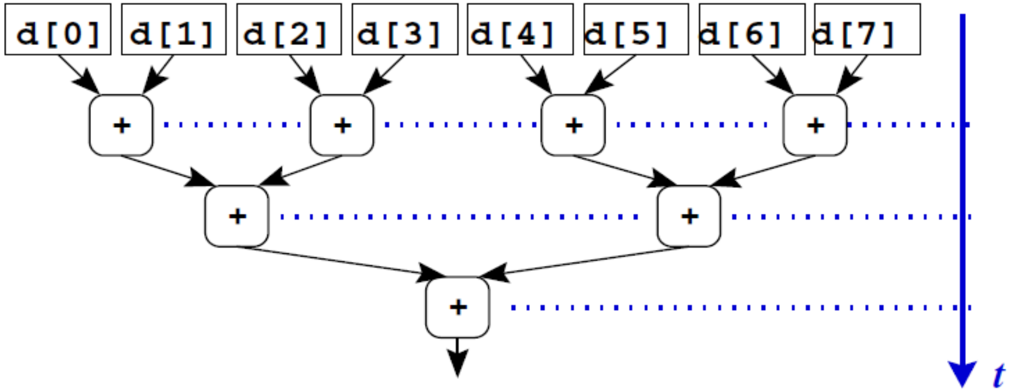
Divide&Conquer Parallel Sum Algorithm in the PRAM / Circuit (DAG) cost model

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.

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 $O(1)$

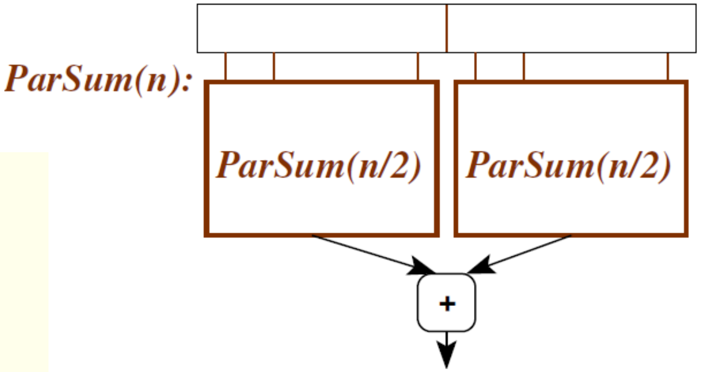
Recurrence equation for parallel execution time:

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

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

Task-parallel implementation
(shared memory, e.g. in OpenMP 3+, 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 = spawn parsum( d, mid+1, to );
        sync;
        return sumleft + sumright;
    }
}
```

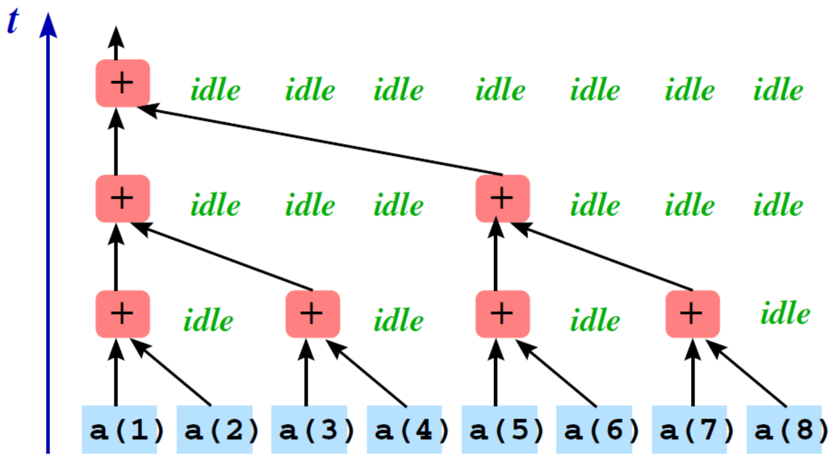
```
// The main program:

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

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

Circuit / DAG model

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


t
+
idle idle idle idle idle idle idle
+ idle idle idle + idle idle idle
+ idle + idle + idle + idle
a(1) a(2) a(3) a(4) a(5) a(6) a(7) a(8)

- **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 → 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:

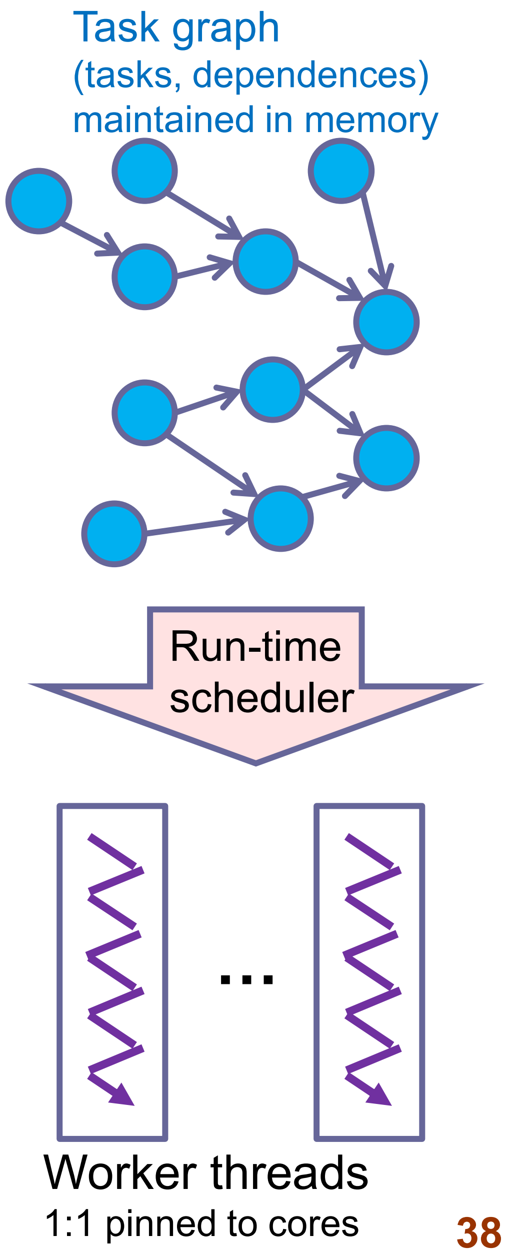
- Requires **algorithm engineering**
- E.g. stop the parallel divide-and-conquer e.g. at subproblem size *n/p* and switch to sequential divide-and-conquer (= task agglomeration)
  - For parallel sum:
    - Step 0. Partition the array of *n* elements in *p* slices of *n/p* elements each (= domain decomposition)
    - 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 once its input operands are available (predecessor tasks have finished).
  - ☺ (More) Automatic load balancing
    - Tasks with their workloads and dependences need not be known prior to runtime
  - ☹ Runtime overhead for explicit task representation and management



Task graph
(tasks, dependences)
maintained in memory

Run-time scheduler

…

Worker threads
1:1 pinned to cores

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

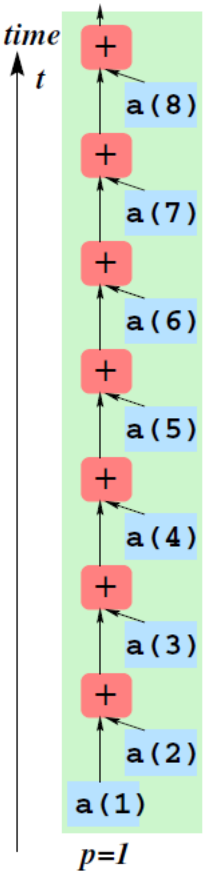
Example:
seq. sum algorithm

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

$n-1$ additions

n loads

$O(n)$ other



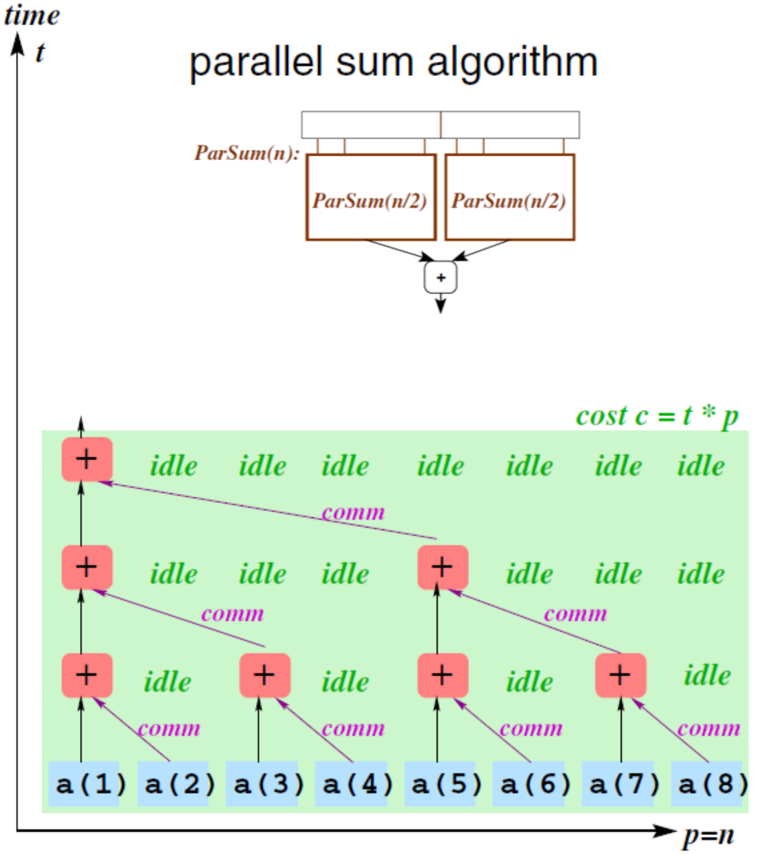
$$t(1,n) = t_{seq}(n) = O(n)$$

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

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

$$= O(n)$$

parallel sum algorithm



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

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

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

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

Background: Parallel Time, Work, Cost

- **Work** is the total number of non-idle-waiting **basic operations** (instructions or other operations taking only a constant number of time steps – arithmetics, memory accesses, branches, … – performed by the algorithm.
  - Hence: parallel work = the sum over the number of such operations on each process(or), accumulated over all process(or)s.
  - Usually, a *worst-case* (over all inputs of same size) metric like time, given as a function in the size of the input.
  - In sequential computing, time and work always coincide.
  - We are especially interested in parallel algorithms that are (asymptotically) **work-optimal**, i.e., do not do asymptotically more work than the best sequential algorithm for the same problem.

- (Parallel) **Cost** is the (worst-case) parallel time multiplied by the number of processors used.
  - At least as large as the *work*, but may be larger, even asymptotically larger, due to idle waiting for other processes,
like in the above case of divide-and-conquer parallel sum.
  - In sequential computing, time and cost always coincide.
    - A sequential program never needs to wait for itself.
  - For a **cost-effective** parallel algorithm, its cost = O(work).

c See the compendium for more details and exercises.

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 $S_{abs} = \dfrac{T_s}{T(p)}$

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

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

Speedup *S*(*p*) with *p* processors is usually in the range (0…*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

→ 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

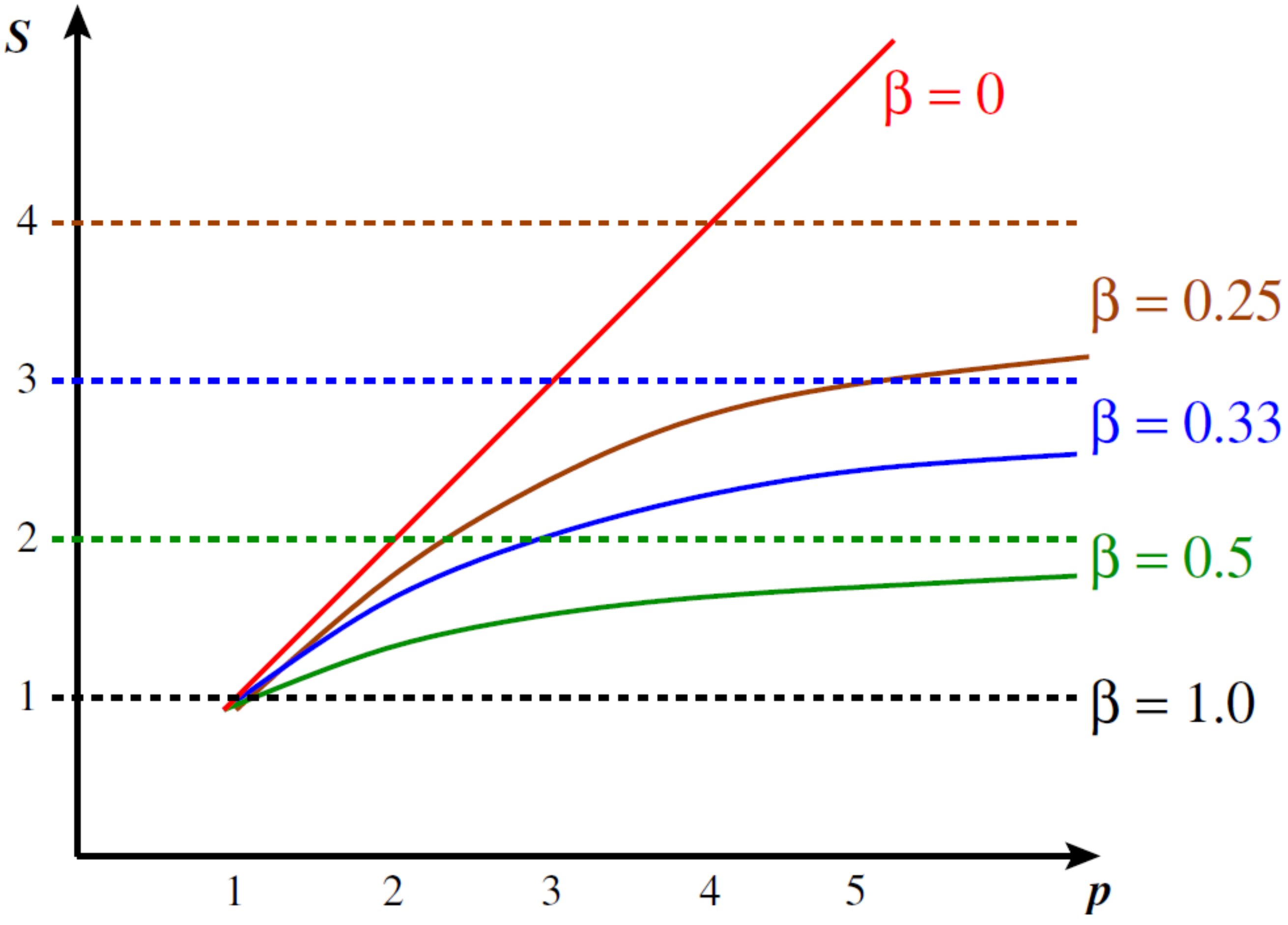
If the sequential part of A is a *fixed* fraction of the total work irrespective of the problem size n, that is, if there is a constant β with

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

the relative speedup of A with p processors is limited by

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

Amdahl's Law



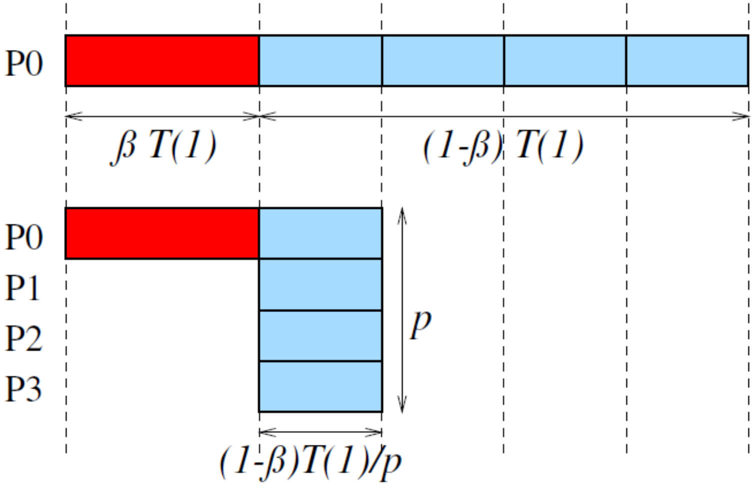
$$S(p) = \frac{p}{\beta p + (1-\beta)} < 1/\beta$$

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, $T_{A^p}(p) = (1-\beta)T(p) = (1-\beta)T(1)/p$:

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



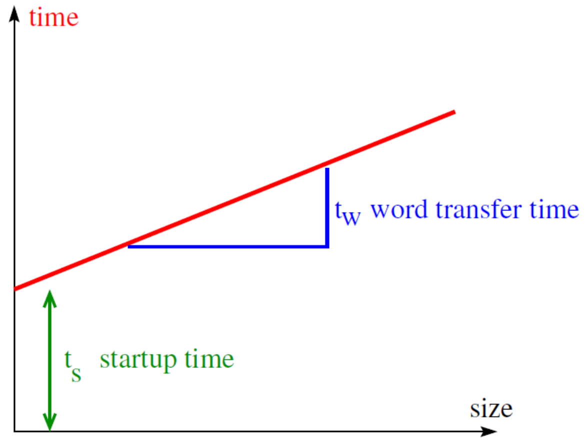
Towards More Realistic Cost Models

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

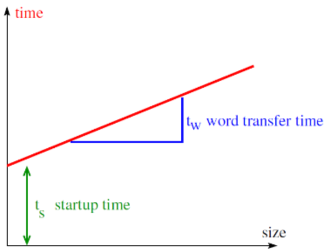
$t_{startup}$ = startup time (time to send a 0-byte message)

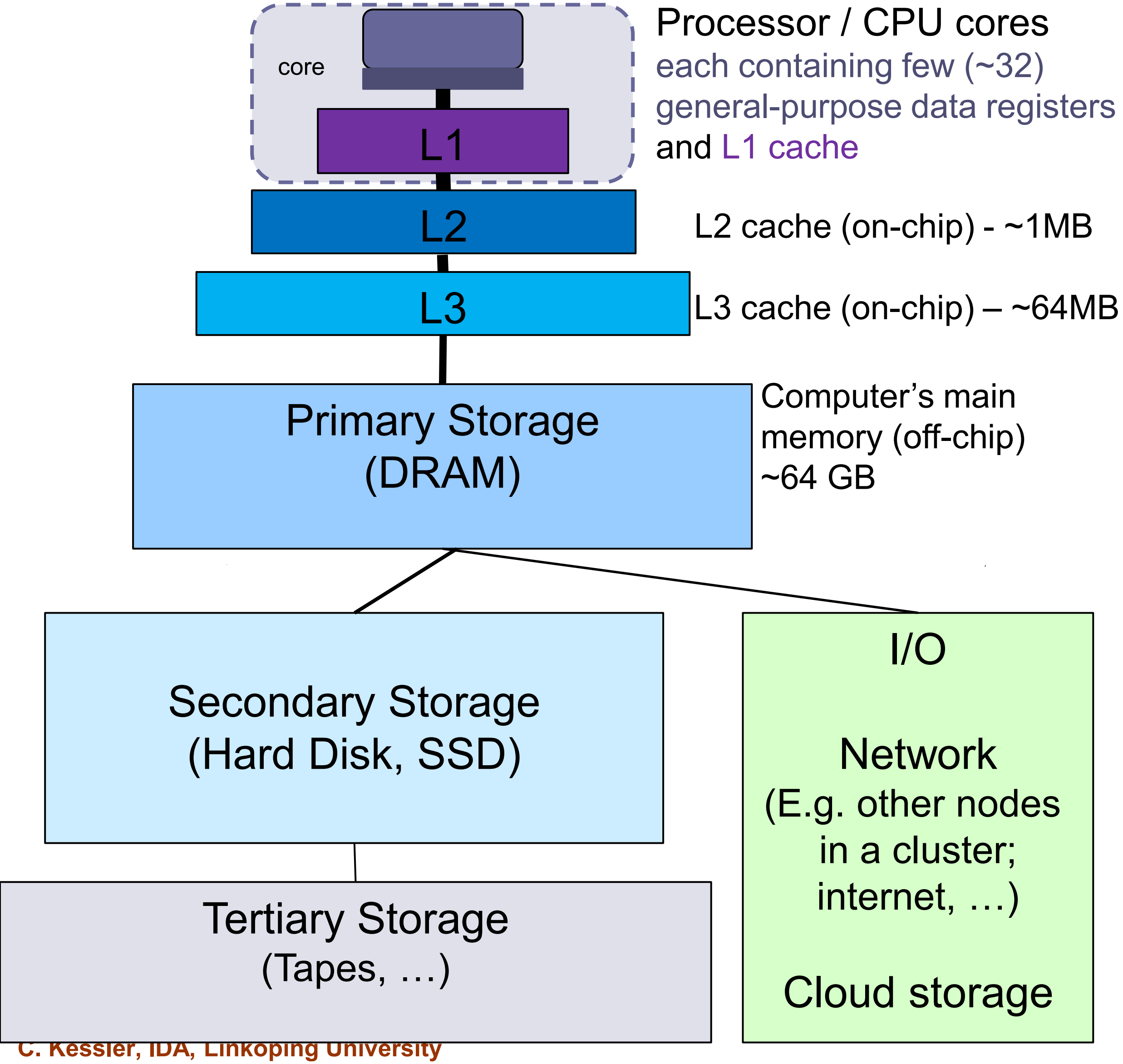
accounts for hardware and software overhead.

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

depends on the network bandwidth.

Memory Hierarchy
And The Real Cost of Data Access





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

L2 cache (on-chip) - ~1MB

L3 cache (on-chip) – ~64MB

Primary Storage (DRAM)

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

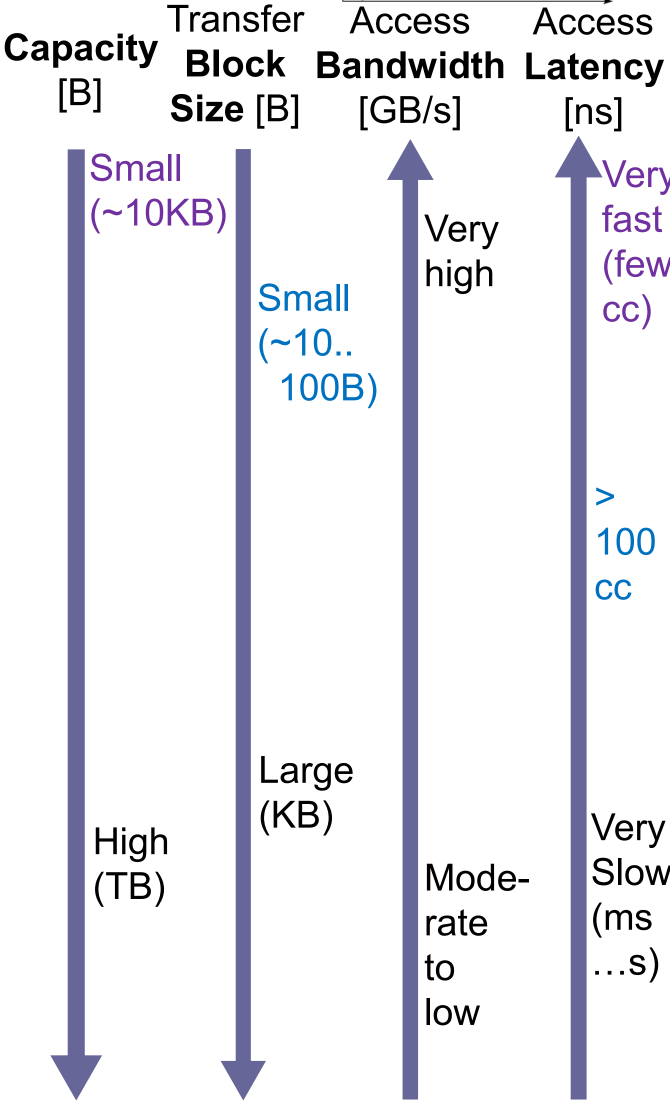
Secondary Storage (Hard Disk, SSD)

Tertiary Storage (Tapes, …)

I/O

Network (E.g. other nodes in a cluster; internet, …)

Cloud storage



Capacity [B]: Small (~10KB) … High (TB)

Transfer **Block Size** [B]: Small (~10.. 100B) … Large (KB)

Access **Bandwidth** [GB/s]: Very high … Mode-rate to low

Access **Latency** [ns]: Very fast (few cc) … > 100 cc … Very Slow (ms …s)

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.

LINKÖPING UNIVERSITY

Some Parallel Algorithmic Design Patterns

Christoph Kessler, IDA,
Linköpings universitet.

Data Parallelism

Given:

- One (or several) data containers $\boldsymbol{x}$, $\boldsymbol{y}$, … *with* n *elements each*, e.g. array(s) $\boldsymbol{x} = (x_1,...x_n)$, $\boldsymbol{y} = (y_1,\ldots,y_n)$, …
- An operation f on individual elements of x, y, … (e.g. *incr, sqrt, mult*, ...)

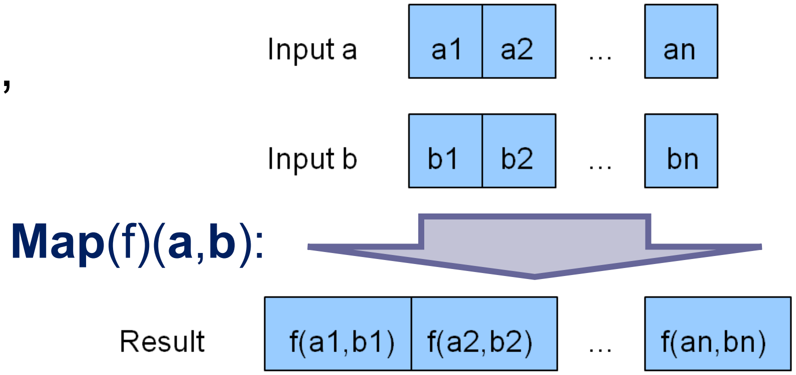
Compute: $\boldsymbol{z} = f(\boldsymbol{x}) = (\ f(x_1), ..., f(x_n)\ )$ (similarly for arities > 1)

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*:*

- $\boldsymbol{z} = \boldsymbol{Map}\ (\ f\ )\ (\ \boldsymbol{x}\ )$



Data-parallel Reduction

Given:

- A data container ***x*** with *n* elements, e.g. array ***x*** = $(x_1,\dots x_n)$
- A binary, associative operation *op* on individual elements of *x* (e.g. *add, max, bitwise-or*, ...)

op *associative:*

$(x_1 \text{ op } x_2) \text{ op } x_3 = x_1 \text{ op } (x_2 \text{ op } x_3)$

Compute: $y = OP_{i=1\dots n}\, \boldsymbol{x} = x_1 \; op \; x_2 \; op \; \dots \; op \; x_n$

Idea:

op *associative* →

$((x_1 \text{ op } x_2) \text{ op } x_3) \text{ op } x_4 = (x_1 \text{ op } x_2) \text{ op } (x_3 \text{ op } x_4)$

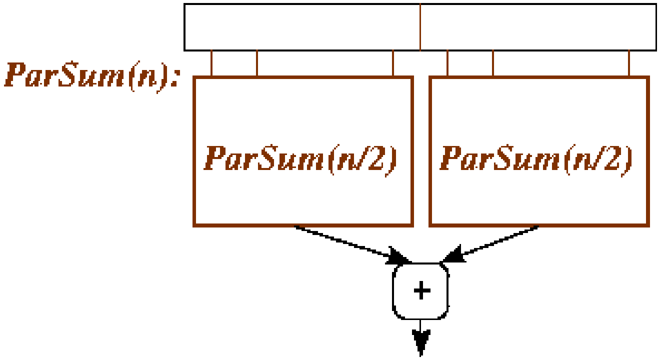
Data-parallel Reduction

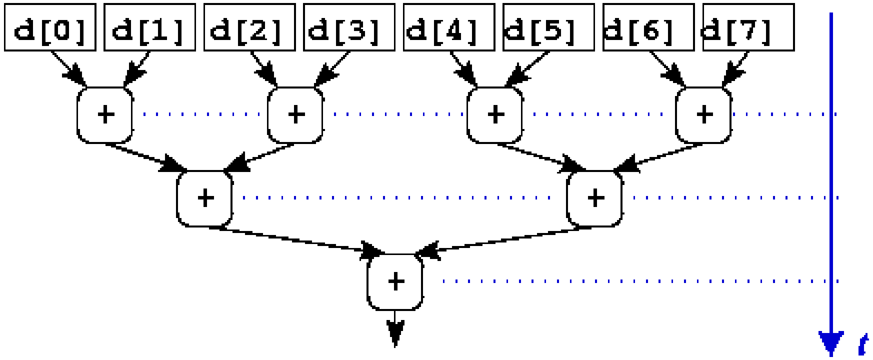
Given:

- A data container ***x*** with *n* elements,
  e.g. array ***x*** = $(x_1,...x_n)$
- A binary, associative operation *op* on individual elements of *x*
  (e.g. *add, max, bitwise-or*, ...)

Compute: $y = OP_{i=1...n}\, \mathbf{x} = x_1\ op\ x_2\ op\ ...\ op\ x_n$

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***: $y = \sum_i x_i * z_i$

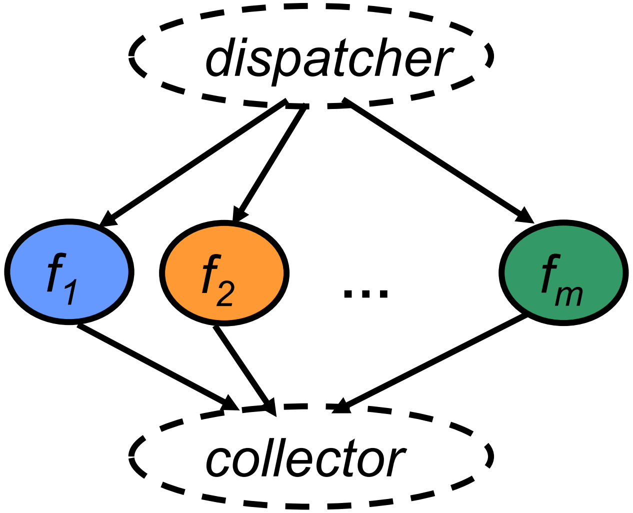
  f = scalar multiplication,

  g = scalar addition

Task Farming

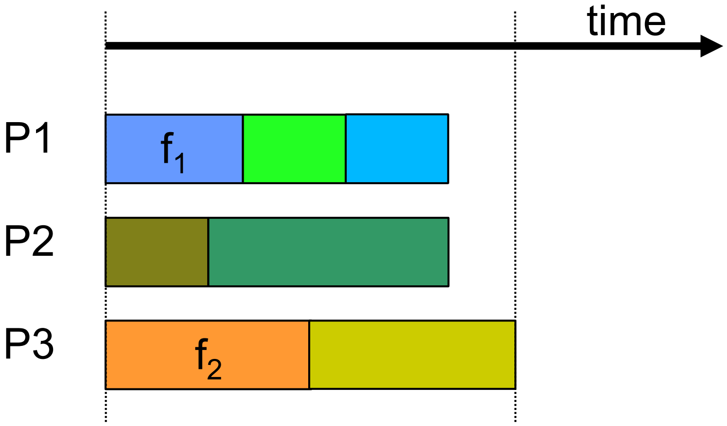
Independent subcomputations $f_1, f_2, ..., f_m$ could be done in parallel and/or in arbitrary order, e.g.

- 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



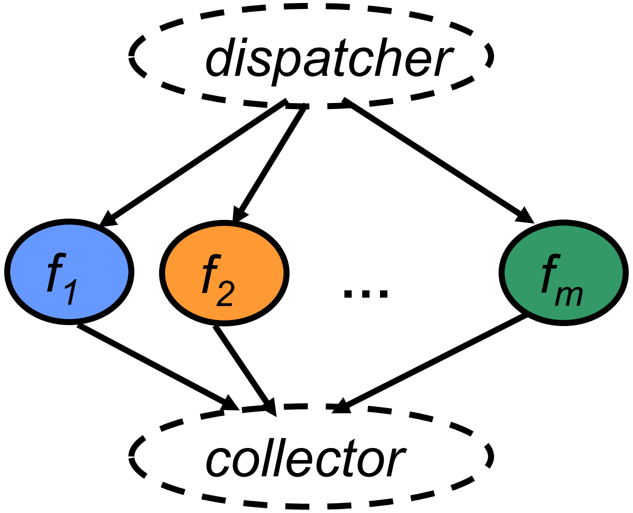
Notation with higher-order function:

- ***Farm*** $(f_1, ..., f_m)$ $(x_1,...,x_n)$

Task Farming

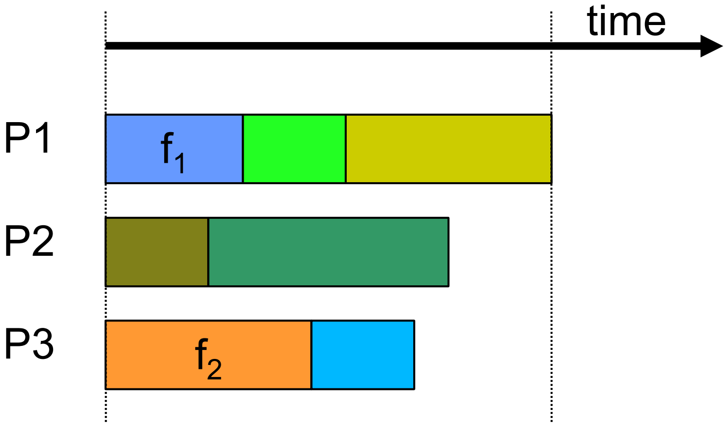
Independent subcomputations $f_1, f_2, \ldots, f_m$ could be done in parallel and/or in arbitrary order, e.g.

- 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*

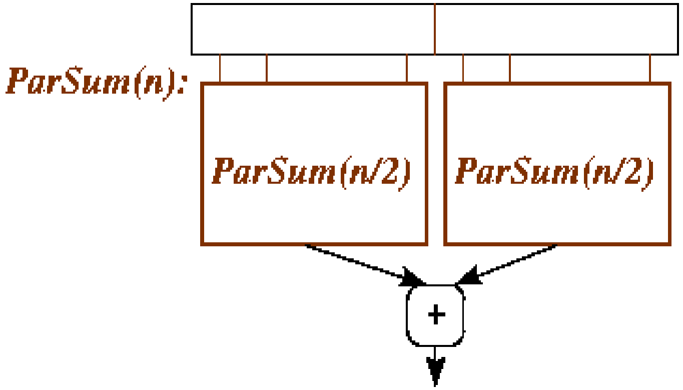
Parallel Divide-and-Conquer:

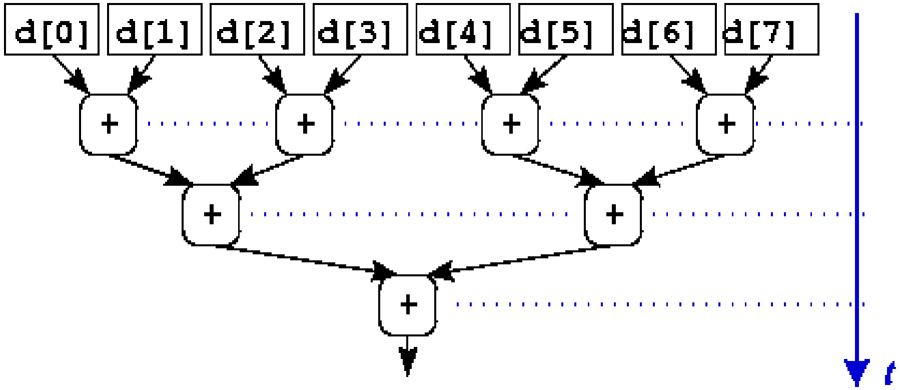
- Recursive calls can be done in parallel.
- Parallelize, if possible, also the divide and combine phase.
- Switch to sequential divide-and-conquer when enough parallel tasks have been created.

Notation with higher-order function:

- *solution* = ***DC*** (*divide, combine, istrivial, solvedirectly*) (*P, n*)

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.

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

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

(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

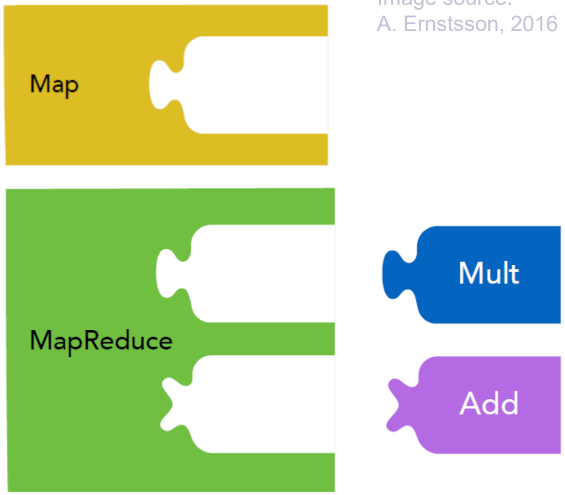


Image source: A. Ernstsson, 2016

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 );
}
```

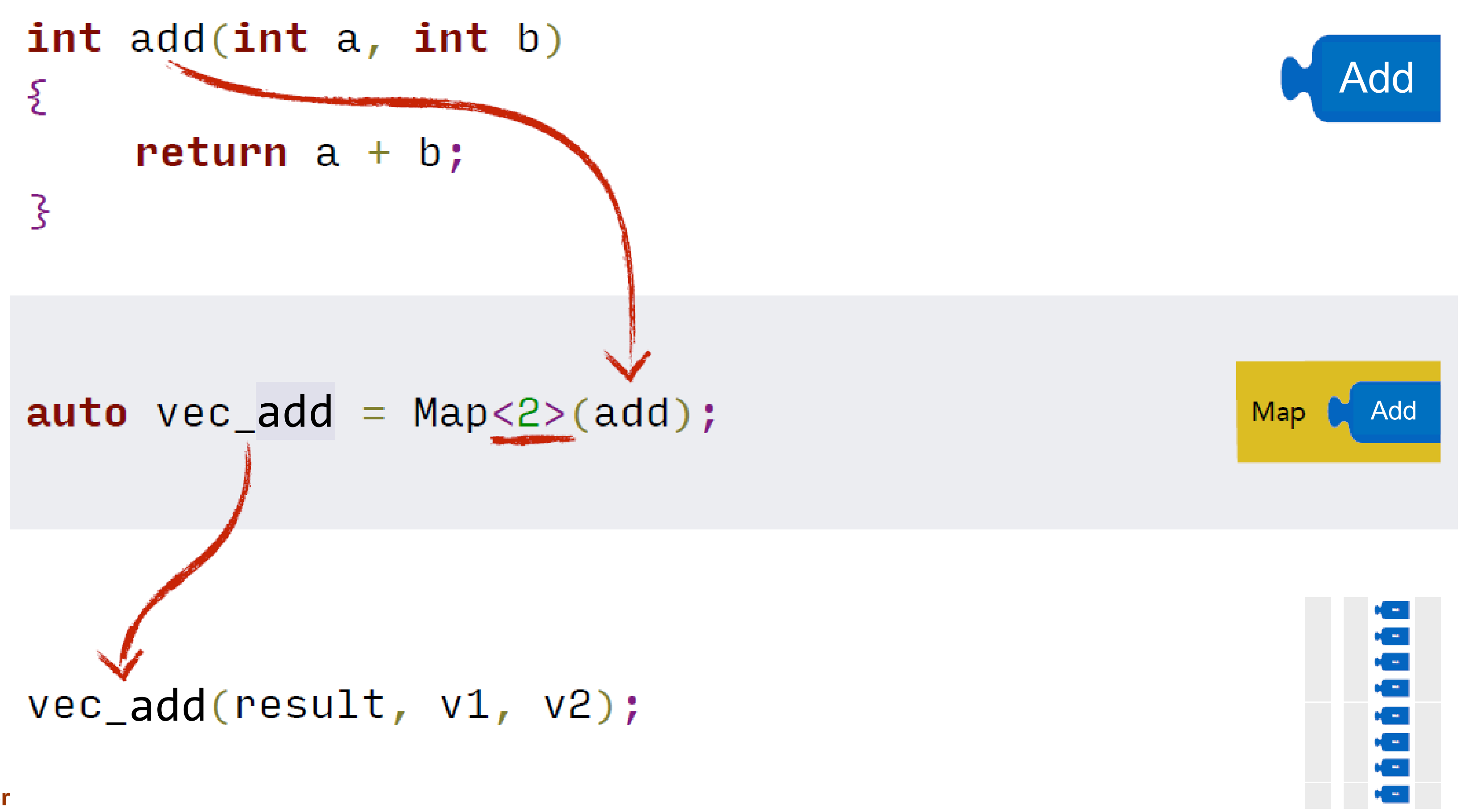
For frequent combinations, may define advanced skeletons, e.g.:

```
{
    MapReduce( sqr, add )( x );
}
```

SkePU https://skepu.github.io

- Skeleton programming library for heterogeneous multicore systems, based on C++
- Example: Vector addition in SkePU [Ernstsson *et al*. 2016, 2021]

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);
```

High-Level Parallel Programming with Skeletons

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

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

😐 Enforces structuring, using a 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/Hadoop *MapReduce,* Apache *Spark*
    (for distributed data mining applications)

Questions for Reflection

- Draw an example task graph with at least 5 tasks that fulfills the assumptions made in Amdahl's Law. (Hint: The divide-and-conquer parallel sum algorithm above does *not* qualify here – why?)
  Show which of the tasks contribute to the parallelizable and to the sequential parts of the work. Assume for simplicity that each task takes 1 unit of time.
  Identify the longest chain of dependences in the task graph and give a good lower bound for the parallel execution time.
- How would you implement a global sum computation *in parallel* for a cluster with distributed memory, given a message-passing programming model with operations for sending and receiving blocks of data in memory?
  Assume that each of the *p* cluster nodes initially has a partition of 1/*p* th of the input array in its memory.
  - What is the parallel time, work and cost complexity of your solution?
  - How would you adapt the above algorithm for more parallelism if the cluster nodes are internally shared-memory parallel computers?
- 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?
- What are the possible advantages and disadvantages of a very fine or very coarse granularity of tasks (work per task) in dynamic scheduling?
- How would you extend the skeleton programming approach for computations that operate on secondary storage (file/DB accesses)?

Further Reading

C. Kessler: ***Design and Analysis of Parallel Algorithms – An Introduction***.
Compendium for the theory part of TDDE65 and TDDD56, Edition Dec. 2023. PDF.
http://www.ida.liu.se/~TDDE65/handouts (login: parallel, password see whiteboard)

- Chapter 2 on analysis of parallel algorithms as background reading

Introduction to programming parallel computers in general:

- Wilkinson, Allen: Parallel Programming, 2nd edition. Addison Wesley, 2004.
- See also literature on programming in MPI, OpenMP and other parallel programming models.

On the PRAM model and the Design and Analysis of Parallel Algorithms:

- J. Keller, C. Kessler, J. Träff: ***Practical PRAM Programming.*** Wiley Interscience, New York, 2001.
- 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 with SkePU:

- https://skepu.github.io