HPI Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Parallel Programming and Heterogeneous Computing

E2 - Summary

Max Plauth, Sven Köhler, Felix Eberhardt, Lukas Wenzel and Andreas Polze
Operating Systems and Middleware Group

Course Topics

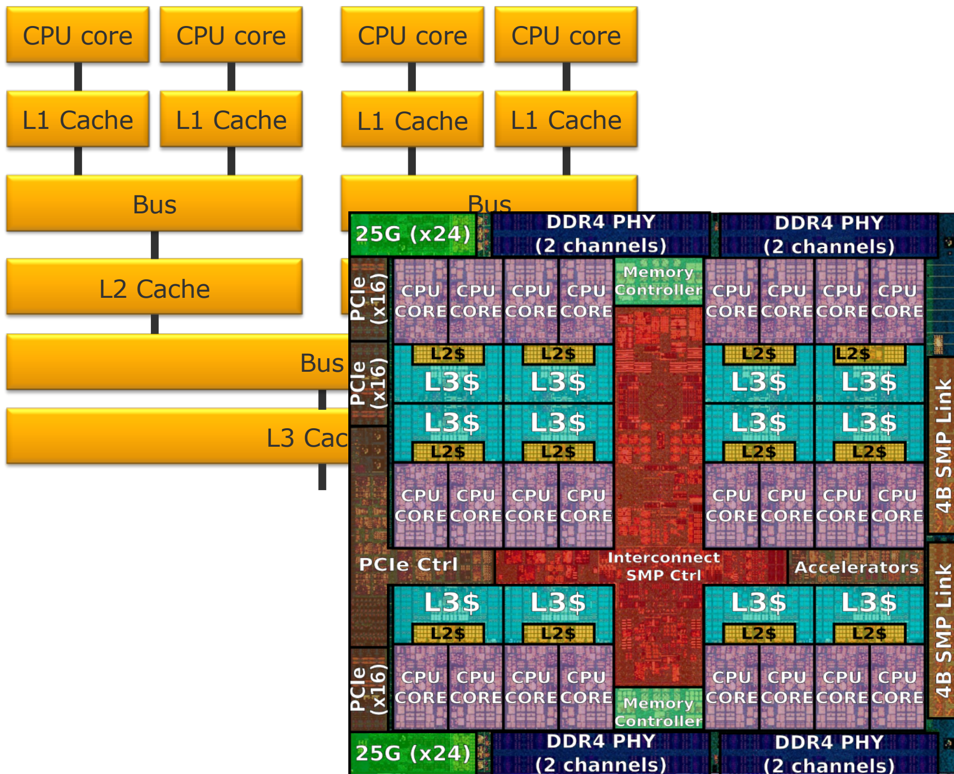
A. The Parallelization Problem
   - Power wall, memory wall, Moore’s law
   - Terminology and metrics
B. Shared Memory Parallelism
   - Theory of concurrency, hardware today and in the past
   - Programming models, optimization, profiling
C. Heterogeneous Computing
   - On-Chip Accelerators (e.g. SIMD, special purpose accelerators, etc.)
   - External Accelerators (e.g. GPUs, FPGAs, etc.)
D. Shared Nothing Parallelism
   - Theory of concurrency, hardware today and in the past
   - Programming models, optimization, profiling

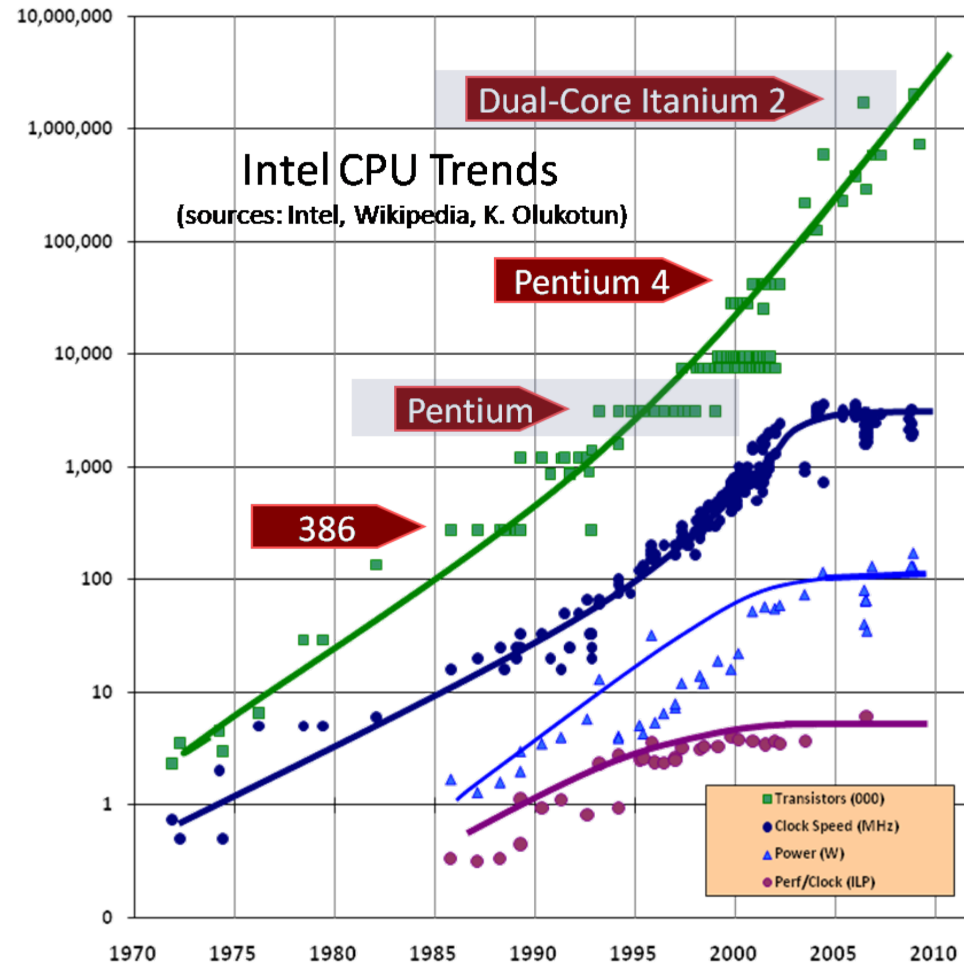
Hasso Plattner Institut
Digital Engineering • Universität Potsdam

A: Why Parallel?, Terminology, Hardware, Metrics, Workloads, Foster‵s Methodology

Moore's Law vs. Walls: Speed, Power, Memory, ILP

Dynamic Power ~
Number of Transistors (N) x
Capacitance (C) x
Voltage² (V²) x Frequency (F)

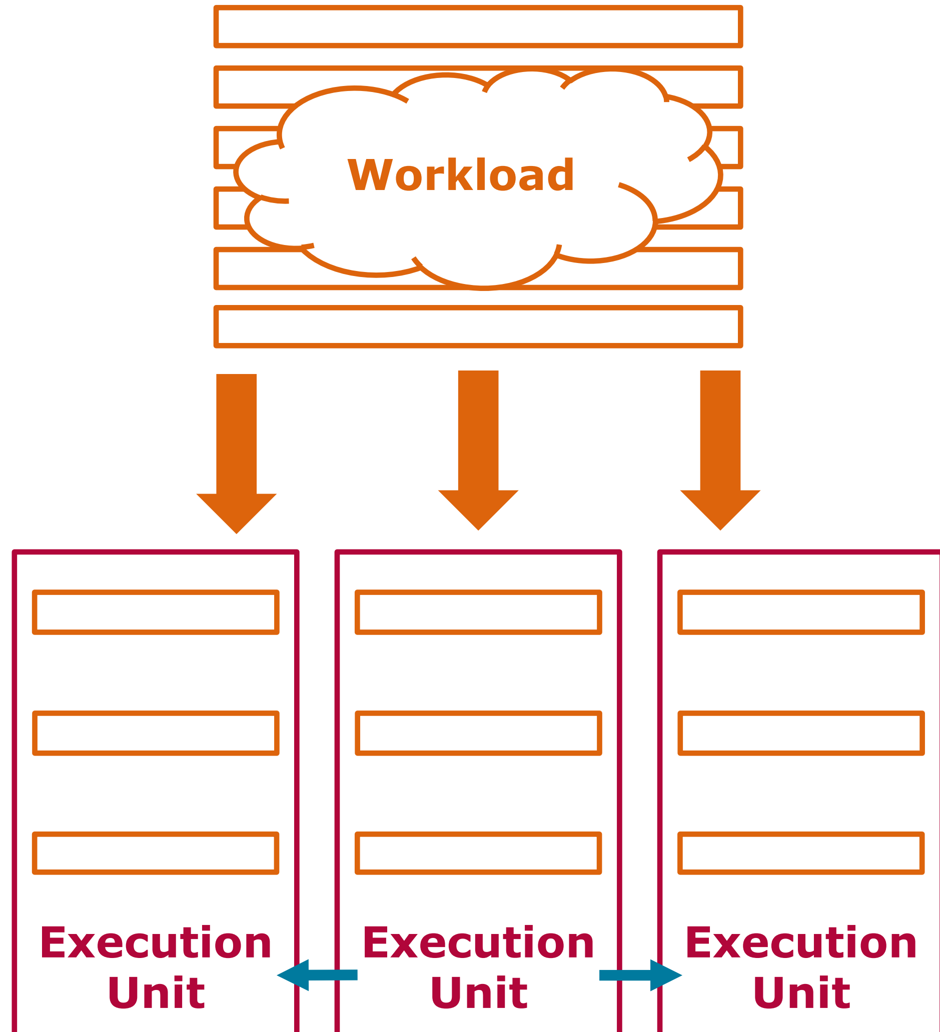




ParProg 2020 Introduction: Why Parallel?

Max Plauth

[Pfister1998]
Three Ways of Doing Things Faster



- Work Harder (execution capacity)
- Work Smarter (optimization)
- **Get Help (parallelization)**

: Workload

collection of operations that are executed to produce a desired result

~ *Program, Application*

: Execution Unit

facility that is capable of executing the operations of a workload

An Important Distinction

Concurrency

Capability of a machine to have multiple tasks in progress at any point in time

- Can be realized without parallel hardware

Parallelism

Capability of a machine to perform multiple tasks simultaneously

- Requires parallel hardware

Any parallel program is a concurrent program,
some concurrent programs cannot be executed correctly in parallel.

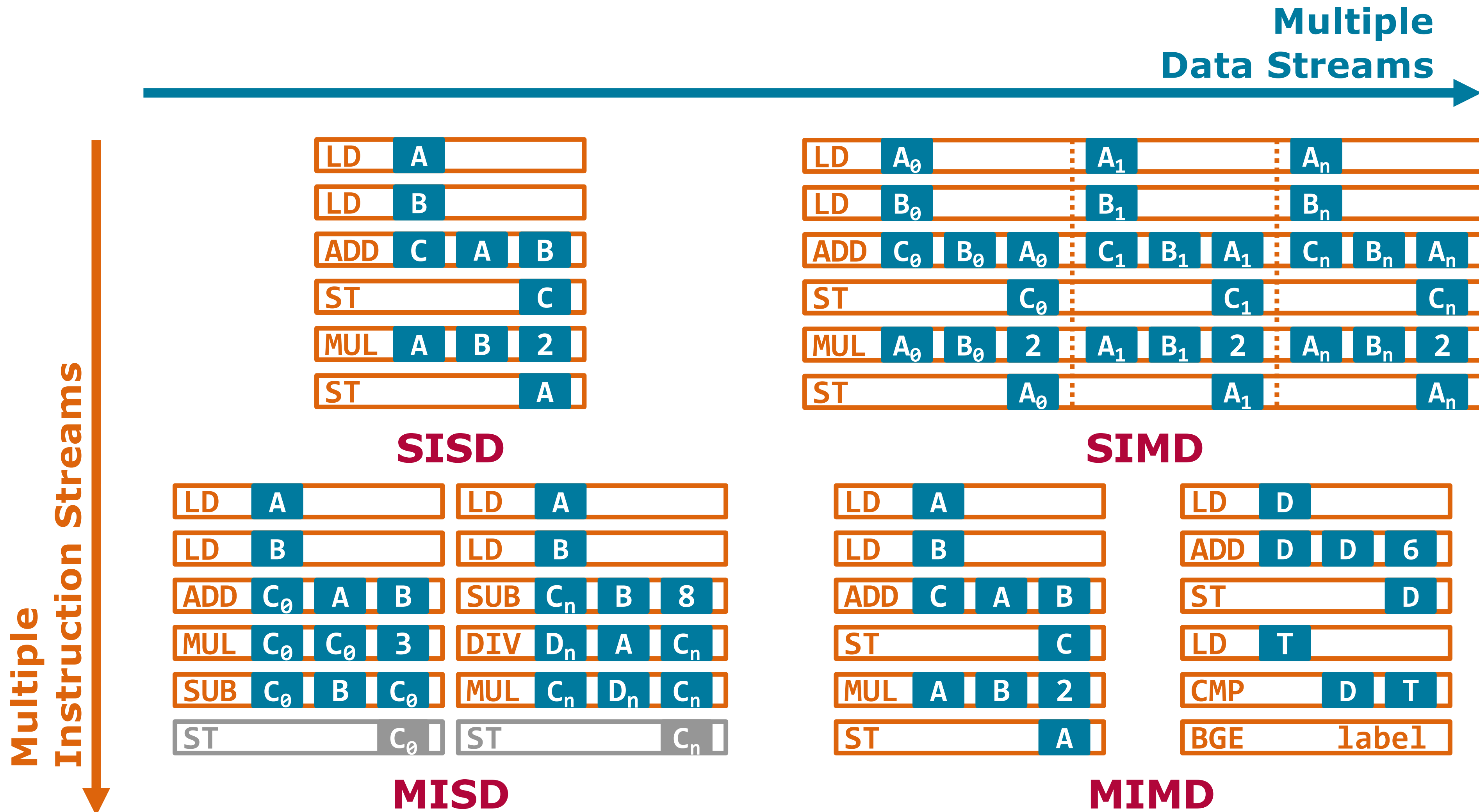
Distribution

Form of Parallelism, where tasks are performed by multiple communicating machines

Concurrency ⊃ Parallelism ⊃ Distribution

sometimes Concurrency \ Parallelism called "Concurrency"

Hardware Taxonomy [Flynn1966]



HPI Hasso Plattner Institut

MIMD Hardware Taxonomy

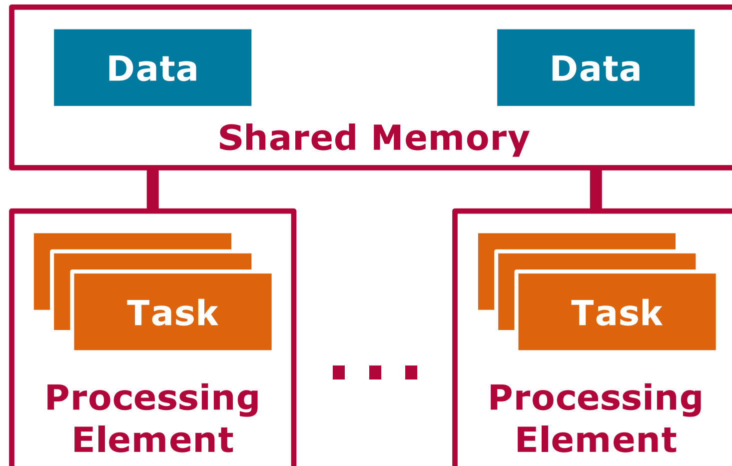
MIMD

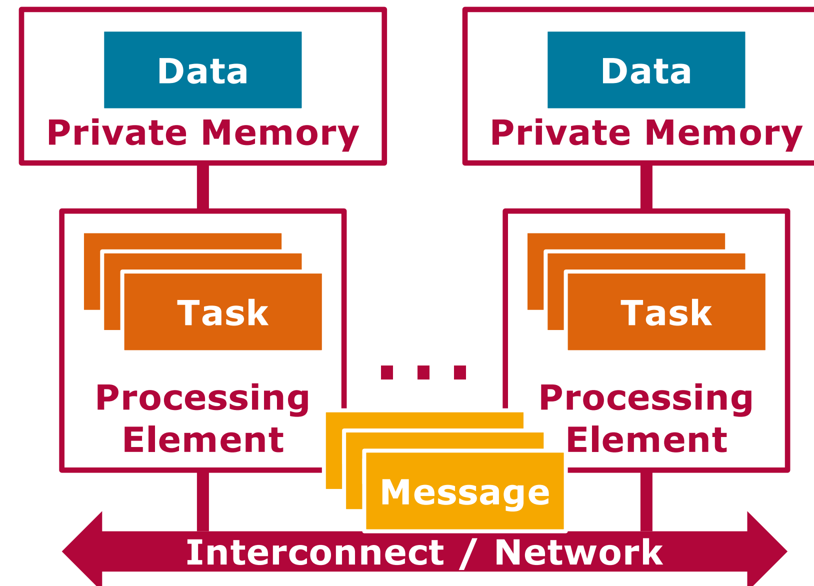
SM-MIMD
(Shared Memory)

Processing elements can directly access a **common address space**

DM-MIMD
(Distributed Memory)

Processing elements can access their **private address spaces** and **exchange messages**





SM-MIMD Hardware

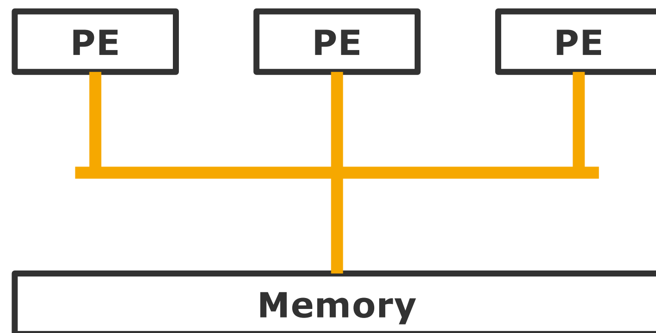
MIMD

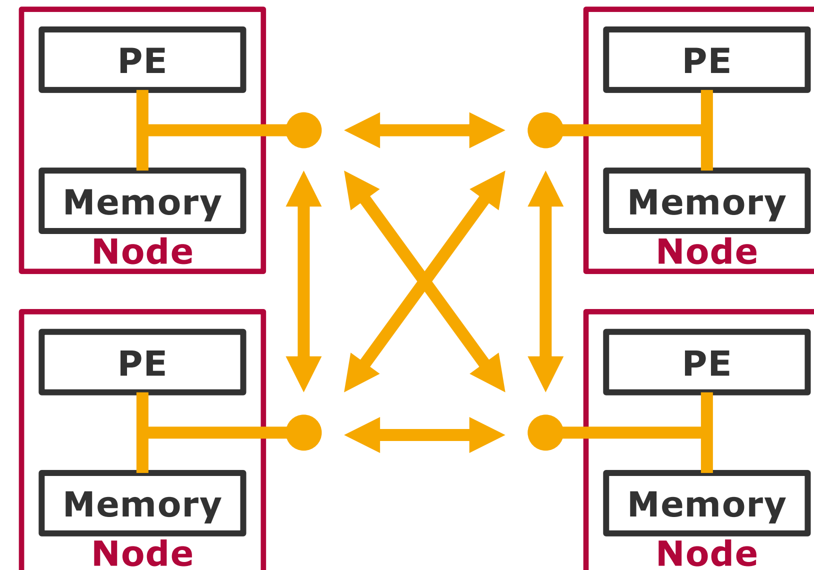
SM-MIMD
(Shared Memory)

DM-MIMD
(Distributed Memory)

UMA
(Uniform Memory Access)

NUMA
(Non-Uniform Memory Access)





Recap
Optimization Goals

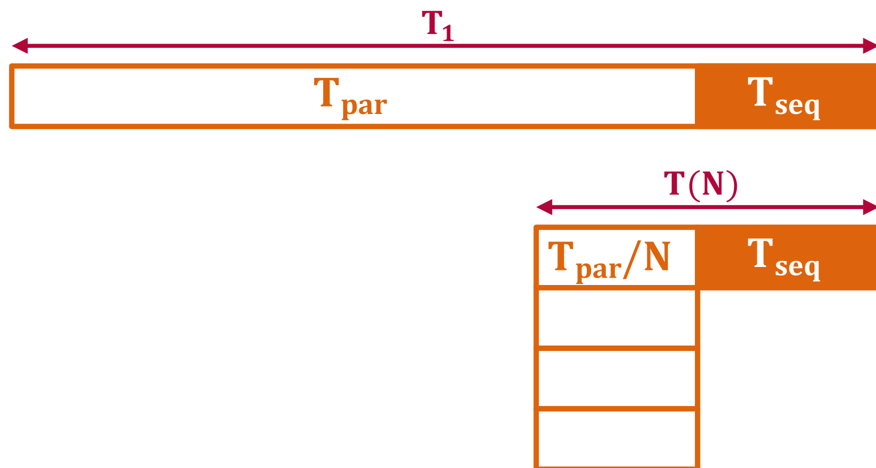
- Decrease **Latency** – process a single workload faster (= **speedup**)
- Increase **Throughput** – process more workloads in the same time
- ➢ Both are **Performance** metrics

- **Scalability**: make best use of additional resources
  - **Scale Up**: Utilize additional resources on a machine
  - **Scale Out**: Utilize resources on additional machines
- **Cost/Energy Efficiency**:
  - minimize cost/energy requirements for given performance objectives
  - *alternatively: maximize performance for given cost/energy budget*
- **Utilization**: minimize idle time (=waste) of available resources
- **Precision-Tradeoffs**: trade performance for precision of results

Anatomy of a Workload

The **longest task** puts a **lower bound on** the shortest **execution time.**

Modeling discrete tasks is impractical → simplified **continuous model.**



$$T(N) = \frac{T_{par}}{N} + T_{seq}$$

Replace absolute times by **parallelizable fraction** P:

$$T_{par} = T_1 \cdot P$$
$$T_{seq} = T_1 \cdot (1 - P)$$

$$T(N) = T_1 \cdot \left(\frac{P}{N} + (1 - P)\right)$$

[Amdahl1967] Amdahl‘s Law

Amdahl's Law derives the speedup $\mathbf{s_{Amdahl}(N)}$ for a parallelization degree $\mathbf{N}$

$$\mathbf{s_{Amdahl}(N)} = \frac{T_1}{T(N)} = \frac{T_1}{T_1 \cdot \left(\frac{P}{N} + (1 - P)\right)} = \frac{\mathbf{1}}{\frac{\mathbf{P}}{\mathbf{N}} + (\mathbf{1} - \mathbf{P})}$$

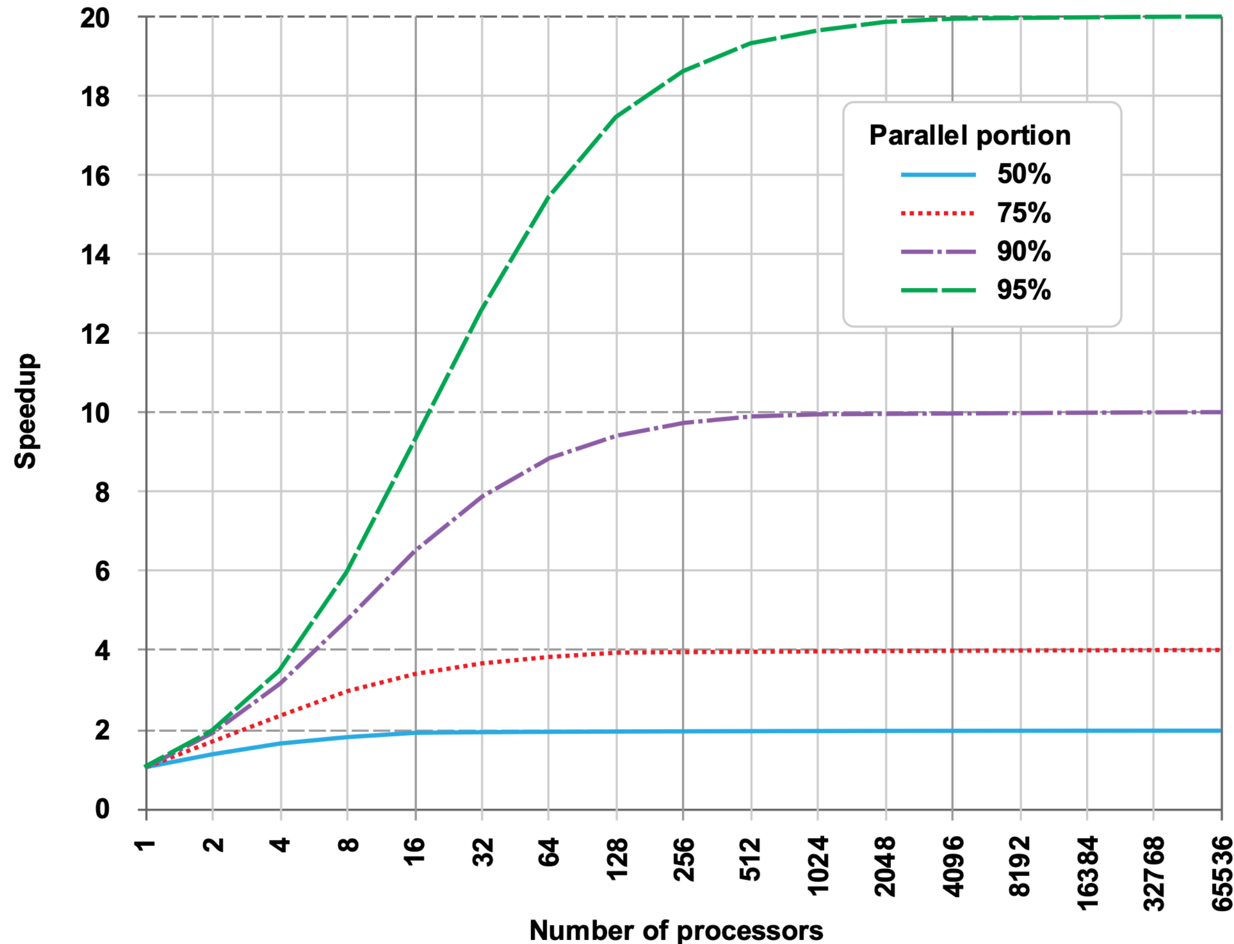
Even for **arbitrarily large** $\mathbf{N}$, the speedup converges to a **fixed limit**

$$\lim_{N \to \infty} s_{Amdahl}(N) = \frac{\mathbf{1}}{\mathbf{1} - \mathbf{P}}$$

For getting reasonable speedup out of 1000 processors, the sequential part must be substantially below 0.1%

[Amdahl1967] Amdahl's Law

Hasso Plattner Institut



By Daniels220 at English Wikipedia, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=6678551

*Regardless of processor count, **90% parallelizable** code allows not more than a **speedup by factor 10**.*

- Parallelism requires highly parallelizable workloads to achieve a speedup
- What is the sense in large parallel machines?

Amdahl's law assumes a simple speedup scenario!

- isolated execution of a **single workload**
- **fixed workload size**

[Gustafson1988] Gustafson-Barsis' Law

Consider a **scaled speedup scenario**, allowing a variable workload size $\mathbf{w}$.

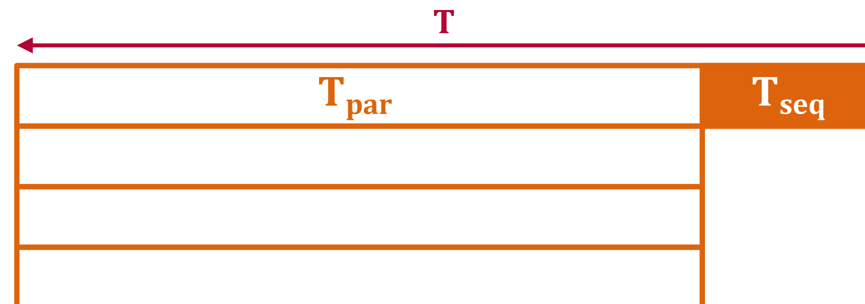
Amdahl ~ *What is the shortest execution time for a given workload?*

Gustafson-Barsis ~ *What is the largest workload for a given execution time?*

Determine the scaled speedup $\mathbf{s}_{\mathbf{Gustavson}}(\mathbf{N})$ through the increase in workload size $\mathbf{w}(\mathbf{N})$ over the fixed execution time $\mathbf{T}$

$$\mathbf{s}_{\mathbf{Gustafson}}(\mathbf{N}) = \mathbf{P} \cdot \boldsymbol{N} + (\mathbf{1} - \boldsymbol{P})$$

$$\mathbf{w}_1 \sim \mathbf{T}_{\mathbf{par}} + \mathbf{T}_{\mathbf{seq}}$$



$$\mathbf{w}(\mathbf{N}) \sim \mathbf{N} \cdot \mathbf{T}_{\mathbf{par}} + \mathbf{T}_{\mathbf{seq}}$$

[Karp1990] Karp-Flatt-Metric

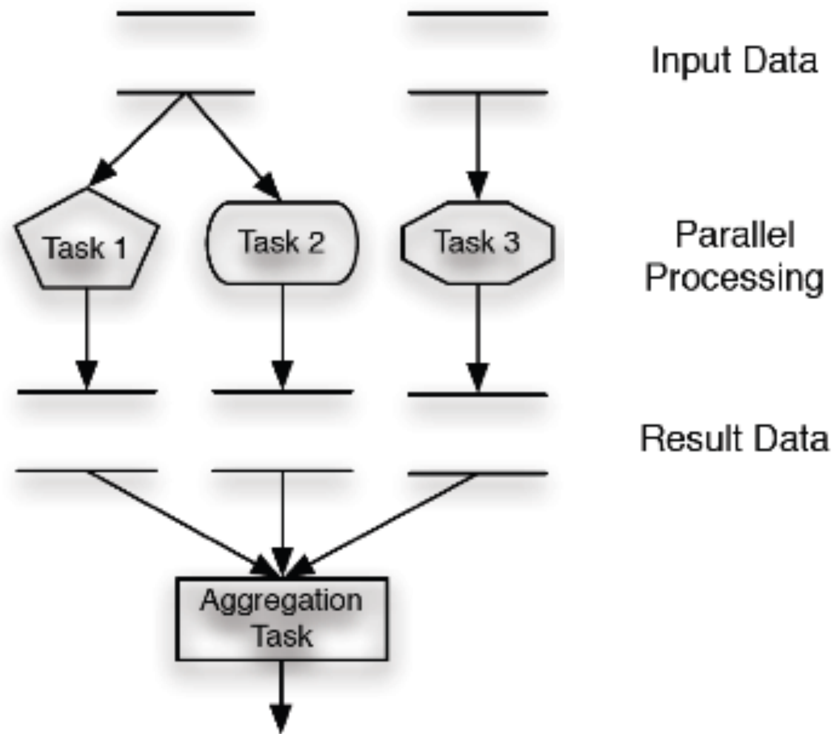
Parallel fraction $\mathbf{P}$ is a hypothetical parameter and not easily deduced from a given workload.

- Karp-Flatt-Metric determines sequential fraction $\mathbf{Q = 1 - P}$ empirically
  1. Measure baseline execution time $\mathbf{T_1}$
     by executing workload on a single execution unit
  2. Measure parallelized execution time $\mathbf{T(N)}$
     by executing workload on $\mathbf{N}$ execution units
  3. Determine speedup $\mathbf{s(N)} = {}^{\mathbf{T_1}}/_{\mathbf{T(N)}}$
  4. Calculate Karp-Flatt-Metric

$$\mathbf{Q(N)} = \frac{\frac{\mathbf{1}}{\mathbf{s(N)}} - \frac{\mathbf{1}}{\mathbf{N}}}{\mathbf{1} - \frac{\mathbf{1}}{\mathbf{N}}}$$

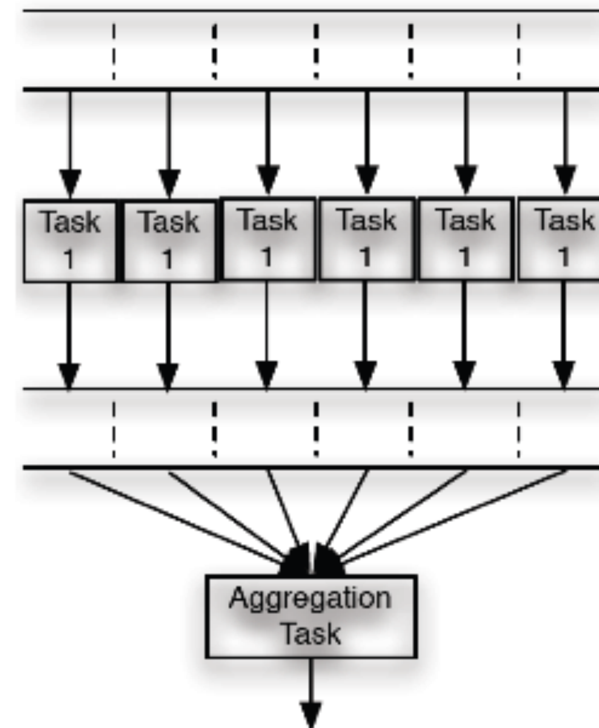
Workloads

"task-level parallelism"



- Different tasks being performed at the same time
- Might originate from the same or different programs

"data-level parallelism"



- Parallel execution of the same task on disjoint data sets

Designing Parallel Algorithms [Foster]

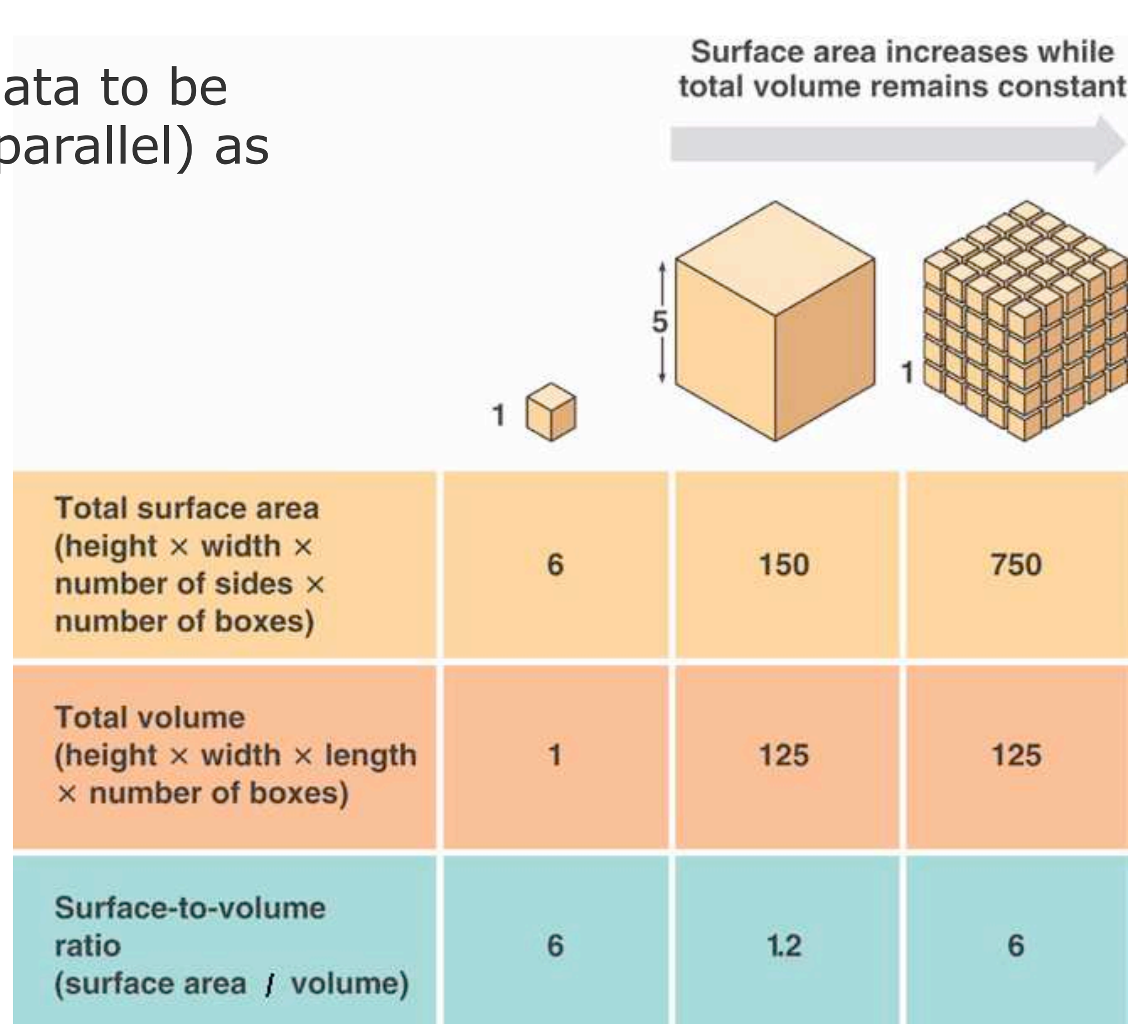
- A) Search for concurrency and scalability
  - **Partitioning**
    Decompose computation and data into the *smallest possible* tasks
  - **Communication**
    Define necessary coordination of task execution

- B) Search for locality and other performance-related issues
  - **Agglomeration**
    Consider performance and implementation costs
  - **Mapping**
    Maximize execution unit utilization, minimize communication

- Might require backtracking or parallel investigation of steps

Surface-To-Volume Effect [Foster, Breshears]

Visualize the data to be processed (in parallel) as sliced 3D cube



[nicerweb.com]

HPI Hasso Plattner Institut
Digital Engineering · Universität Potsdam

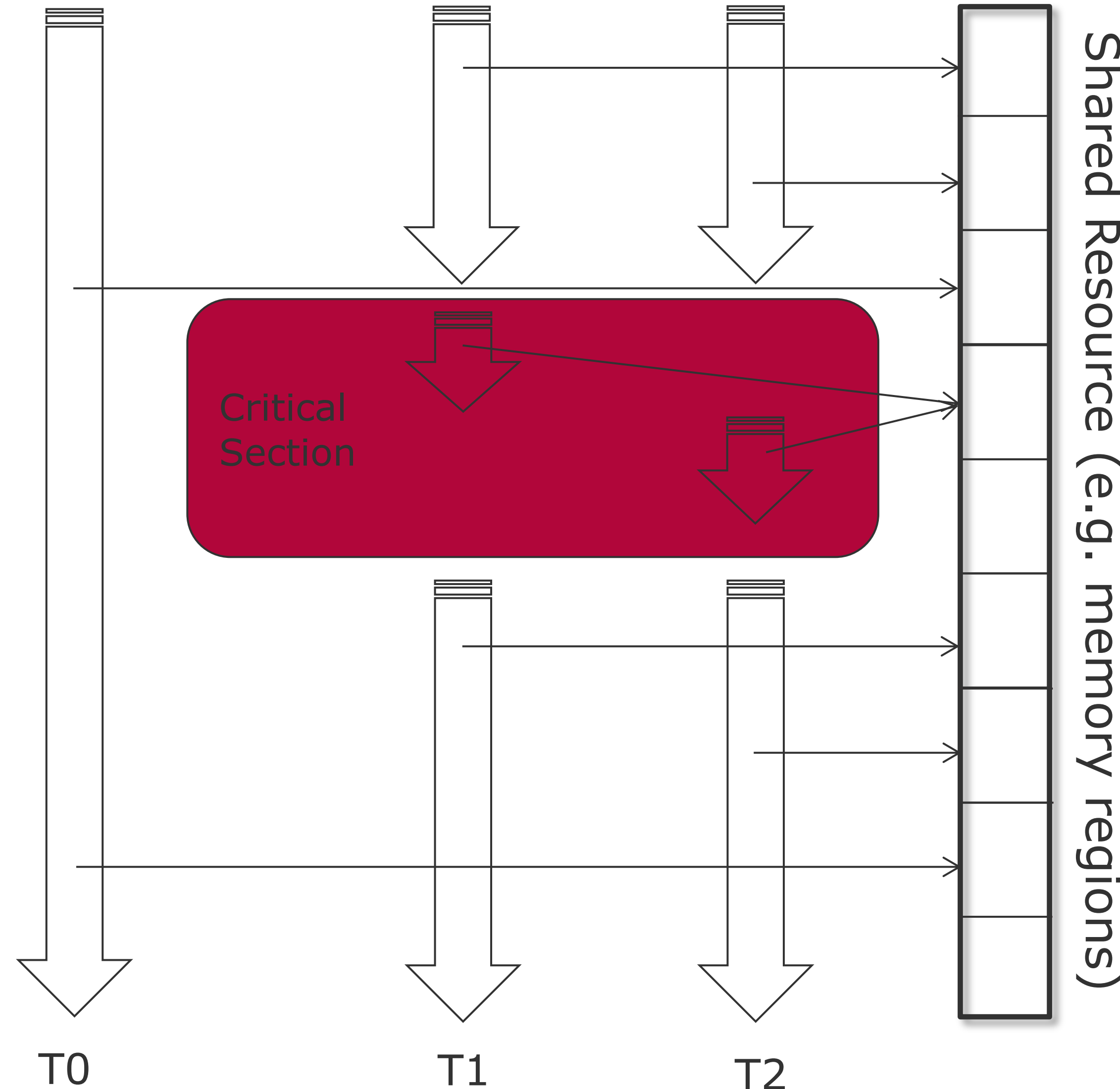
B1: Shared Memory Systems (Concurrency & Synchronization)

Critical Section

- **Mutual Exclusion** demand: Only one task at a time is allowed into its critical section, among all tasks that have critical sections for the same resource.

- **Progress** demand: If no other task is in the critical section, the decision for entering should not be postponed indefinitely. Only tasks that wait for entering the critical section are allowed to participate in decisions.

- **Bounded Waiting** demand: It must not be possible for a task requiring access to a critical section to be delayed indefinitely by other threads entering the section (**starvation problem**)



Cooperating Sequential Processes [Dijkstra1965] Solution: Dekker got it!

- Solution: Dekker's algorithm, attributed by Dijkstra
  - Combination of approach #4 and a variable `turn`, which realizes mutual blocking avoidance through prioritization
  - Idea: Spin for section entry only if it is your turn

```
"begin integer c1, c2, turn;
    c1:= 1; c2:= 1; turn:= 1;
    parbegin
    process 1: begin A1: c1:= 0;
                    L1: if c2 = 0 then
                            begin if turn = 1 then goto L1;
                                  c1:= 1;
                              B1: if turn = 2 then goto B1;
                                  goto A1
                            end;
                    critical section 1;
                    turn:= 2; c1:= 1;
                    remainder of cycle 1; goto A1
               end;
    process 2: begin A2: c2:= 0;
                    L2: if c1 = 0 then
                            begin if turn = 2 then goto L2;
                                  c2:= 1;
                              B2: if turn = 1 then goto B2;
                                  goto A2
                            end;
                    critical section 2;
                    turn:= 1; c2:= 1;
                    remainder of cycle 2; goto A2
               end
    parend
end" .
```

Test-and-Set Instructions

- **Test-and-set** processor instruction, wrapped by the operating system or compiler
  - Write to a memory location and return its old value as atomic step
  - Also known as **compare-and-swap (CAS)** or **read-modify-write**
- Idea: Spin in writing 1 to a memory cell, until the old value was 0
  - Between writing and test, no other operation can modify the value
- Busy waiting for acquiring a **(spin) lock**
- Efficient especially for short waiting periods
- For long periods try to *deactivate* your processor between loops.

```
function Lock(boolean *lock) {
    while (test_and_set (lock))
      ;
}

#define LOCKED 1
 int TestAndSet(int* lockPtr) {
     int oldValue;
     oldValue = SwapAtomic(lockPtr, LOCKED);
     return oldValue == LOCKED;
 }
```

Coroutines

```
var q := new queue
coroutine produce
    loop
        while q is not full
            create some new items
            add the items to q
        yield to consume
coroutine consume
    loop
        while q is not empty
            remove some items from q
            use the items
        yield to produce
```

```
def generator():
    for i in range(5):
        yield i * 2

for item in generator():
    print(item)
```

Other High-Level Primitives

- Today: Multitude of high-level synchronization primitives
- **Spinlock**
  - Perform busy waiting, lowest overhead for short locks

- **Reader / Writer Lock**
  - Special case of mutual exclusion through semaphores
  - Multiple „Reader" tasks can enter the critical section at the same time, but „Writer" task should gain exclusive access
  - Different optimizations possible:
    minimum reader delay, minimum writer delay, throughput, …

Coffman Conditions [Coffman1970]

- *1970. E.G. Coffman and A. Shoshani.*
  Sequencing tasks in multiprocess systems to avoid deadlocks.
  - All conditions must be fulfilled to allow a deadlock to happen
  - **Mutual exclusion condition** - Individual resources are available or held by no more than one task at a time
  - **Hold and wait condition** – Task already holding resources may attempt to hold new resources
  - **No preemption condition** – Once a task holds a resource, it must voluntarily release it on its own
  - **Circular wait condition** – Possible for a task to wait for a resource held by the next thread in the chain
- Avoiding circular wait turned out to be the easiest solution for deadlock avoidance
- Avoiding mutual exclusion leads to **non-blocking synchronization**
  - These algorithms no longer have a critical section

B2: Programming Models

POSIX Threads (Pthreads)

pthread

```
_create
_self
_cancel
_exit
_join
_kill
_attr_setstacksize
_attr_setstackaddr
_mutex_lock
_mutex_trylock
_mutex_unlock
_cond_signal
_cond_timedwait
_cond_wait
_rwlock_rdlock
_rwlock_unlock
_rwlock_wrlock
_barrier_wait
_key_create
_setspecific
[...]
```

C++11

- C++11 specification added support concurrency constructs
- Allows asynchronous tasks with *std::async* or *std::thread*
- Relies on *Callable* instance (functions, member functions, lambdas, ...)

```
#include <future>
#include <iostream>

void write_message(std::string const& message) {
  std::cout<<message;
}

int main() {
  auto f = std::async(write_message,
    "hello world from std::async\n");
  write_message("hello world from main\n");
  f.wait();
}
```

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

void write_message(std::string const& message) {
    std::cout<<message;
}

int main() {
   std::thread t(write_message,
     "hello world from std::thread\n");
   write_message("hello world from main\n");
   t.join();
}
```

https://en.cppreference.com/w/cpp/thread

C++11: Futures & Promises

- Launch policy for the *async call* can be specified
  - Deferred or immediate launch of the activity
- As for all asynchronous task types, a **future** is returned
  - Object representing the (future) result of an asynchronous operation, allows to block on the result reading
  - Original concept by Baker and Hewitt [1977]
- A **promise** object can store a value that is later acquired via a future object
  - Separate concept since futures are only readable
  - Can provide a dummy barrier implementation
- Future == Handle, Promise == Value
- Promise and future as concept also available in Java 5, Smalltalk, Scheme, CORBA, …

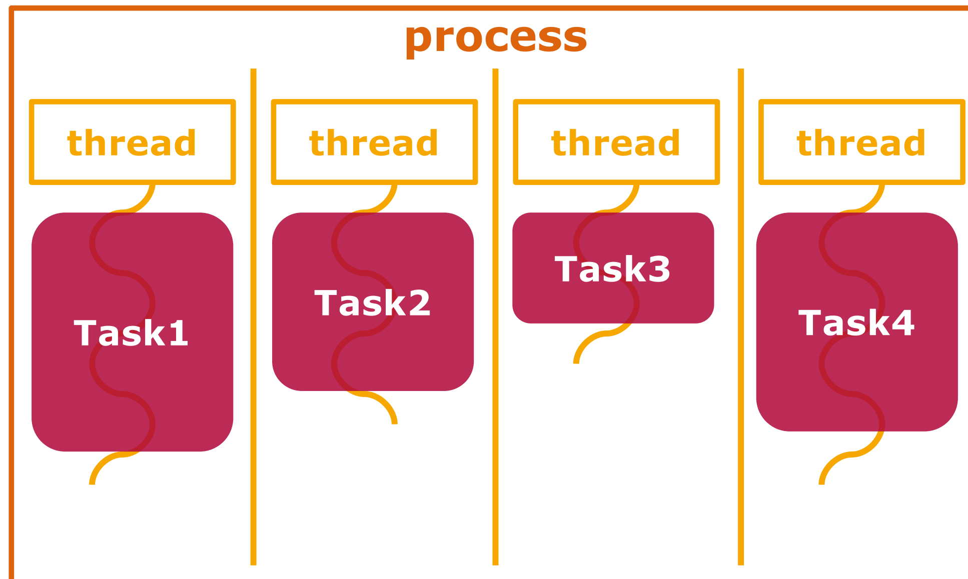
Explicit vs Implicit Threading

Thread generation, synchronization, data access:

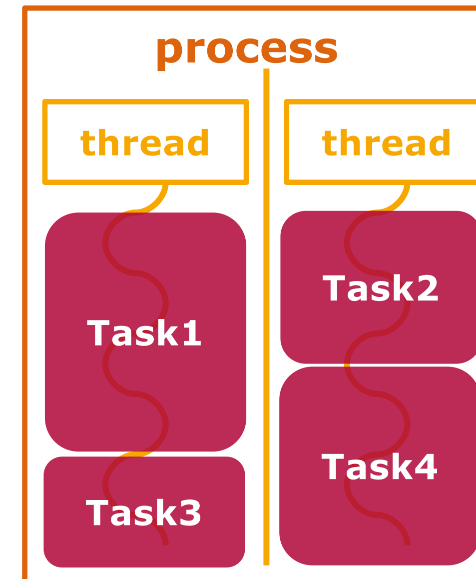
Explicit, as part of some sequential code (OS API, C++/Java/Python Threads)

Implicit, based on a framework (OpenMP, OpenCL, Intel TBB, ...)

Explicit Threading



Implicit Threading



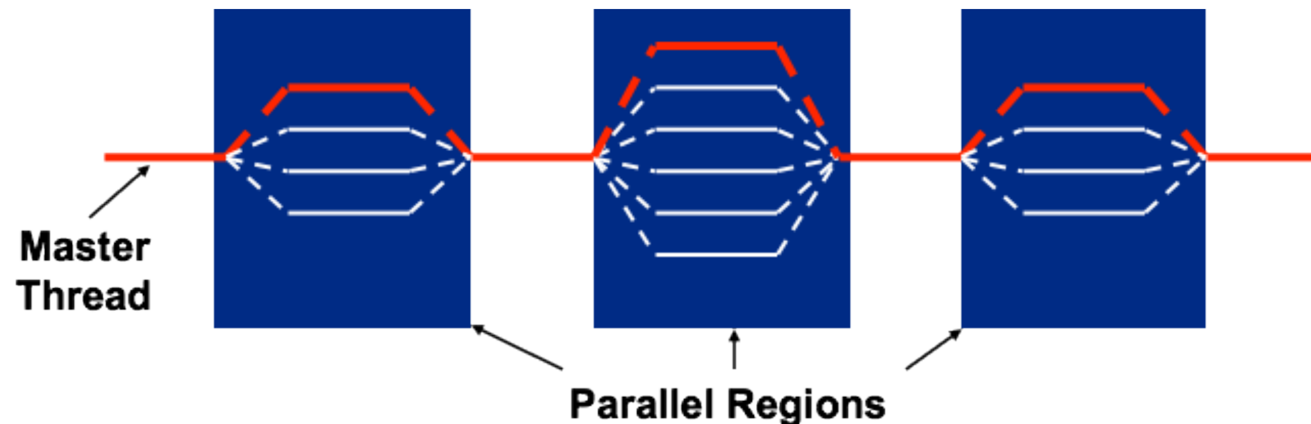
OpenMP

Specification for C/C++ and Fortran language extension

- Portable shared memory thread programming
- High-level abstraction of task- and loop parallelism
- Derived from compiler-directed parallelization of serial language code (HPF), with support for incremental change of legacy code
- **Multiple implementations** exist

Programming model: Fork-Join-Parallelism

- Master thread spawns group of threads for limited code region



OpenMP Loop Parallelization Scheduling

- `schedule (static, [chunk])`:
  - Contiguous ranges of iterations (chunks) are assigned to the threads
  - Low overhead, round robin assignment to free threads
  - Static scheduling for predictable and similar work per iteration
  - Increasing chunk size reduces overhead, improves cache hit rate
  - Decreasing chunk size allows finer balancing of work load
  - Default is one chunk per thread
- `schedule (guided, [chunk])`
  - Dynamic schedule, shrinking ranges per step
  - Starts with large block, until minimum chunk size is reached
  - Good for computations with increasing iteration length (e.g. prime sieves)
- `schedule (dynamic, [chunk])`
  - Idling threads grab iteration (or chunk) as available (work-stealing)
  - Higher overhead, but good for unbalanced/unpredicable iteration work load

Work Stealing

Blumofe, Leiserson, Charles:
Scheduling Multithreaded Computations by Work Stealing (FOCS 1994)

Problem of scheduling scalable multithreading problems on SMP

Work sharing: When processors create new work,
the scheduler migrates threads for balanced utilization

Work stealing: Underutilized core takes work from other processor,
leads to less thread migrations

- Goes back to work stealing research in Multilisp (1984)
- Supported in OpenMP implementations, TPL, TBB, Java, Cilk, …

Randomized work stealing: Lock-free ready dequeue per processor

- Task are inserted at the bottom, local work is taken from the bottom
- If no ready task is available, the core steals the top-most one from another randomly chosen core; added at the bottom

- Ready tasks are executed, or wait for a processor becoming free

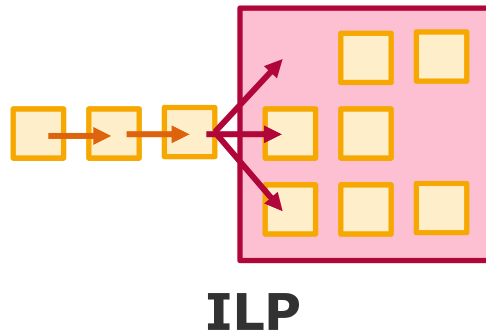
Large body of research about other work stealing variations

Hasso Plattner Institut
Digital Engineering · Universität Potsdam

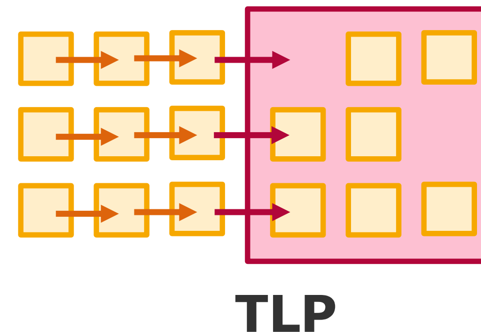
B3: Hardware

Shared-Memory Hardware
Exploiting Instruction Level Parallelism

- ILP arises naturally within a workload
  - Programmers think in terms of a single instruction sequence
- TLP is explicitly encoded within a workload
  - Programmers designates parallel operations using multiple instruction sequences



ILP



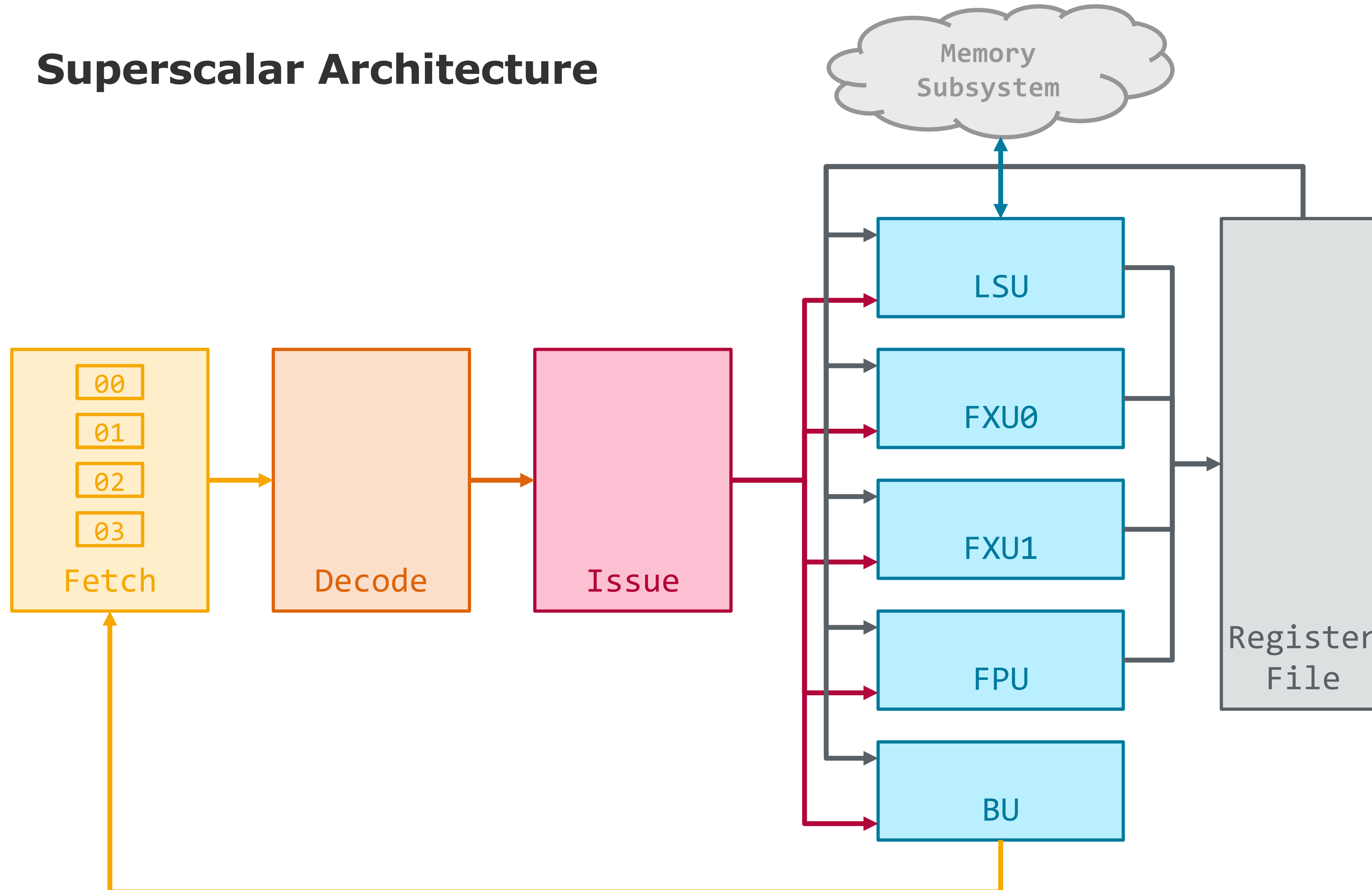
TLP

Why consider ILP in a parallel programming lecture?
Knowledge of common ILP mechanisms and assumptions enables performance optimization on single-thread granularity!

HPI Hasso Plattner Institut

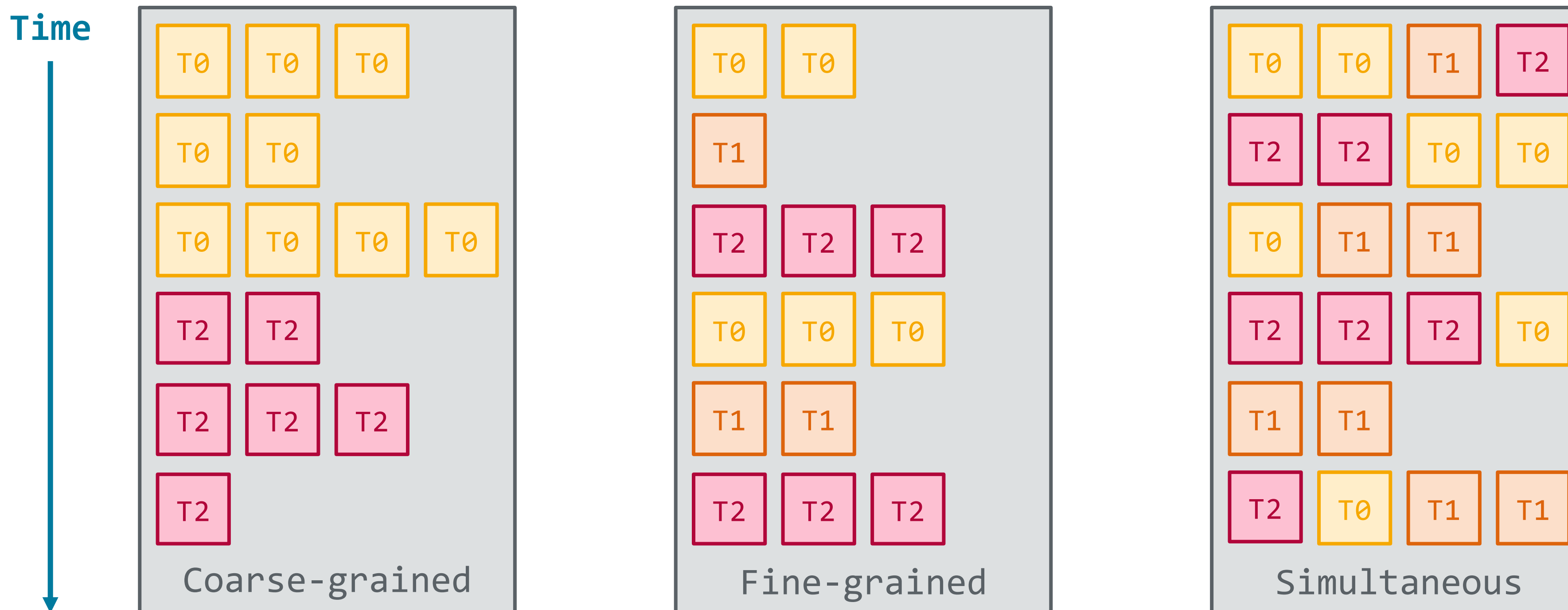
Hasso Plattner Institut

Superscalar Architecture



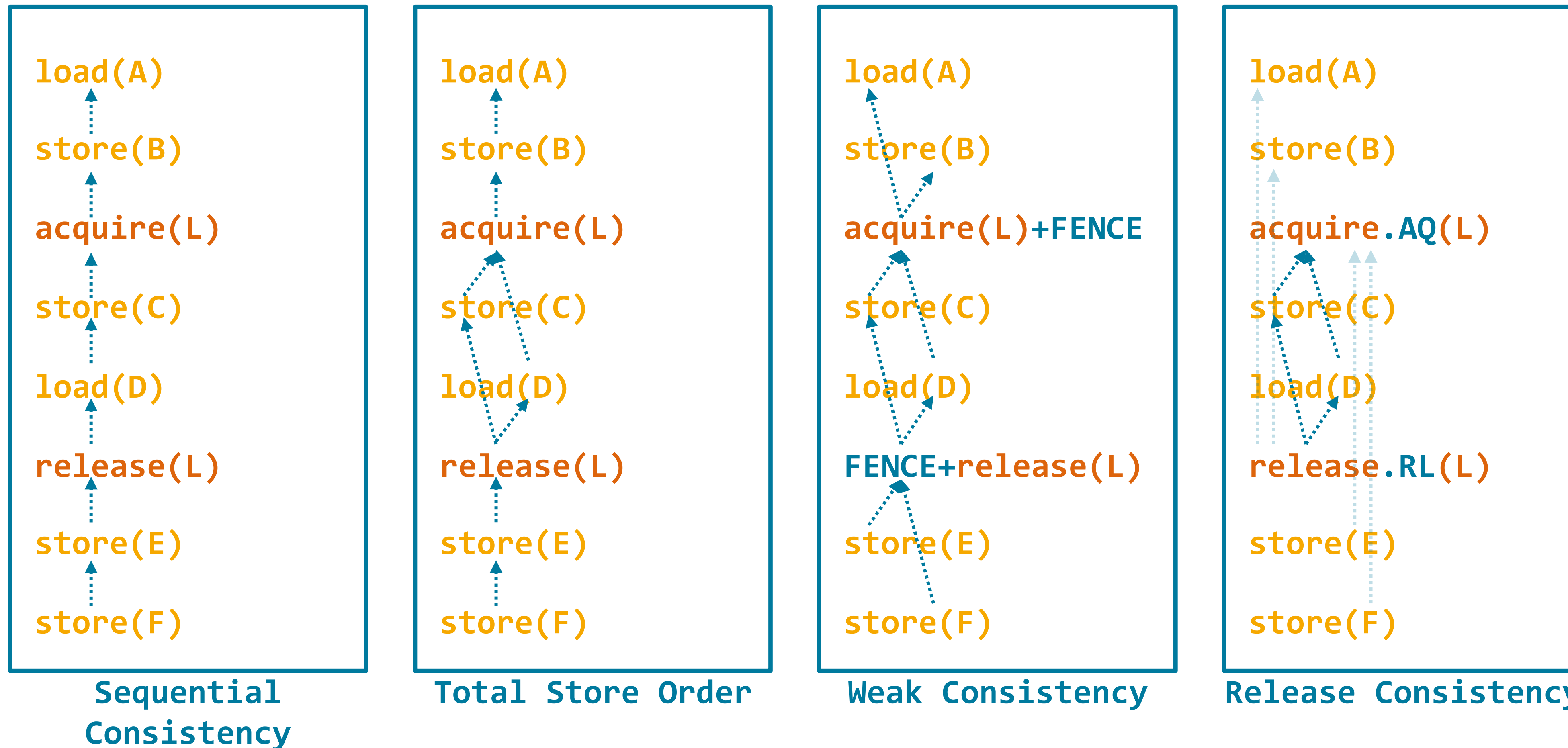
Single-Core Multithreading

- Threads are the smallest units of parallelism under programmers' explicit control
- There are different execution schemes for multiple threads on a single core:



Shared-Memory Hardware
Memory Consistency Models

Overview



Shared-Memory Hardware
Coherent Cache Hierarchy

HPI Hasso Plattner Institut

MSI Coherence Protocol

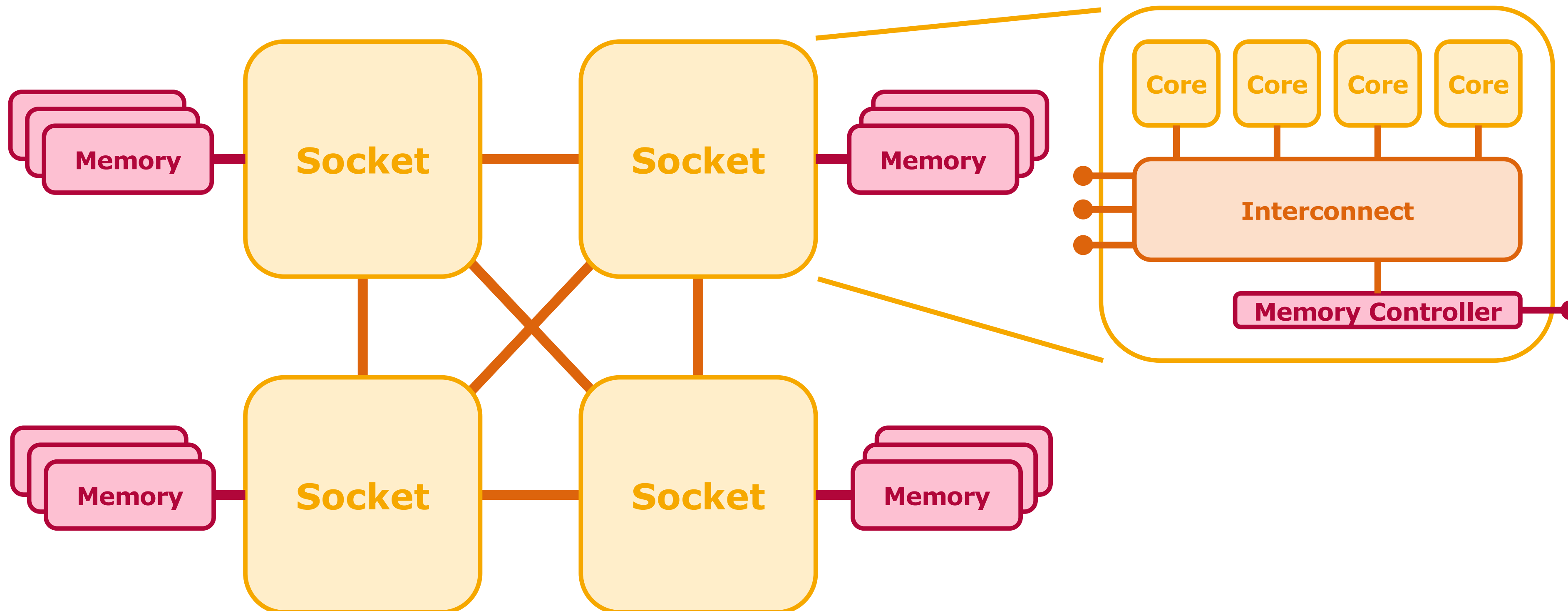
- MSI is a simple coherence protocol, based on a state machine
- Seen from a particular cache, each cache line is in one of three states:
  - Invalid: The cache line is not present in the cache, this cache may service neither **Load** nor **Store** operations
  - Shared: The cache line is present in this and probably other caches, this cache may service **Load** operations
  - Modified: The cache line is only present in this cache, this cache may service **Load** and **Store** operations

Hasso Plattner Institut
Digital Engineering · Universität Potsdam

B4: NUMA

Non-Uniform Memory Access Concept

- Part of the main memory is directly attached to a socket (**local memory**)
- Memory attached to a different socket can be accessed indirectly via the other socket‵s memory controller and interconnect (**remote memory**)
- Socket + local memory form a **NUMA node**



Non-Uniform Memory Access
Placement Decisions

Tradeoff:

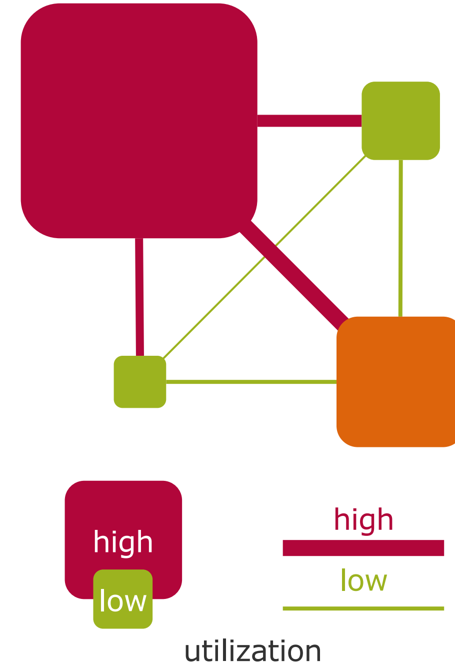
computational load balancing ◇ data locality

Thread Placement: Realized in the OS through an **Affinity Mask**

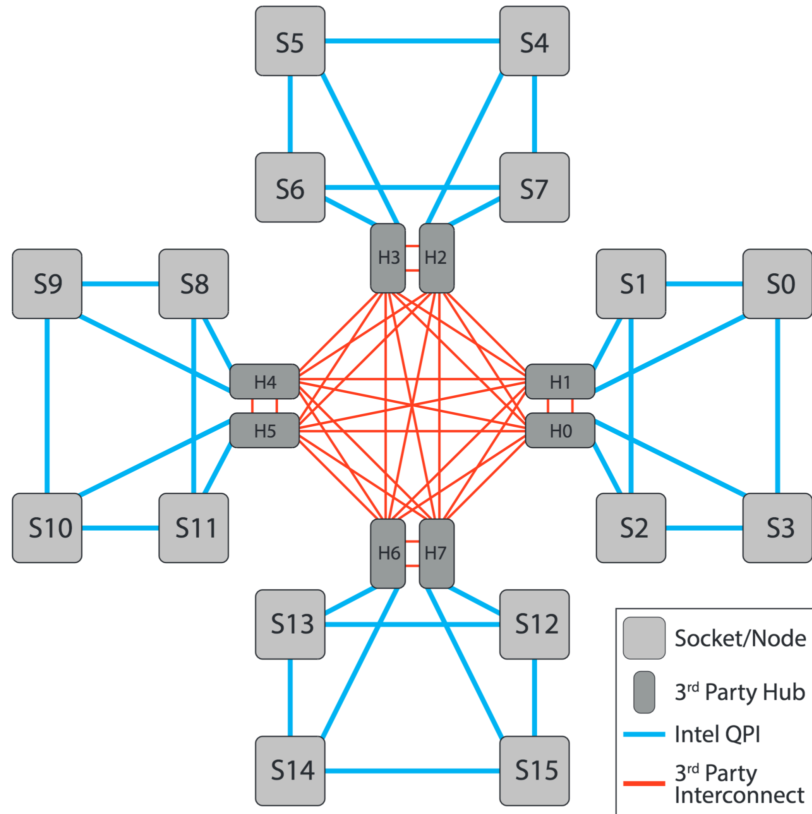
- **Pinning** (= only a single bit set)
- Affinity mask can be adjusted at runtime
- ➢ ***Computation follows data***

Data Placement: Realized in the OS on page granularity (4k, 64k, … 64GB)

- **Static**: Placement policies apply at allocation tome
  - First-touch · Allocate on fixed node(s) · Interleaving
- **Dynamic**: Pages can migrate at runtime
- ➢ ***Data follows computation***



Non-Uniform Memory Access
Topology Examples: SGI UV-300H



N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	10	16	19	16	50	50	50	50	50	50	50	50	50	50	50	50
2	16	10	16	19	50	50	50	50	50	50	50	50	50	50	50	50
3	19	16	10	16	50	50	50	50	50	50	50	50	50	50	50	50
4	16	19	16	10	50	50	50	50	50	50	50	50	50	50	50	50
5	50	50	50	50	10	16	19	16	50	50	50	50	50	50	50	50
6	50	50	50	50	16	10	16	19	50	50	50	50	50	50	50	50
7	50	50	50	50	19	16	10	16	50	50	50	50	50	50	50	50
8	50	50	50	50	16	19	16	10	50	50	50	50	50	50	50	50
9	50	50	50	50	50	50	50	50	10	16	19	16	50	50	50	50
10	50	50	50	50	50	50	50	50	16	10	16	19	50	50	50	50
11	50	50	50	50	50	50	50	50	19	16	10	16	50	50	50	50
12	50	50	50	50	50	50	50	50	16	19	16	10	50	50	50	50
13	50	50	50	50	50	50	50	50	50	50	50	50	10	16	19	16
14	50	50	50	50	50	50	50	50	50	50	50	50	16	10	16	19
15	50	50	50	50	50	50	50	50	50	50	50	50	19	16	10	16
16	50	50	50	50	50	50	50	50	50	50	50	50	16	19	16	10

How would you roll out a matrix multiplication workload on this system?

What tools / control mechanisms can you use?

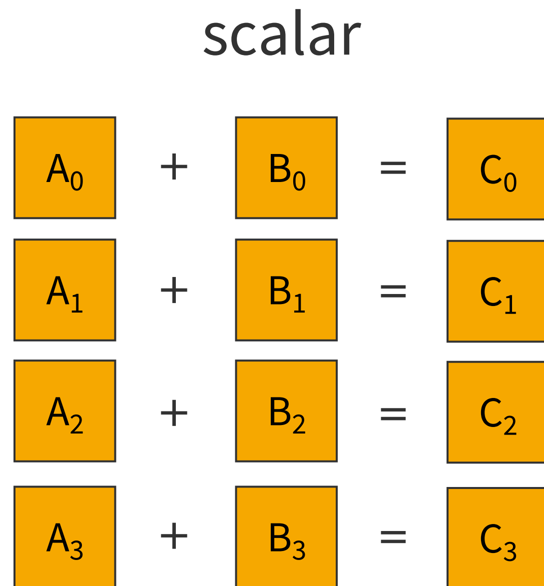
HPI Hasso Plattner Institut
Digital Engineering · Universität Potsdam

C1: SIMD

Scalar vs. SIMD

How many instructions are needed to add four numbers from memory?

scalar

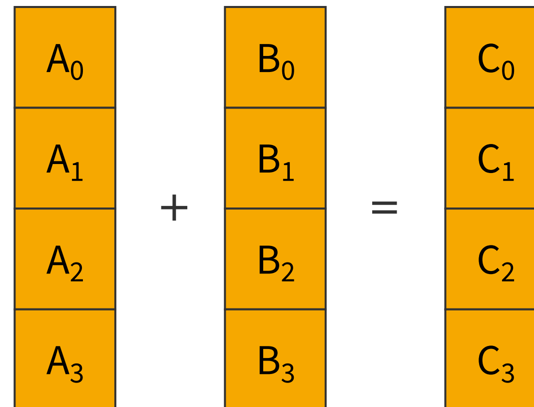


4 additions

8 loads

4 stores

4 element SIMD



1 addition

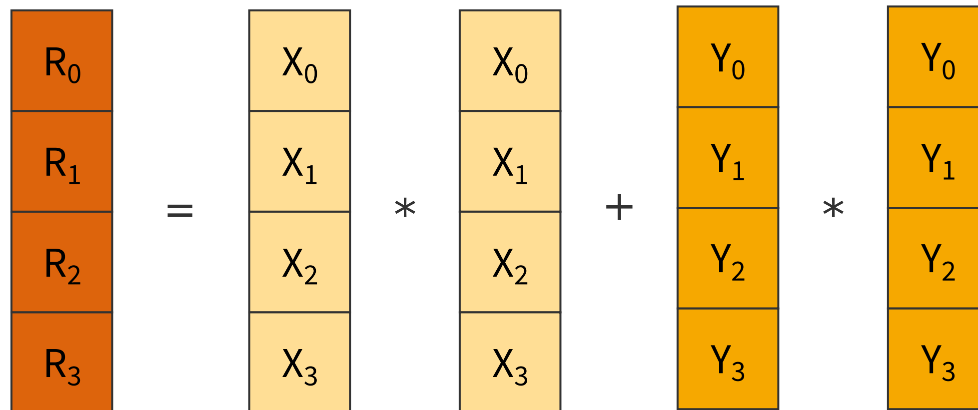
2 loads

1 store

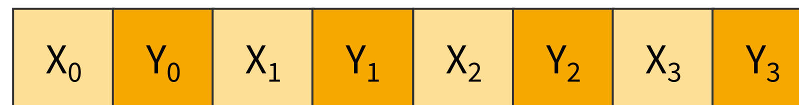
Vector Data Realignment and Permutation (1)

Sometimes memory is not correctly ordered for a certain tasks.

Example: Squared absolute of 2D points ($r^2 = p_x{}^2 + p_y{}^2$)

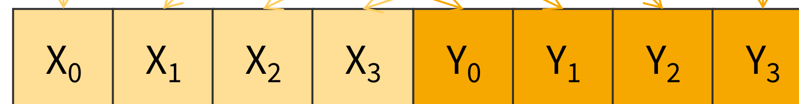


in memory:




```
struct point2d[];
```

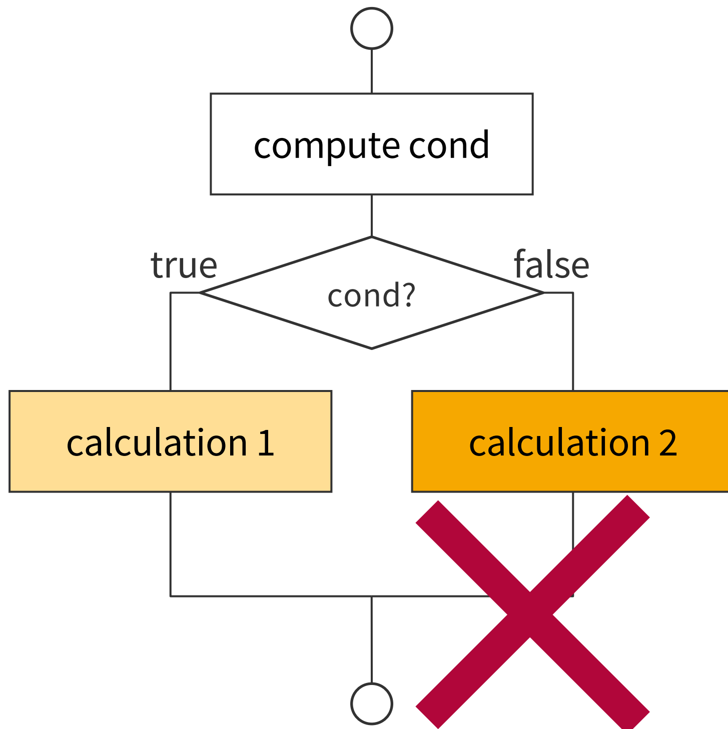
in registers:

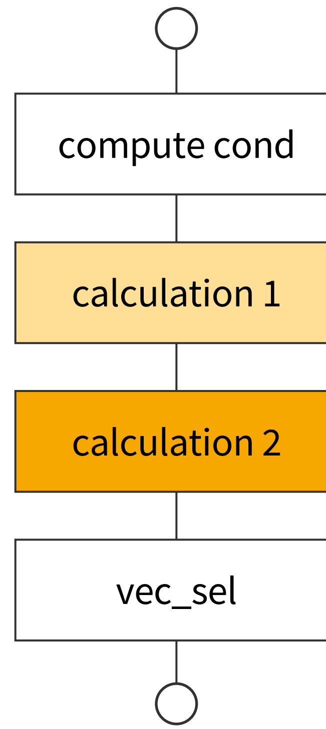


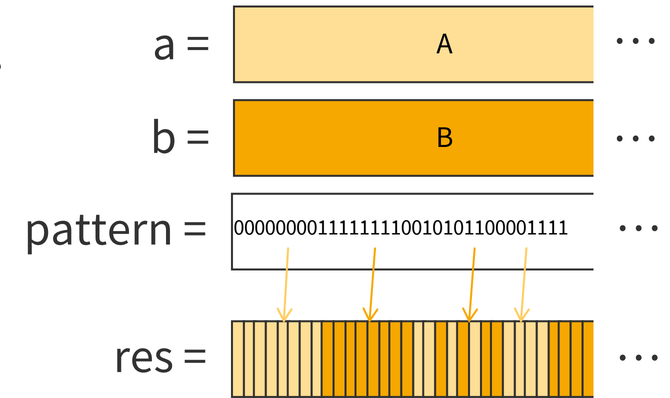
Conditional Programming (1)

There are **no branches** for element computation in AltiVec.

Instead compute both variants and then use **bit-wise select**.

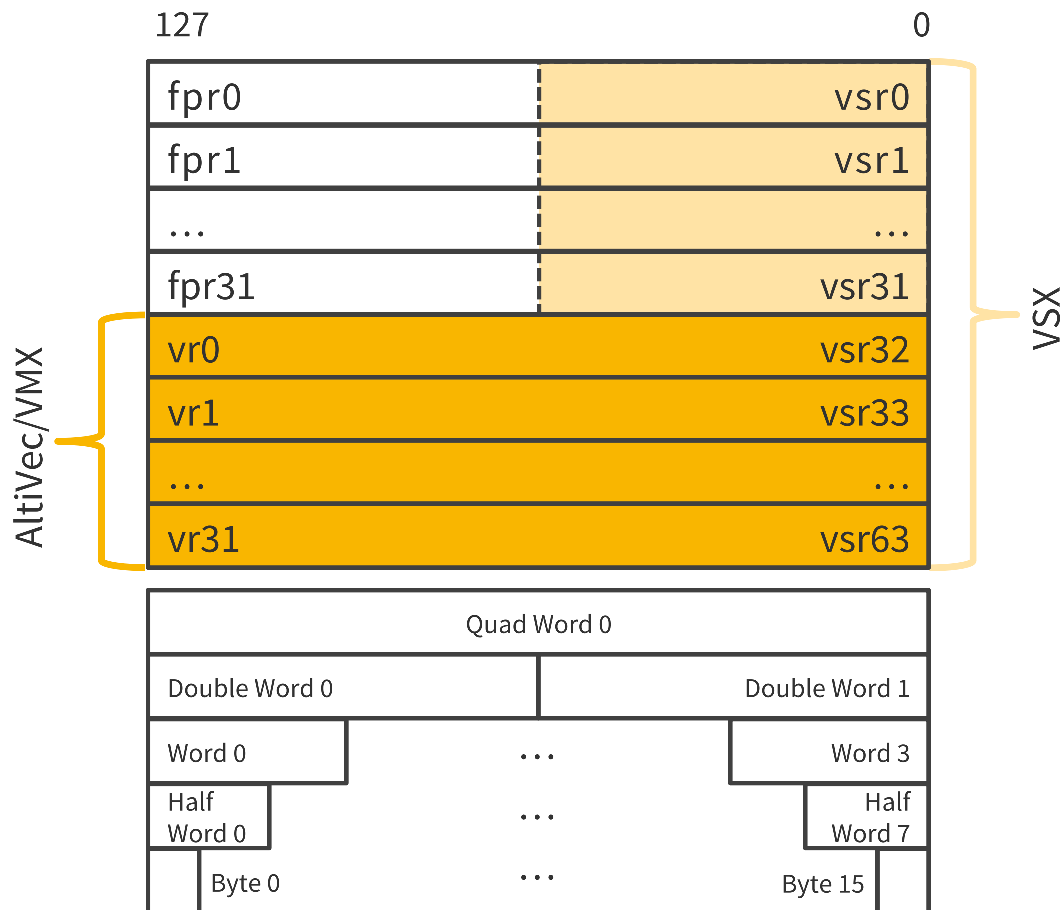






Architecture-Dependent Element Count in Vector Registers

ppc64



amd64

AVX-512 register scheme as extension from the AVX (YMM0-YMM15) and SSE (XMM0-XMM15) registers

511 256	255 128	127 0
ZMM0	YMM0	XMM0
ZMM1	YMM1	XMM1
ZMM2	YMM2	XMM2
ZMM3	YMM3	XMM3
ZMM4	YMM4	XMM4
ZMM5	YMM5	XMM5
ZMM6	YMM6	XMM6
ZMM7	YMM7	XMM7
ZMM8	YMM8	XMM8
ZMM9	YMM9	XMM9
ZMM10	YMM10	XMM10
ZMM11	YMM11	XMM11
ZMM12	YMM12	XMM12
ZMM13	YMM13	XMM13
ZMM14	YMM14	XMM14
ZMM15	YMM15	XMM15
ZMM16	YMM16	XMM16
ZMM17	YMM17	XMM17
ZMM18	YMM18	XMM18
ZMM19	YMM19	XMM19
ZMM20	YMM20	XMM20
ZMM21	YMM21	XMM21
ZMM22	YMM22	XMM22
ZMM23	YMM23	XMM23
ZMM24	YMM24	XMM24
ZMM25	YMM25	XMM25
ZMM26	YMM26	XMM26
ZMM27	YMM27	XMM27
ZMM28	YMM28	XMM28
ZMM29	YMM29	XMM29
ZMM30	YMM30	XMM30

Type	Contents
__m128	4 floats
__m128d	2 doubles
__m128i	integers (8-128bit)
__m256	8 floats
__m256d	4 doubles
__m256i	integers (8-128bit)
__m512	...

What loops can be vectorized

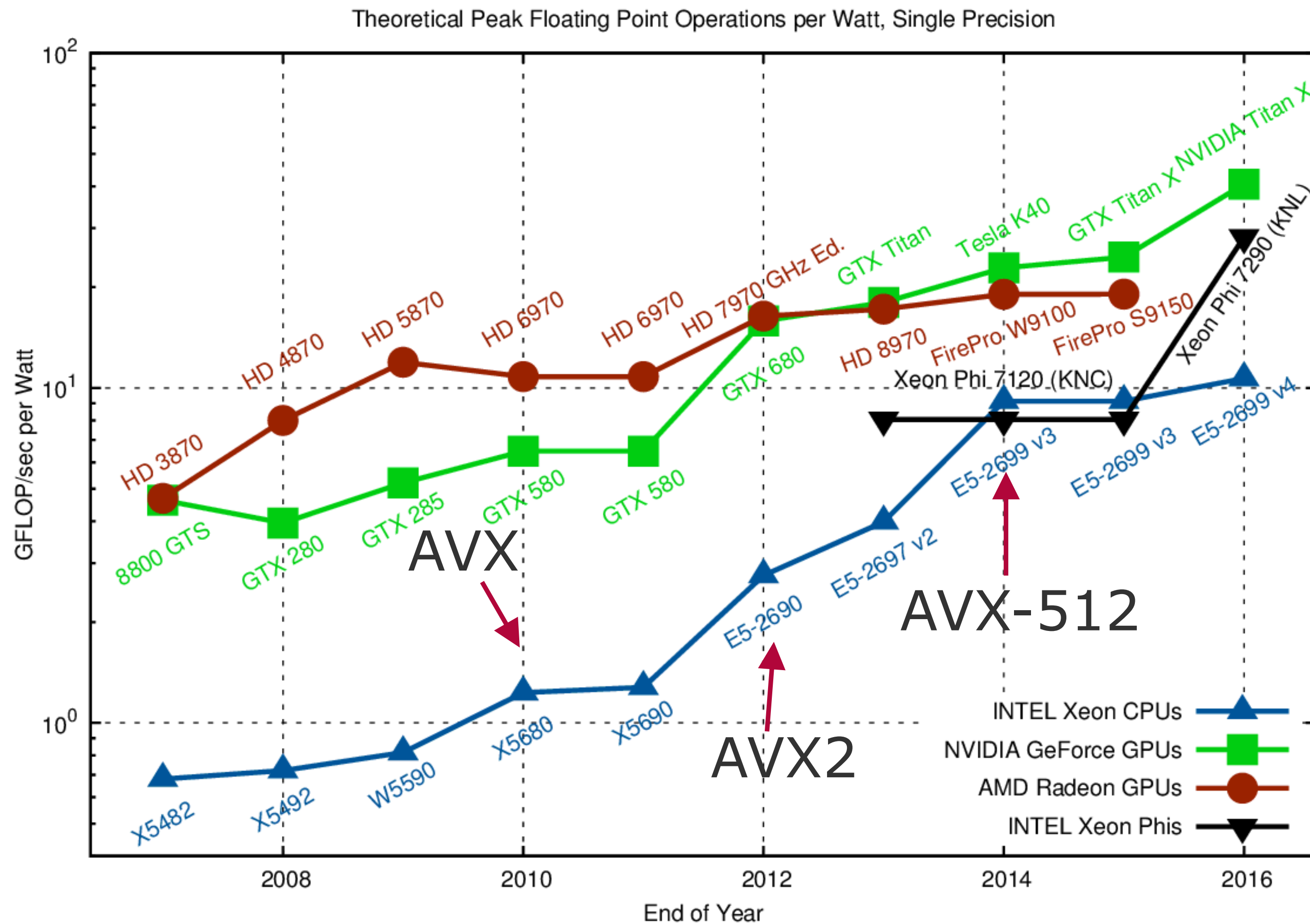
- Countable loops
- Static counts (length does not change)
- Single entry and single exit (read: no data-depended `break`)
- All function calls can be in-lined, or are math intrinsics (`sin`, `floor`, ...)
- Straight-line code (no `switch`-statements), mask-able `if`/`continue`

```
for (int i=0; i<length; i++) {
    float s = b[i]*b[i] - 4*a[i]*c[i];
    if ( s >= 0 ) {
        s = sqrt(s) ;
        x2[i] = (-b[i]+s)/(2.*a[i]);
        x1[i] = (-b[i]-s)/(2.*a[i]);
    } else {
        x2[i] = 0.;
        x1[i] = 0.;
    }
}
```

HPI Hasso Plattner Institut
Digital Engineering · Universität Potsdam

C2: GPUs

Why GPUs?

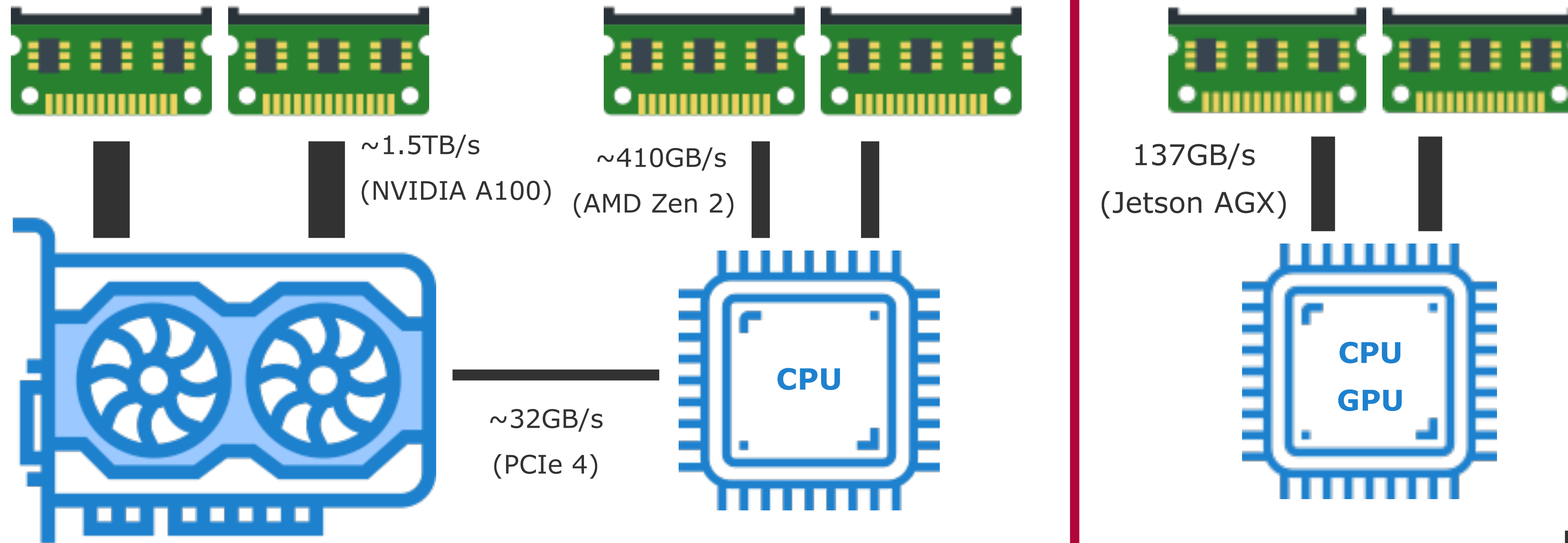


>25% of HPC systems in the Top500 (Nov ’18) are powered by GPUs

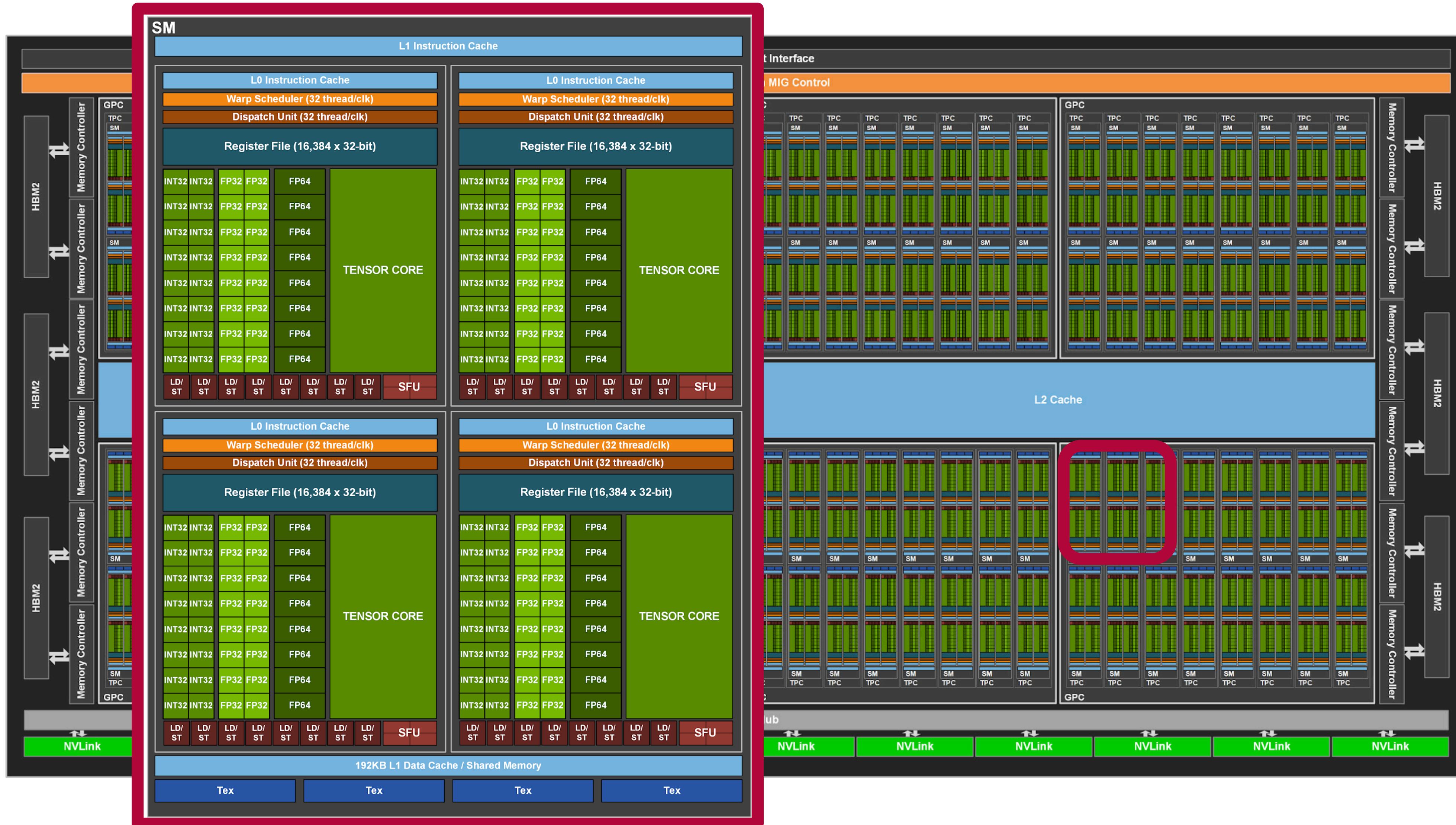
[https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/, https://www.top500.org/statistics/list/]

GPU Hardware: Discrete vs. Integrated GPUs

HPI Hasso Plattner Institut



Hardware: NVIDIA GA100 Full GPU with 128 SMs



ParProg20 C2
GPUs

Max Plauth

CUDA Programming Model: Kernels

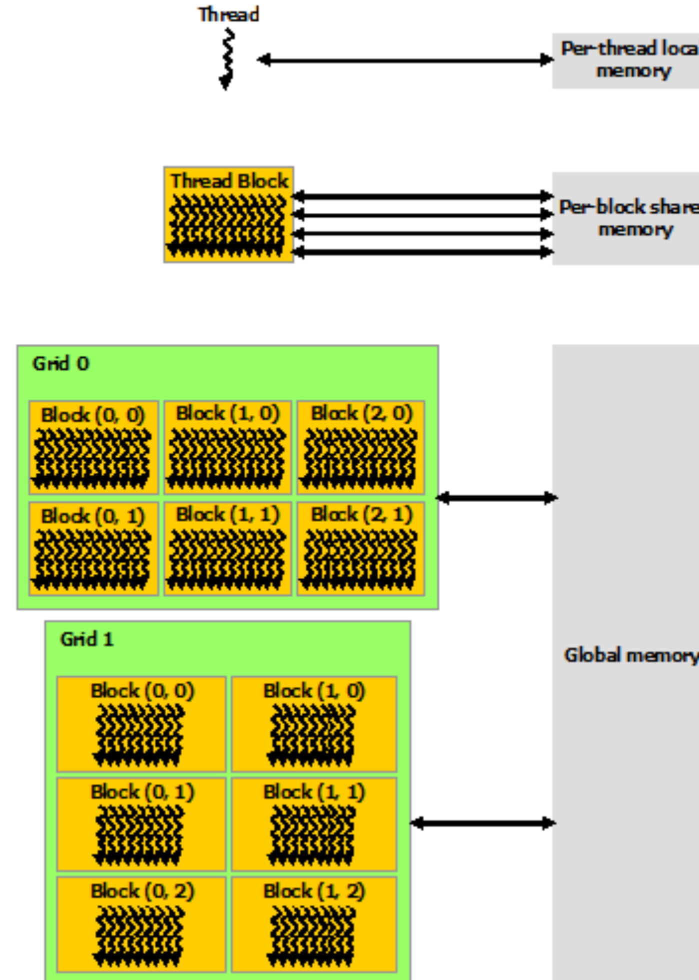
- „a routine compiled for high throughput accelerators" (Wikipedia)
- An instance of a kernel function is executed once per thread
- Indices determine what portion of work is performed by a kernel instance
- Think of kernels as the body of an inner loop

```
void
serial_mul(const float* a,
          const float* b,
             float* c,
             int n)
{
  for(int i = 0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

```
__global__ void
mul(__global__ const float* a,
    __global__ const float* b,
    __global__       float* c)
{
  int id = threadIdx.x +
     blockIdx.x * blockDim.x;
  c[id] = a[id] * b[id];
}
```

CUDA Programming Model: Memory Hierarchy

- Register File
  - Private to each thread
  - Fastest memory, several variables
- Shared Memory
  - Shared per block
  - Fast memory, several kilobytes
  - Managed manually
- Global Memory
  - Shared per process
  - Slowest memory, several gigabytes



Best Practices for Performance Tuning

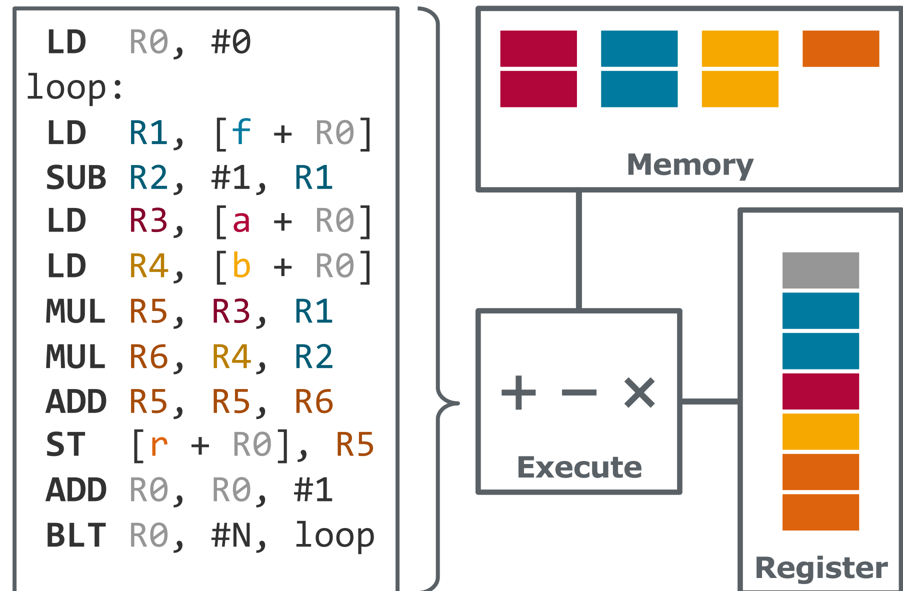
Algorithm Design	• Asynchronous, Recompute, Simple
Memory Transfer	• Chaining, Overlap Transfer & Compute
Control Flow	• Avoid Divergent Branching
Memory Types	• Local Memory as Cache, rare resource
Memory Access	• Coalescing, Bank Conflicts
Sizing	• Work-Group Size, Work / Work-Item
Instructions	• Shifting, Fused Multiply, Vector Types
Precision	• Native Math Functions, Build Options

Hasso Plattner Institut
Digital Engineering • Universität Potsdam

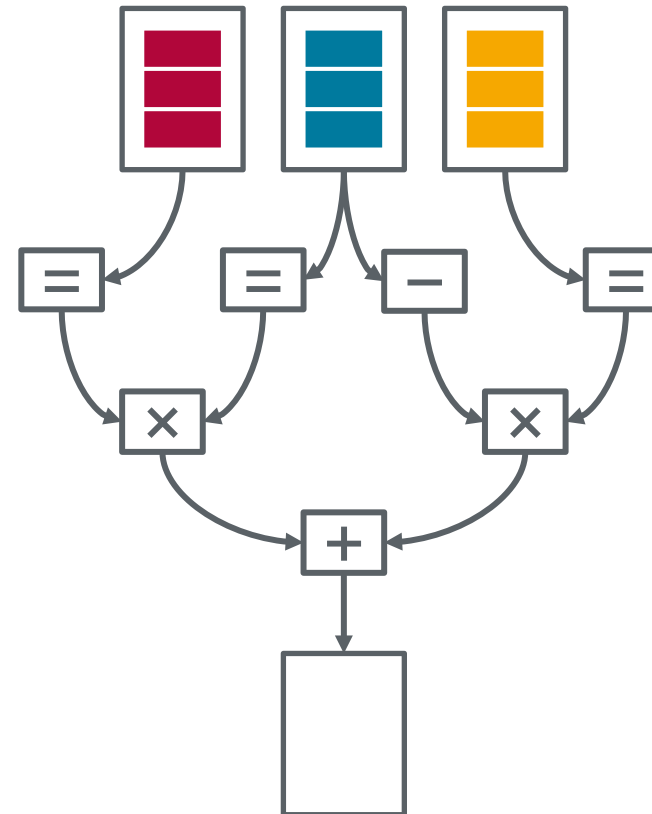
C3: FPGA

Example:

Given Arrays a, b, and f calculate `r[i] = a[i] × f[i] + b[i] × (1 - f[i])`



General Purpose Hardware



Custom Hardware

FPGA Characteristics
Hardware Structure

FPGA fabric is a regular structure of hardware primitives and an interconnect for signal lines

- **Interconnect** can be configured to connect signals lines between primitives
- **Primitives** can be configured to select variations of their basic behavior

Interconnect

IO

RAM/ALU

Logic

2020 C3

elerators

zel

FPGA Characteristics
Performance

Maximum clock frequency is design specific!

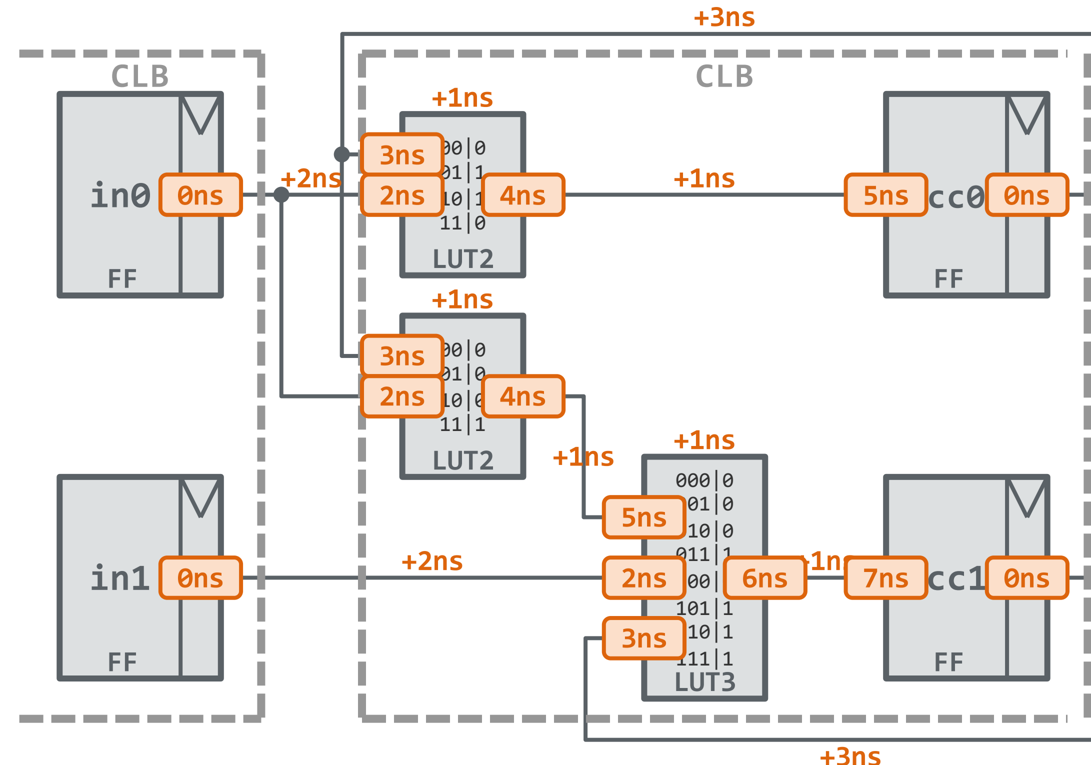
- Combinatorial paths begin and end at flipflops
- Clock period must be longer that the maximum path delay

Maximum delay:

$$\max\{t_\delta\} = 7\text{ns}$$

Clock frequency:

$$f \leq \frac{1}{\max\{t_\delta\}} = 143\text{MHz}$$

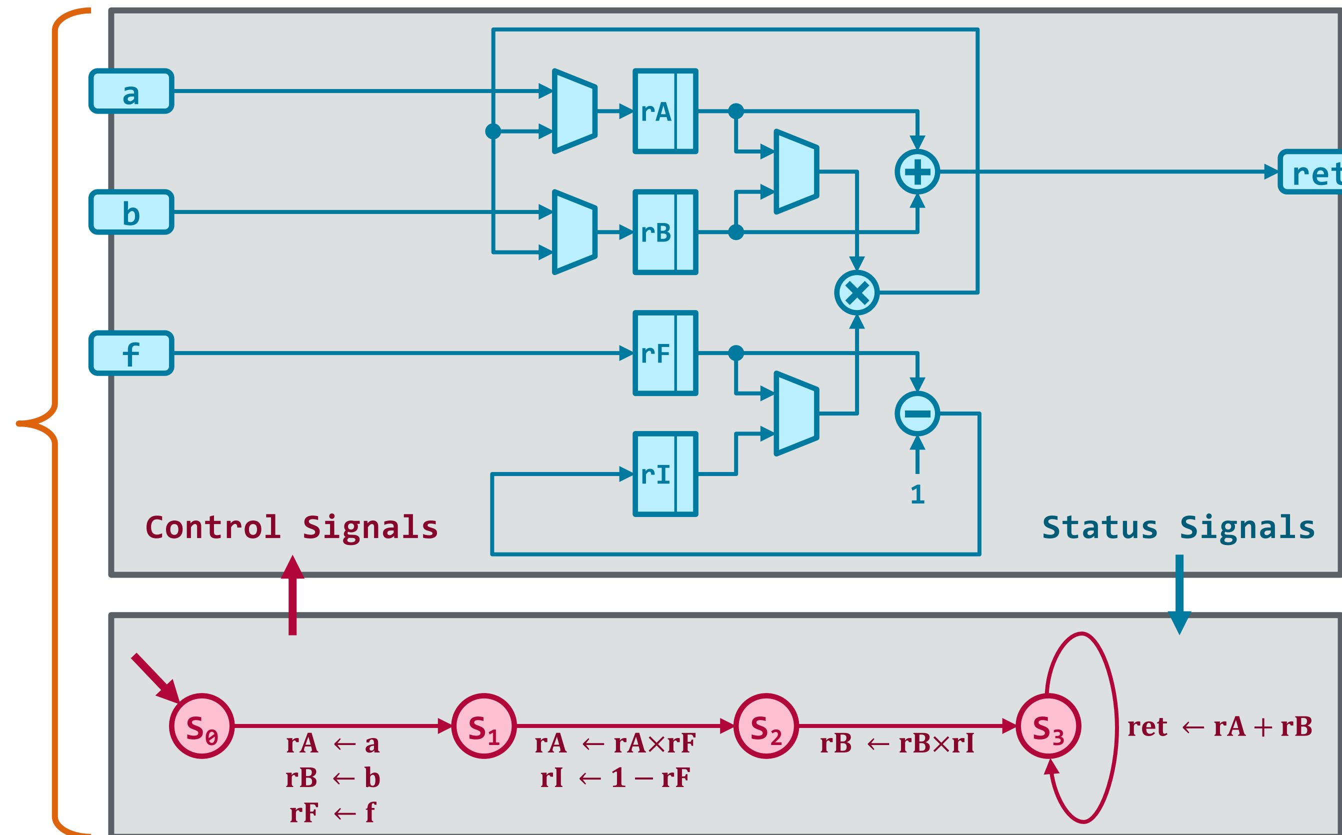


FPGA Design
Basic Patterns

Any program can be transformed into an equivalent hardware design:

- Variables and operations are realized in the **datapath**
- Control flow is realized through a **finite state machine** (FSM) controlling the datapath

```
int proc(int a, int b, int f)
{
  int f_inv = 1 - f;
  a *= f;
  b *= f_inv;
  return a + b;
}
```



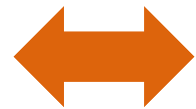
FPGA Design
Dataflow Model

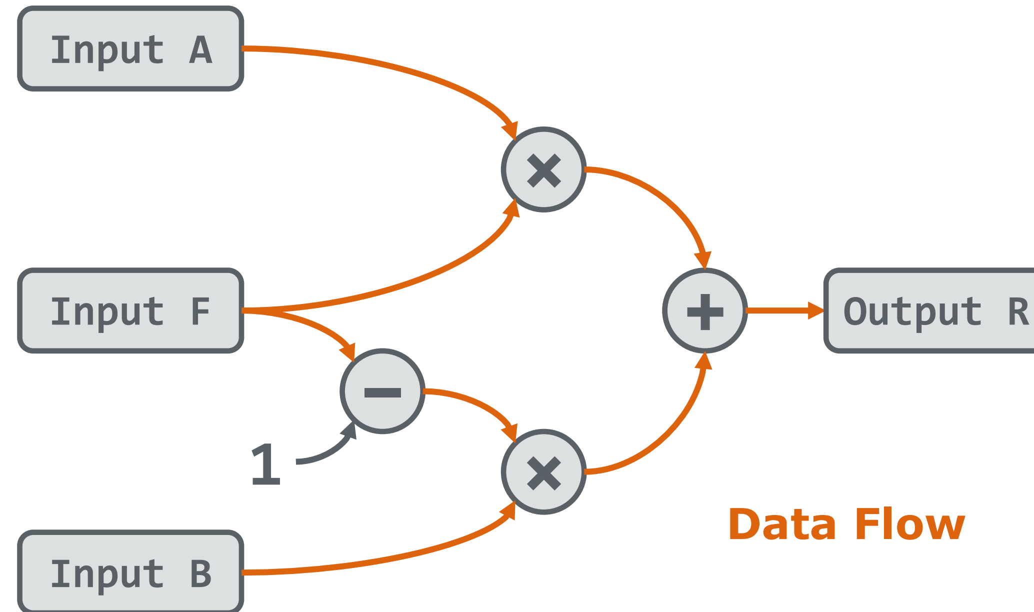
HPI Hasso Plattner Institut

- Dataflow is a computational model based on **streams of data units**, that are processed by traversing a **network of operators**
  - Enables a **flexible kind of task parallelism**, where operations are not orchestrated by control flow but availability of data operands

```
int proc(int a, int b, int f)
{
  int f_inv = 1 - f;
  a *= f;
  b *= f_inv;
  return a + b;
}
```

Control Flow



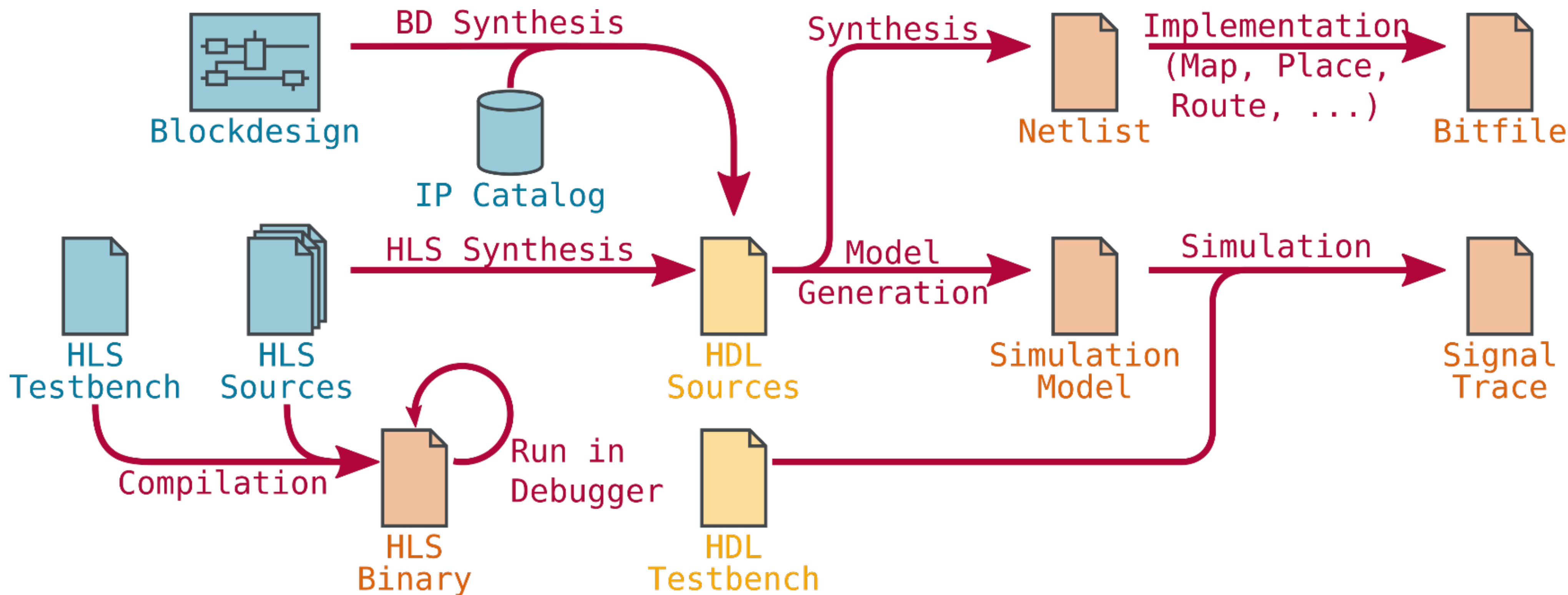


- **Workloads with an efficient dataflow representation usually yield an efficient hardware implementation!**

FPGA Development Workflow

High-level design methods extend the frontend of traditional workflows.

They usually produce HDL descriptions as intermediate artifacts.

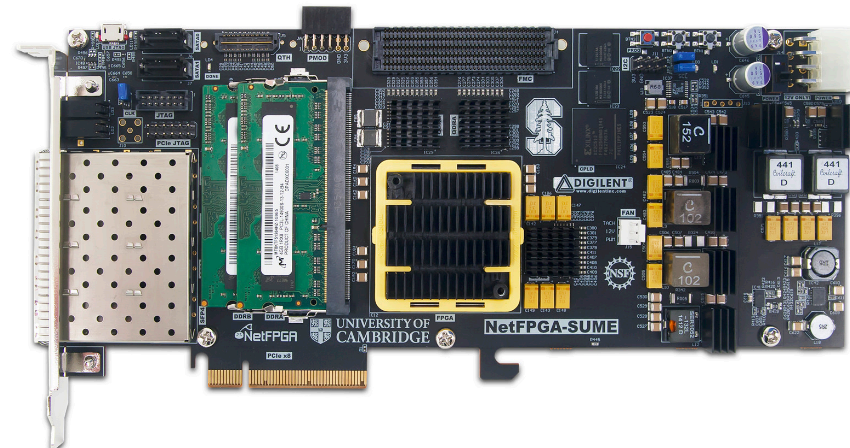


FPGA Accelerators

FPGA accelerator cards provide a **host system interface** as well as **local memory and IO resources**.

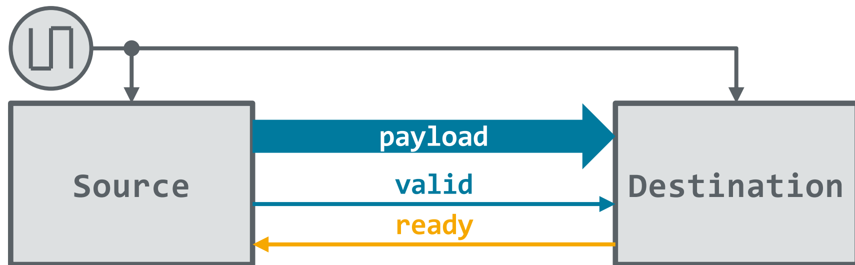
- DRAM modules to complement the limited BRAM capacity on the FPGA
- Flash Storage
- Network Interfaces
- Video and Peripheral Ports
- Auxilliary Accelerators like Crypto Units or A/V Codecs

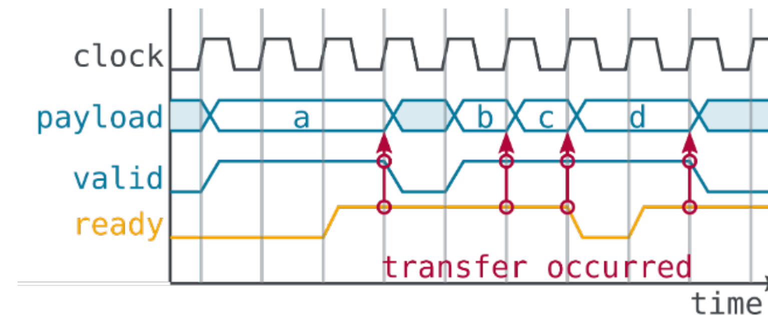
...



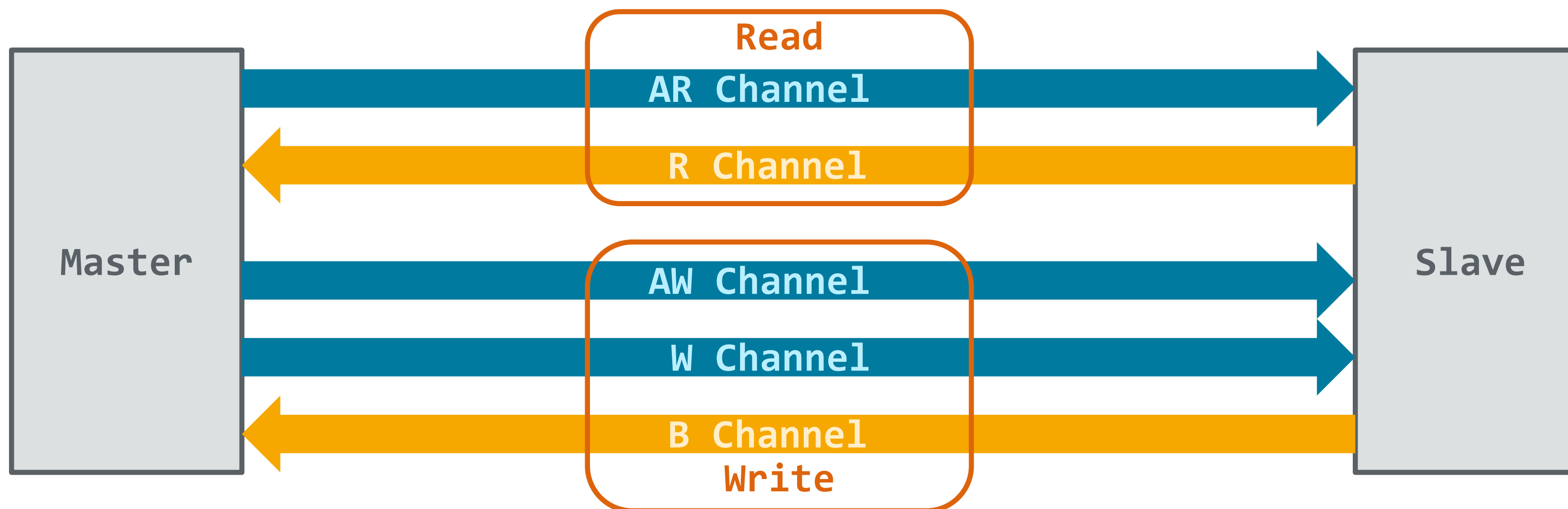
Excursion
AMBA Protocol Family

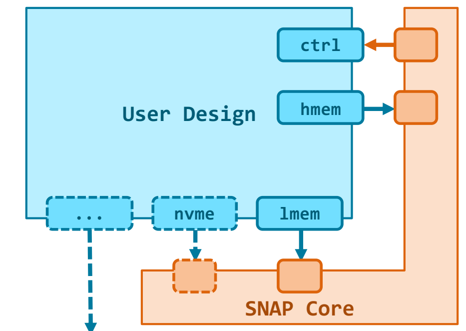
- **Channels** consist of: **Payload • Valid handshake • Ready handshake**





- **Advanced Extensible Interface Stream** (AXI Stream) ~ sequential access
- **Advanced Extensible Interface** (AXI) ~ random access

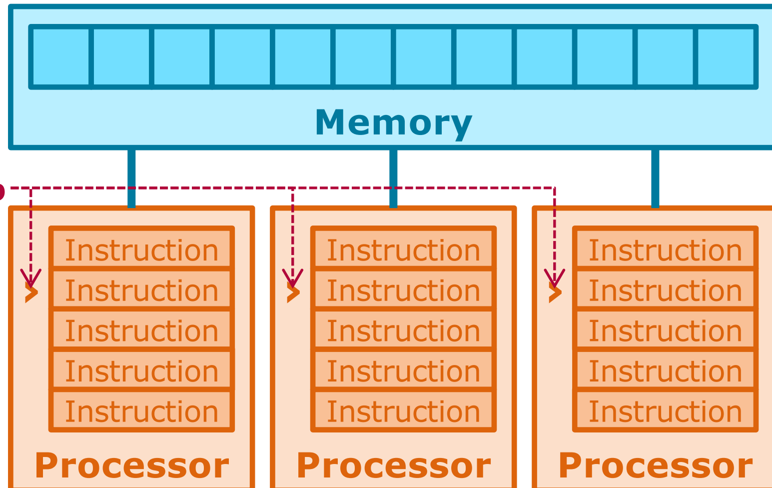




D1: Shared Nothing Basics

Parallel Random Access Machine (PRAM)

Natural extension of the Random Access Machine (RAM) model:



- **Arbitrary amount of memory**
- **Constant memory access latency**
- **Arbitrary number of processors**
- **Lockstep execution**

Multiple processors can read the same address →

Multiple processors can write the same address ↓

Exclusive Read, Exclusive Write **EREW**	Concurrent Read, Exclusive Write **CREW**
Exclusive Read, Concurrent Write **ERCW**	Concurrent Read, Concurrent Write **CRCW**

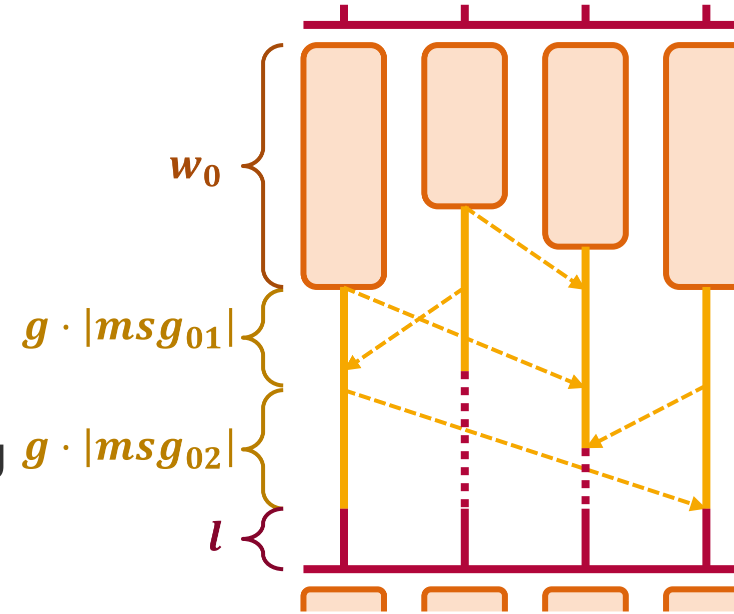
Arbitration Policies:

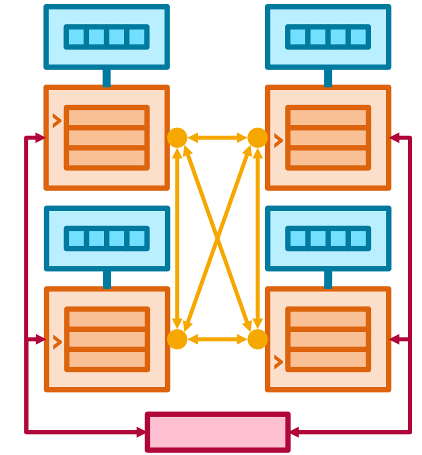
- Common
- Arbitrary
- Priority
- Aggregate (Sum, Max, Avg, …)

[Valiant1990]
Bulk Synchronous Parallel Model (BSP)

Algorithms are divided into three repeating phases, forming multiple <u>supersteps</u>:

1. **Local Computation**
2. **Global Communication**
3. **Synchronization**





Superstep duration varies at runtime depending on computational and communication load.

Performance estimates using the following parameters:

Computation time: $t_W = \max\{w_i\}$

Communication time: $t_C = g \cdot m \cdot h$

g ~ message bandwidth
$m = \max\{|msg_k|\}$ ~ message size
$h = \max\{\#in_i, \#out_i\}$ ~ communication pattern

Synchronization overhead: $t_S = l$

[Culler1993] LogP Model

LogP enables a fine-grained analysis of communication patterns.

Parameters:

P – **#processors**

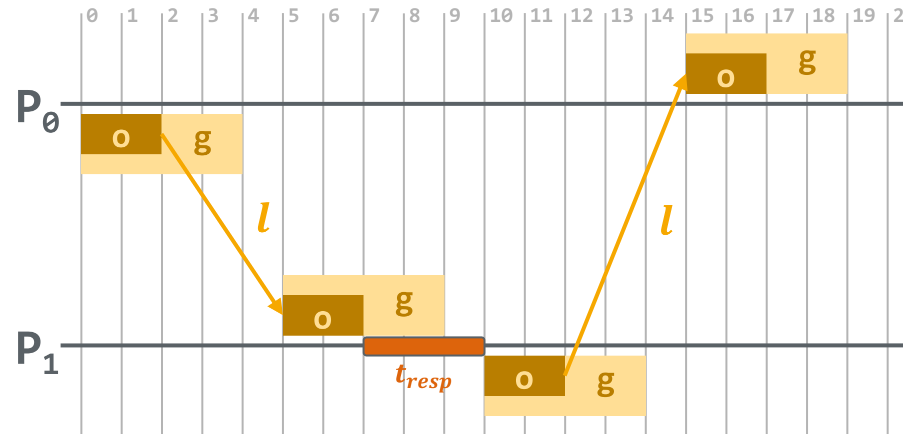
g – **gap** (time in cycles between messages from / to a single processor)

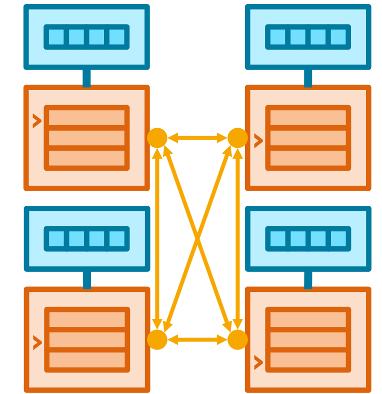
o – **overhead** (time in cycles for send / receive operation)

l – **latency** (time in cycles between transmission and reception of a message)

Example: Request-Response sequence between two processors

- $P = 2\,;\ l = 3\,;\ g = 4\,;\ o = 2\,;t_{resp} = 3$
- $t_{total} = 2 \cdot l + 4 \cdot o + t_{resp} = 17$



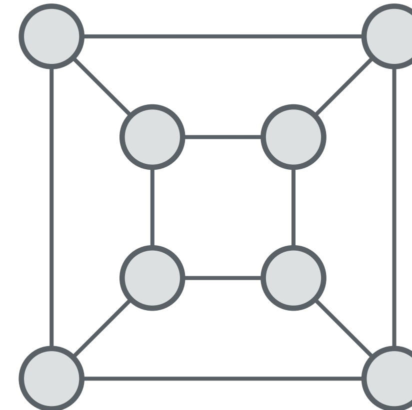


Network Topologies

HPI Hasso Plattner Institut

Topologies are characterized by multiple metrics:

- **Diameter** ~ Latency
  Maximum distance between any two nodes
- **Connectivity** ~ Resilience
  Minimum number of removed edges to cause partition
- **Bisection Bandwidth** ~ Throughput
  Transfer capacity across balanced network cuts
- **Cost** ~ Network complexity
  Total number of edges
- **Degree** ~ Node complexity
  Maximum number of edges per node
- **Link Bandwidth**

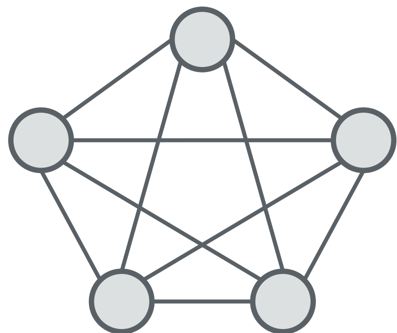


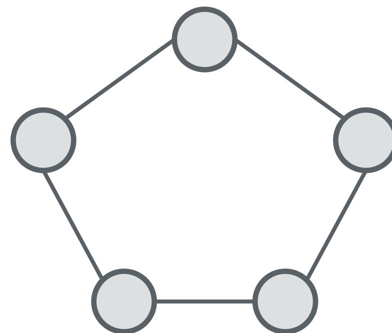
Network Topologies

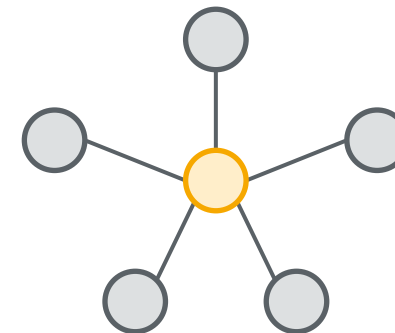
Fully Connected	
Diameter	1
Connectivity	$n-1$
Cost	$\frac{n^2-n}{2}$
Degree	$n-1$

Ring	
Diameter	$\left\lfloor \frac{n}{2} \right\rfloor$
Connectivity	2
Cost	n
Degree	2

Star	
Diameter	2
Connectivity	1 (single node)
Cost	n
Degree	$1 \mid n$ (!)



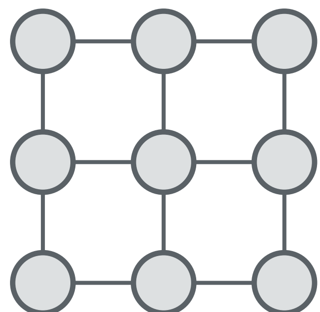




Network Topologies

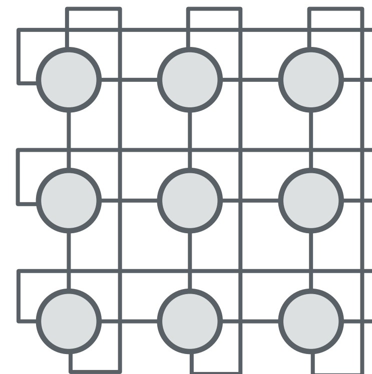
d-Mesh	
Diameter	$d \cdot (k-1)$ $= d \cdot (\sqrt[d]{n} - 1)$
Connectivity	d
Cost	$d \cdot k^{d-1} \cdot (k-1)$ $= d \cdot (n - n^{d-1/d})$
Degree	$2 \cdot d$

d-Torus	
Diameter	$\lfloor d \cdot (k-1)/2 \rfloor$ $= \lfloor d \cdot (\sqrt[d]{n} - 1)/2 \rfloor$
Connectivity	$2 \cdot d$
Cost	$d \cdot k^d = d \cdot n$
Degree	$2 \cdot d$



$d = 2$
$k = 3$
$n = k^d = 9$

d-Hypercube
= **d-Mesh with k = 2**



$d = 2$
$k = 3$
$n = k^d = 9$

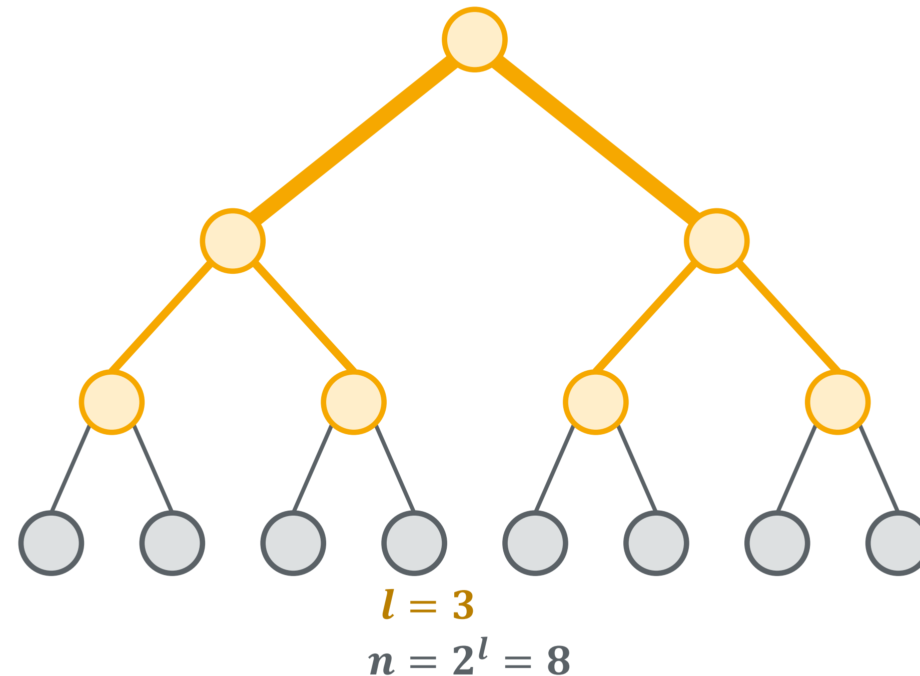
HPI Hasso Plattner Institut

HPI Hasso Plattner Institut

Fat Tree of Depth l

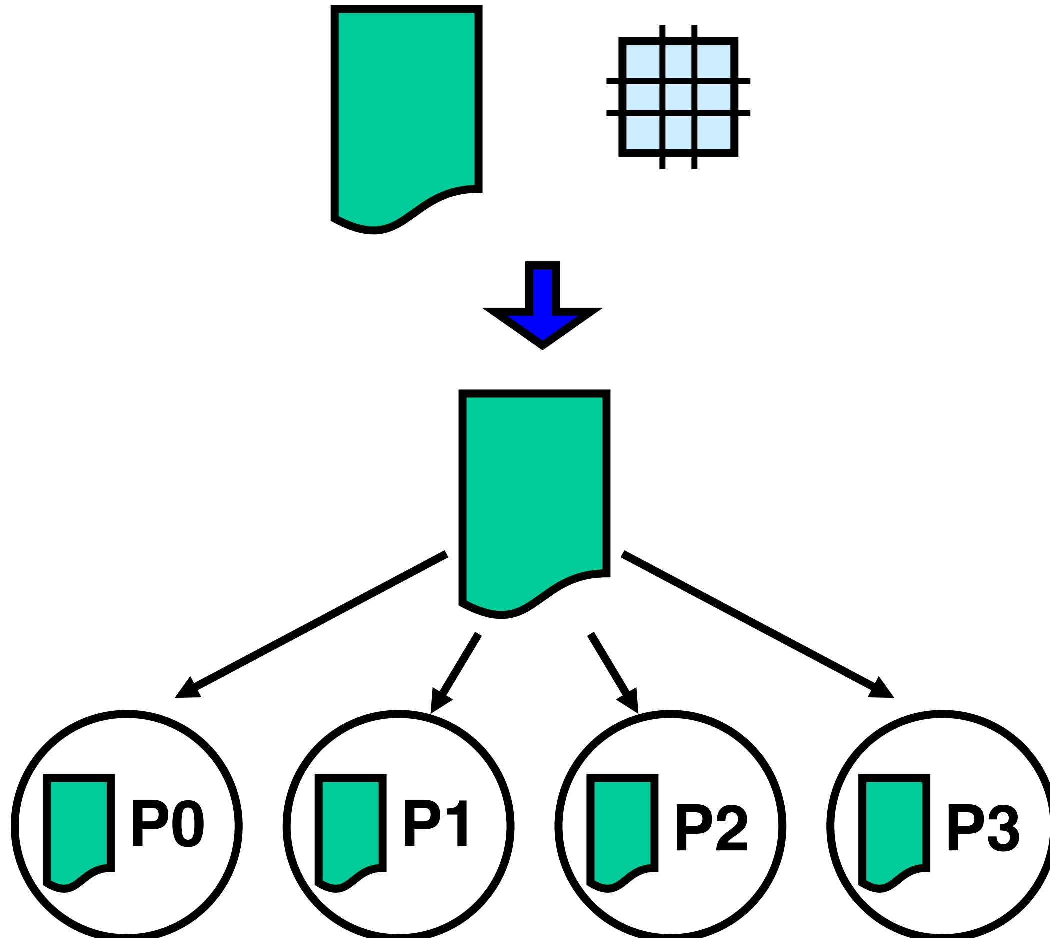
= Binary l-level switch hierarchy,
where uplink bandwidth equals sum of downlink bandwidths

Fat Tree	
Diameter	$2 \cdot l = 2 \cdot log_2(n)$
Connectivity	1
Cost	$2^{l+1} - 2 = 2 \cdot n - 2$
Cost (Bandwidth adjusted)	$l \cdot 2^l = n \cdot log_2(n)$
Degree	$1 \mid 3$

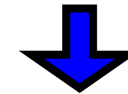


D2: MPI

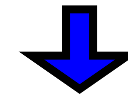
Single Program Multiple Data (SPMD)



seq. program and data distribution

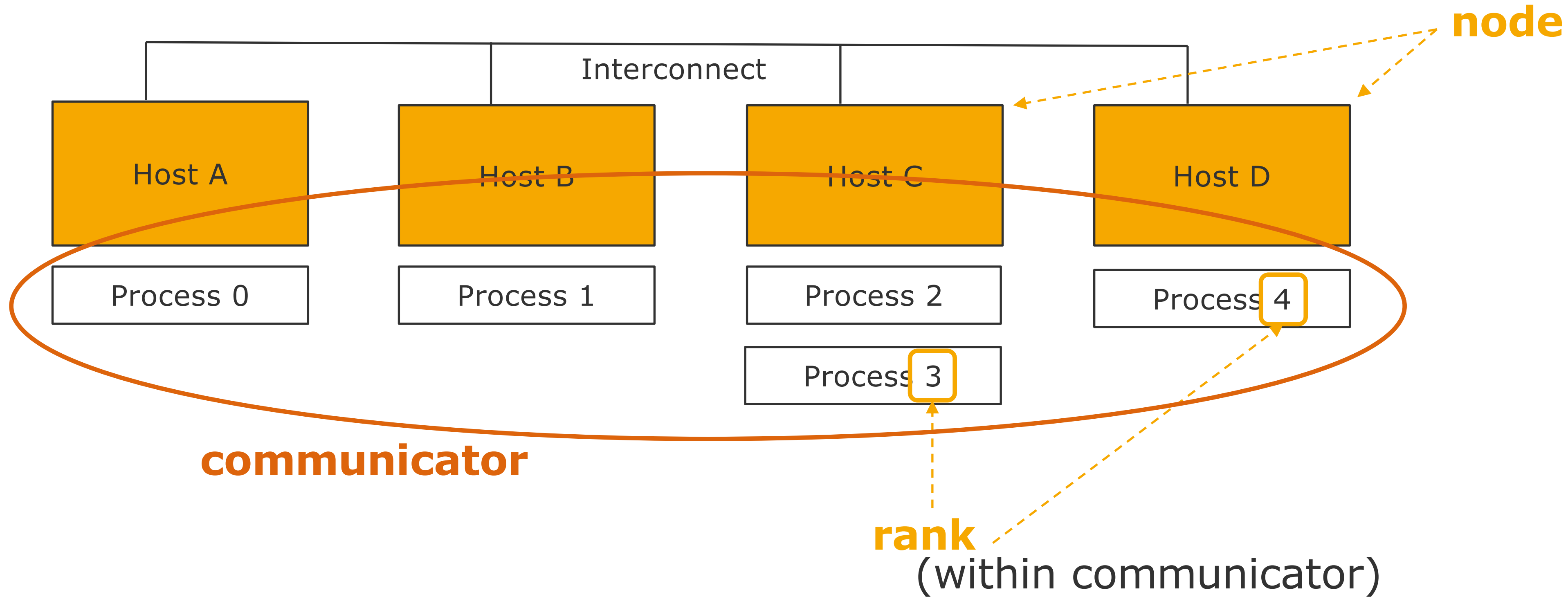


seq. node program with message passing



identical copies with different process identifications

MPI Communication Terminology



Communicator: handle for group of processes (MPI_COMM_WORLD = all)

Size: Number of processes in a communicator

Circular Left Shift Example

shifts <number of positions>

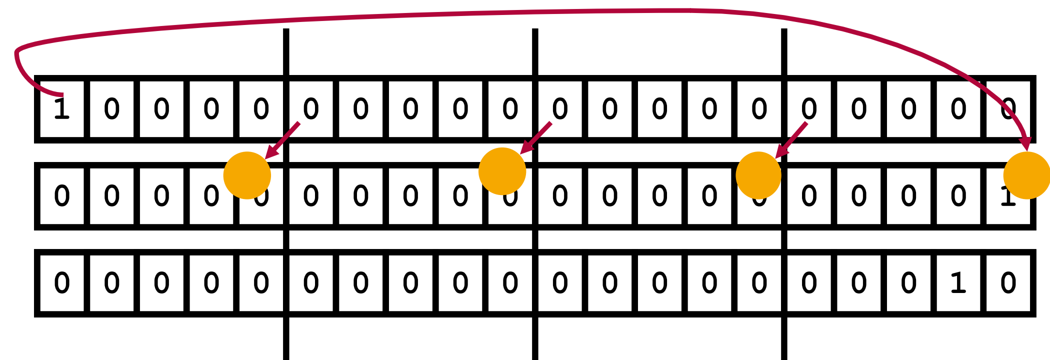
Description

- Position 0 of an array with 100 entries is initialized to 1. The array is distributed among all processes in a blockwise fashion.
- A number of circular left shift operations is executed.
- The number is specified via a command line parameter.

Process 0

Other Processes

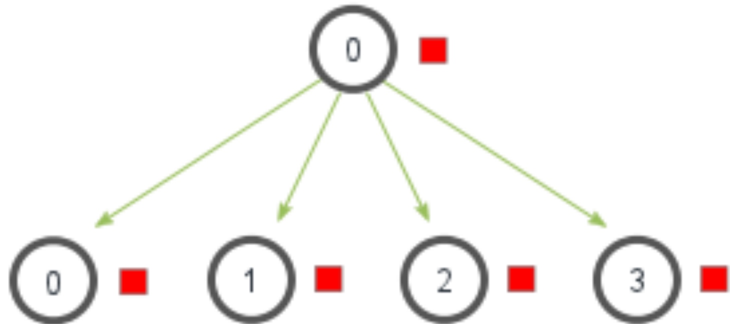
```
for (i=0;i<shifts;i++){
  if (myid==0){
    MPI_Send(&values[0], 1, MPI_INT, l
             MPI_COMM_WORLD);
    for (j=1;j<100/np;j++){
      values[j-1]=values[j];
    }
    MPI_Recv(&values[100/np-1], 1, MPI
             10, MPI_COMM_WORLD, &stat
  }else{
    int buf=values[0];
    for (j=1;j<100/np;j++){
      values[j-1]=values[j];
    }
    MPI_Recv(&values[100/np-1], 1, MPI_INT, rnbr,
            10, MPI_COMM_WORLD, &status);
    MPI_Send(&buf, 1, MPI_INT, lnbr, 10,
             MPI_COMM_WORLD);
  }
}
```

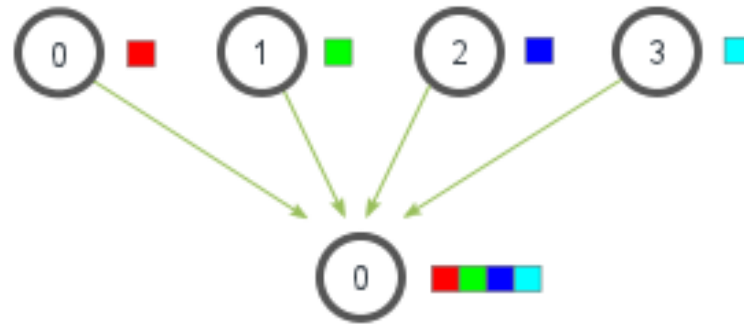


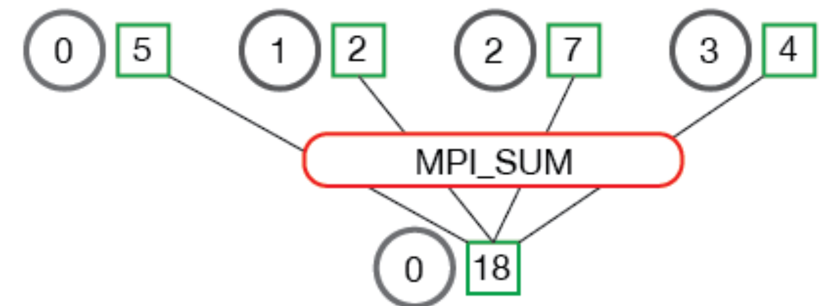
Send and Receive Protocols

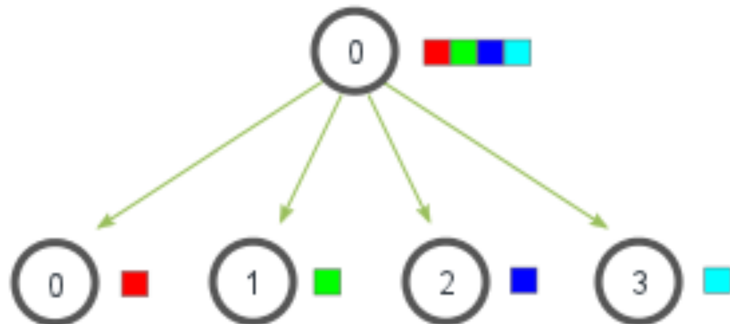
	Blocking	Non-Blocking
Buffered	Send call returns after data has been buffered `MPI_BSend`	Send call returns after initiating DMA transfer to the buffer `MPI_IBSend`
Non-Buffered	Send call returns after matching receive is Available `MPI_SSend`	No semantics promised. `MPI_ISSend`

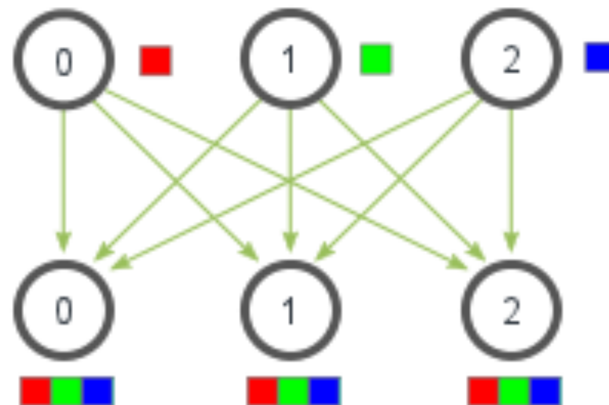
MPI Collective Operations







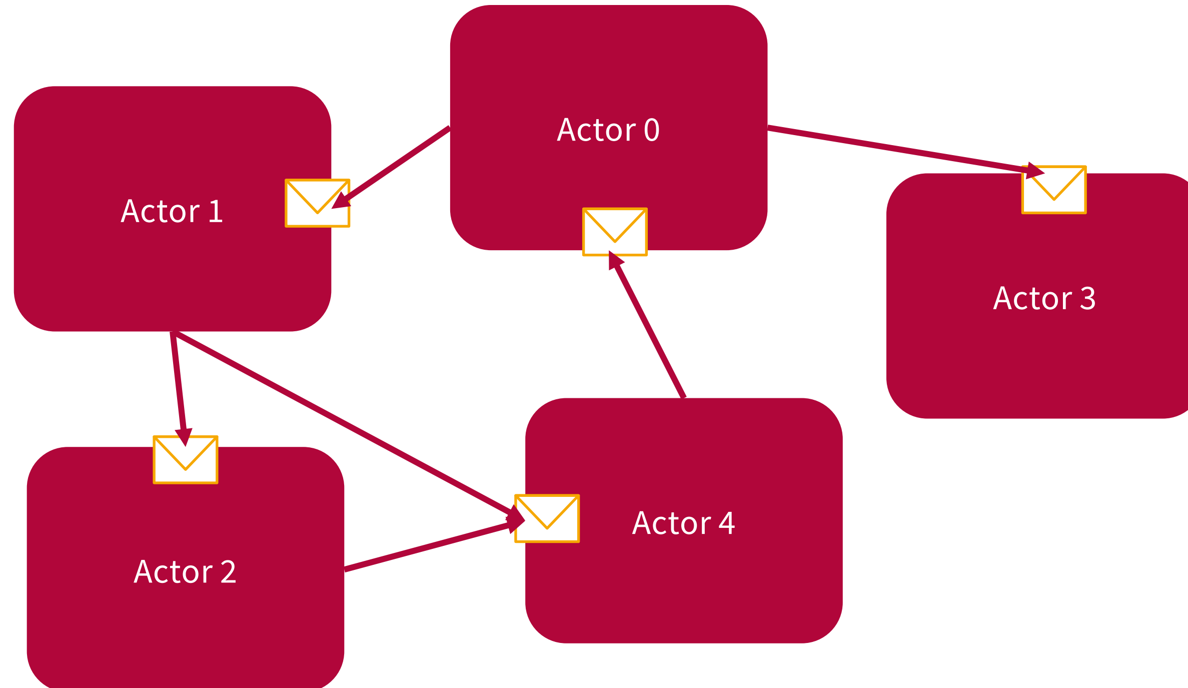




D3: Actors

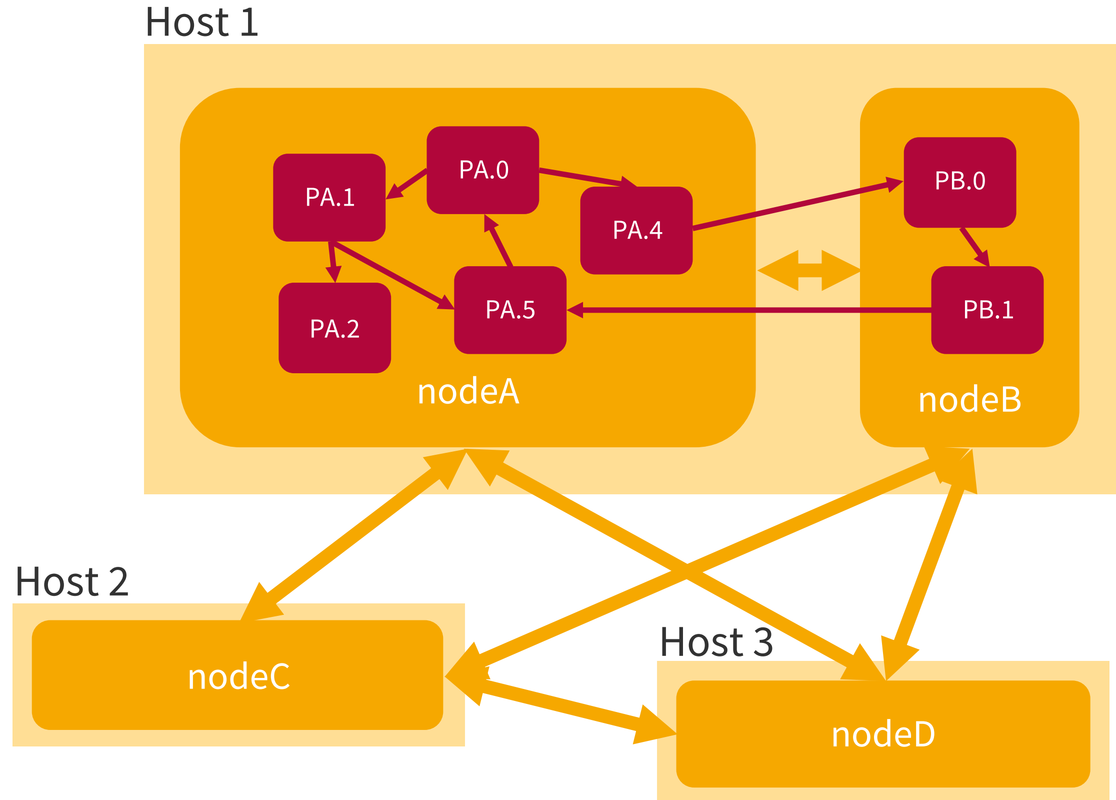
Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Actors



„Everything is an actor“

Erlang Cluster Terminology

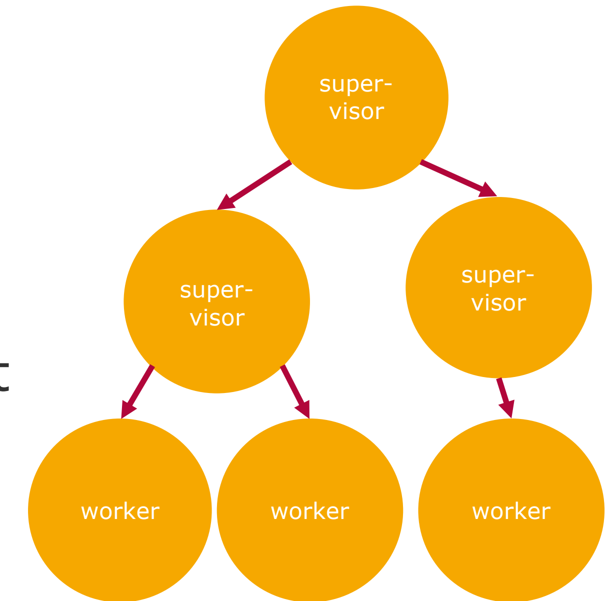


An Erlang cluster consists of multiple interconnected nodes, each running several light-weight processes (actors).

Message passing implemented by shared memory (same node), TCP (ERTS), …

Concurrency in Erlang

- Each concurrent activity is called *process*, started from a function
- Local state is call-stack and local variables
- Only interaction through asynchronous *message passing*
- Processes are reachable via unforgable name (pid)
- Design philosophy is to spawn a worker process for each new event
  - `spawn([node, ]module, function, argumentlist)`
  - Spawn always succeeds, created process may terminate with a runtime error later (*abnormally*)
  - Supervisor process can be notified on fails



Armstrong, Joe. "Concurrency oriented programming in Erlang." Invited talk, FFG (2003).

Enjoy whatever helps you learning.
Much success for the exam!

HPI Hasso Plattner Institut

Digital Engineering · Universität Potsdam

Parallel Programming and Heterogeneous Computing

E2 - Summary

Max Plauth, Sven Köhler, Felix Eberhardt, Lukas Wenzel and Andreas Polze

Operating Systems and Middleware Group