703650 VO Parallel Systems WS2019/2020
Summary and Q/A

Philipp Gschwandtner

Summary of This Entire Semester – Tons of Fun!

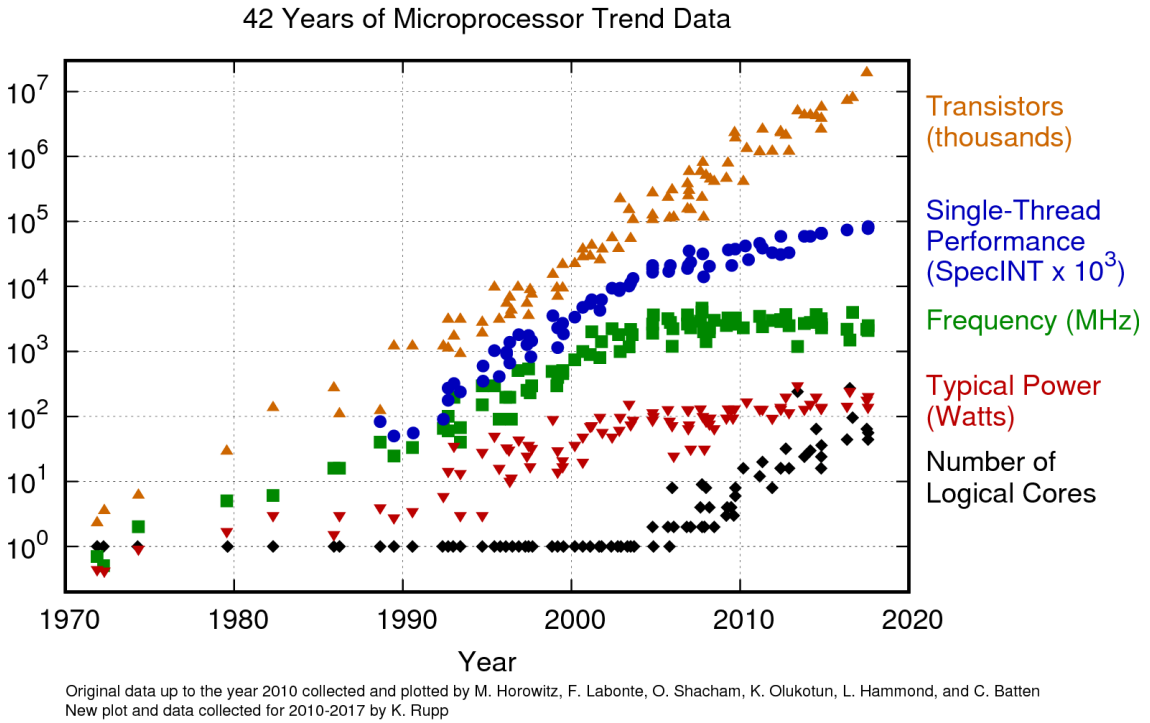
- Week 01: Motivation & A Crash Course in Parallel Hard- and Software
- Week 02: MPI - Message Passing Interface
- Week 03: Measuring and Reporting Data
- Week 04: MPI Derived Datatypes and Virtual Topologies
- Week 05: Domain Decomposition and Load Balancing
- Week 06: 13 Dwarfs of HPC
- Week 07: MPI Groups, Communicators and One-Sided Communication
- Week 08: OpenMP Basics
- Week 09: OpenMP Advanced
- Week 10: Debugging Parallel Programs
- Week 11: Expanding Horizons: Additional Programming Models

Motivation & A Crash Course in Parallel Hard- and Software

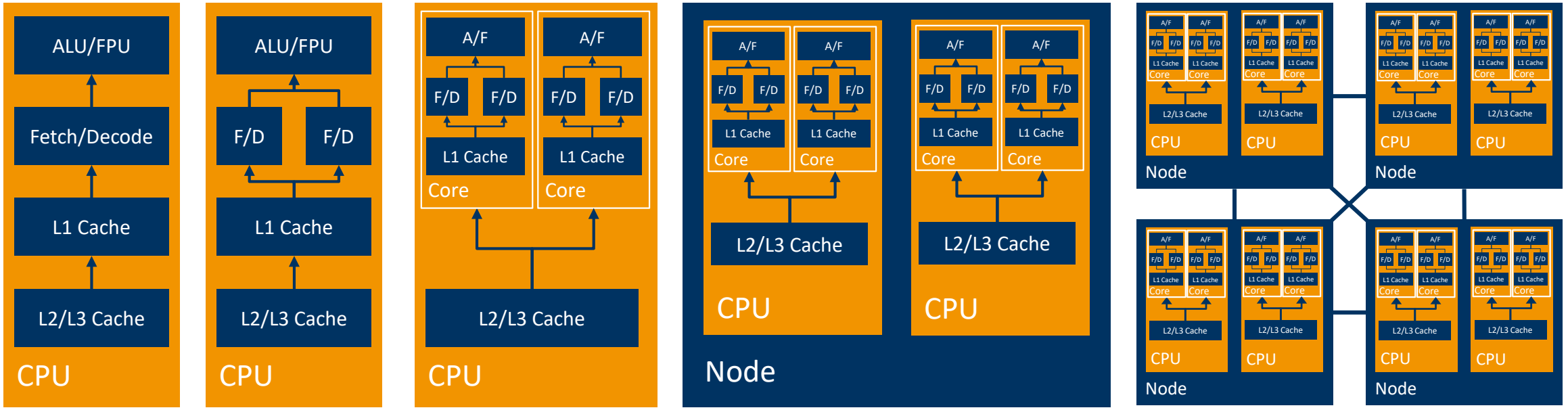
- what is parallelism, why do we need it?
  - applications and problems
  - “three walls”
- parallelism in hardware
  - HT, multi-/many-core, multi-CPU, clusters, NUMA, …
- parallelism in software
  - task & data parallelism, Flynn taxonomy, shared & distributed memory

The Three Walls & Need for Parallelism in Hardware

- power wall
  - increase in transistors means increase in dynamic power consumption
- memory wall
  - growing speed disparity between computational units and memory
- instruction-level parallelism (ILP) wall
  - diminishing returns in overlapping in-core instruction execution



Forms of Parallel Hardware (Fictional Architecture)



single-core CPU

single-core CPU + HT

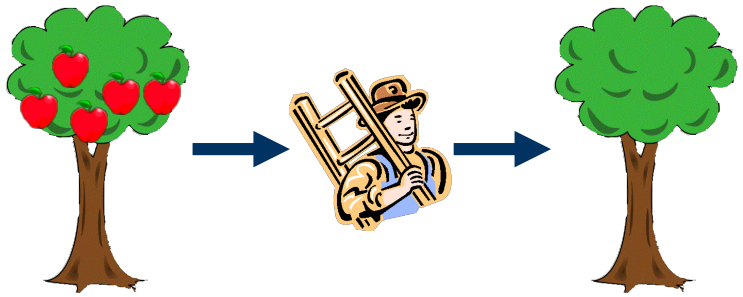
multi-core CPU + HT

shared memory node of multi-core CPUs + HT

cluster of shared memory nodes of multi-core CPUs + HT

High-Level Types of Parallelism & Flynn's Taxonomy

- data parallelism
  - execute parts of the same task on different data simultaneously
- task parallelism
  - execute different tasks within the same problem simultaneously



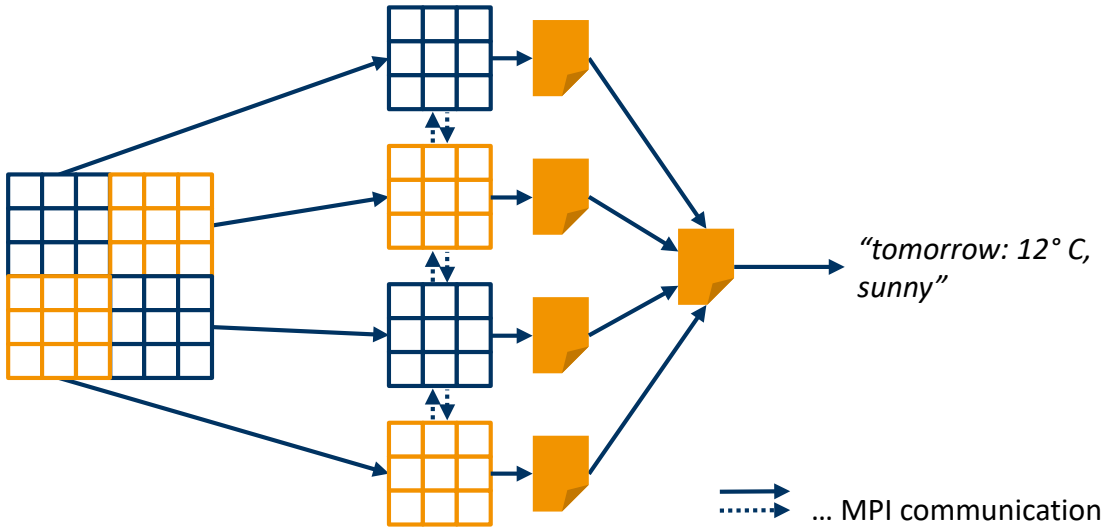
Single Instruction Single Data (SISD)	Single Instruction Multiple Data (SIMD)
Multiple Instruction Single Data (MISD)	Multiple Instruction Multiple Data (MIMD)

Shared/Distributed Memory Implications

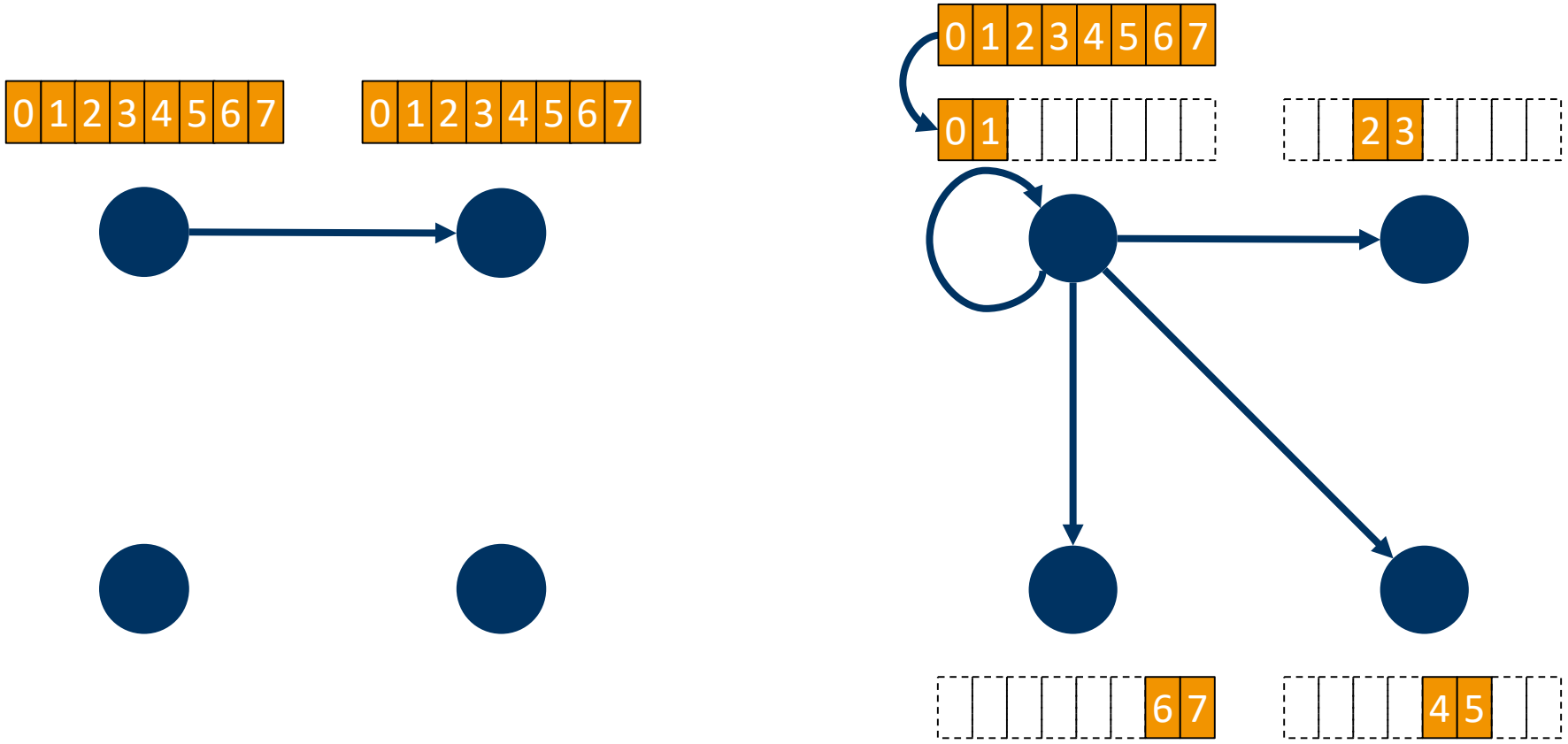
- shared memory
  - direct data access hides access cost (interconnect and memory latency)
  - very little explicit communication
    - frequently leads to race conditions
  - can often be done incrementally from sequential code
  - available memory scales with RAM of a single node (limiting factor – except for exotic hardware types such as SGI UV)
- distributed memory
  - data access cost (interconnect and memory) evident in the code
  - a lot of explicit communication
    - frequently leads to deadlocks
  - difficult to do incrementally, often all-or-nothing approach
  - available memory scales with machine size (number of nodes)

MPI - Message Passing Interface

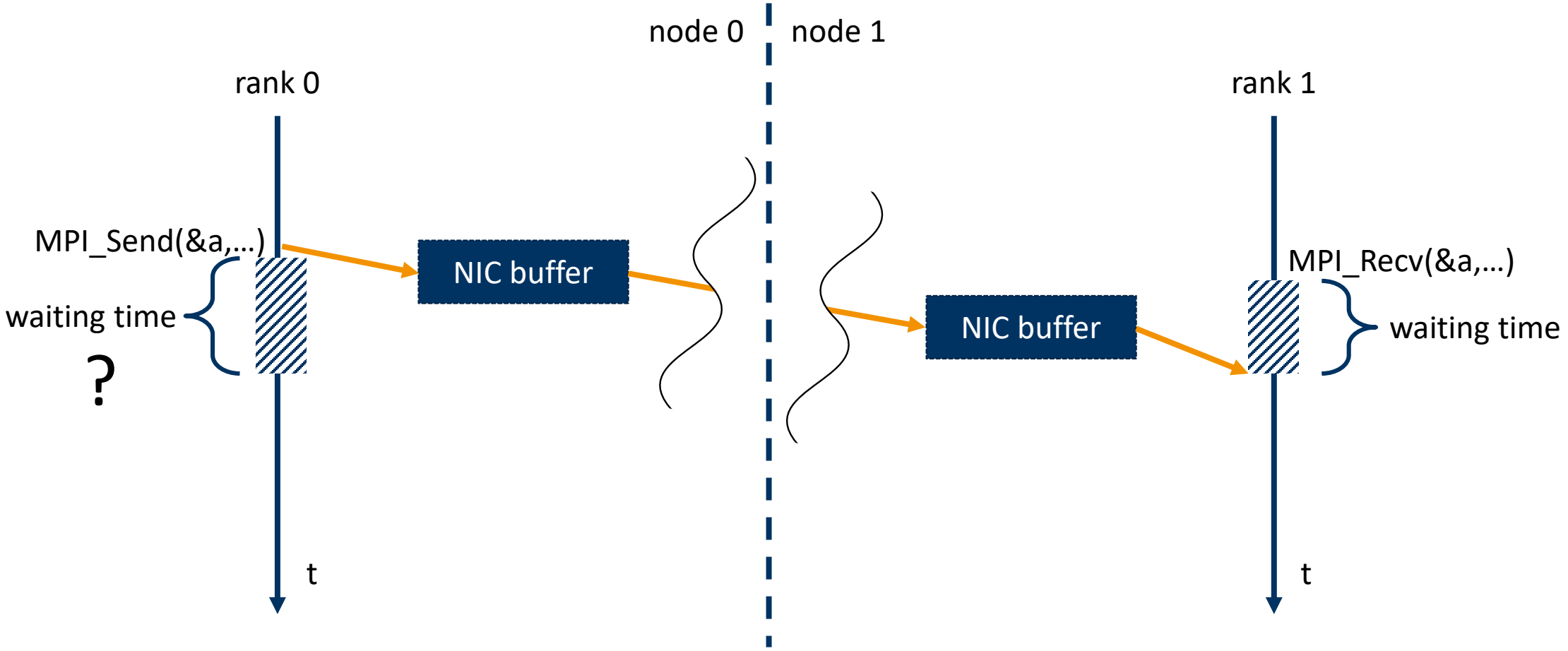
- general concepts about MPI
  - characteristics
  - program model
  - startup
- point-to-point communication
- collective communication
- practical example



Point-to-Point Communication vs. Collective Communication



(Non-)Blocking and (A)Synchronous Communication cont'd



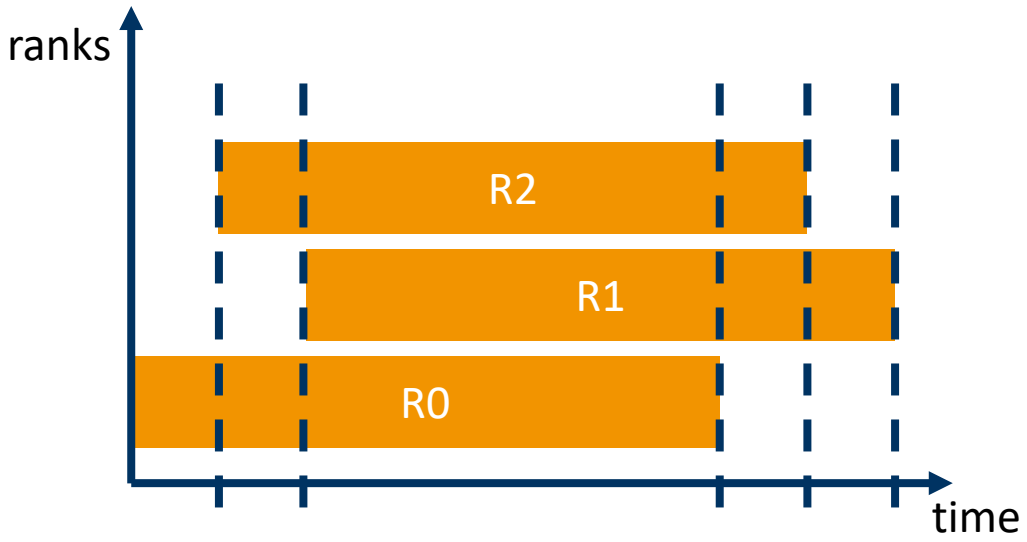
Measuring and Reporting Data

- what and how to measure
  - time
  - time-dependent (speedup, efficiency, …)
  - FLOPS
- use measurements to drive optimizations
  - Amdahl's law
- how to report measurements

Time

- *wall time*
  - time measured by looking at the wall clock
  - disregards degree of parallelism
  - the default when talking “time”
- *cpu time*
  - wall time for each rank
  - cumulative over all ranks, varies with degree of parallelism

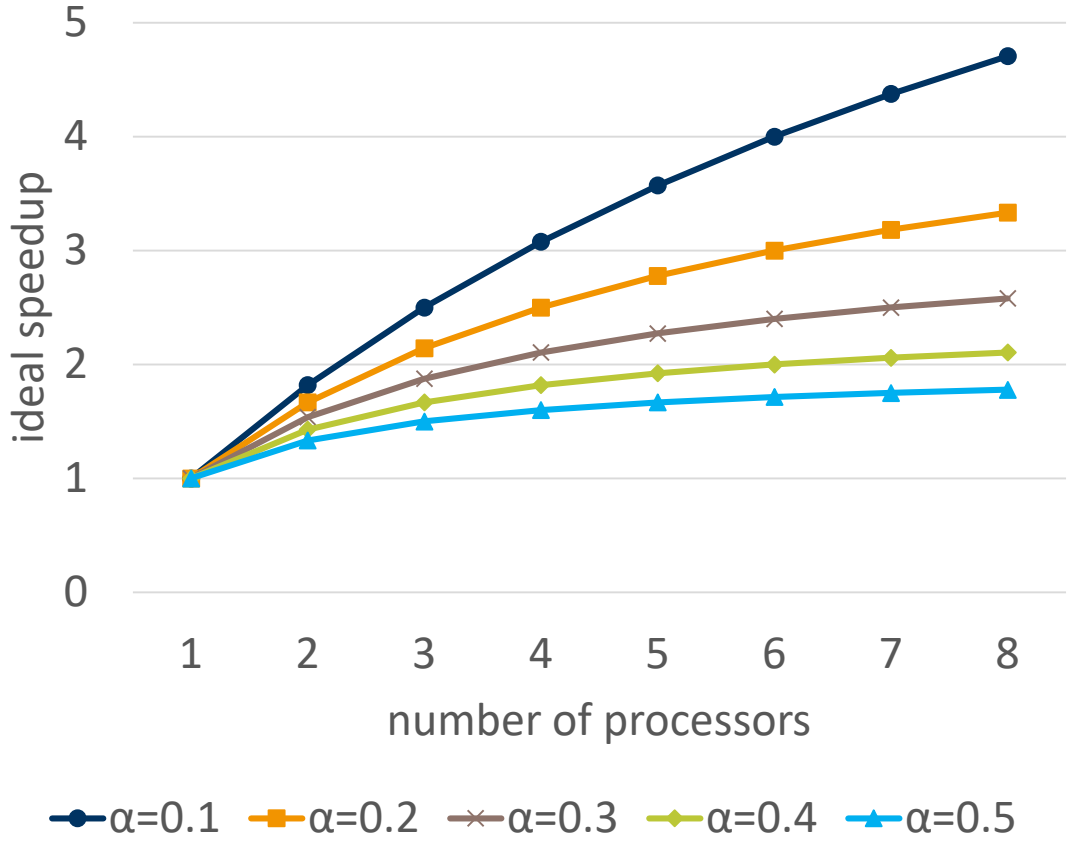
$$t_{\text{wall}} = \max_{0 \le i < N} \left(t_{\text{end},i} \right) - \min_{0 \le i < N} \left(t_{\text{start},i} \right)$$



$$t_{\text{cpu}} = \sum_{i=0}^{N-1} t_i$$

Amdahl's Law cont'd

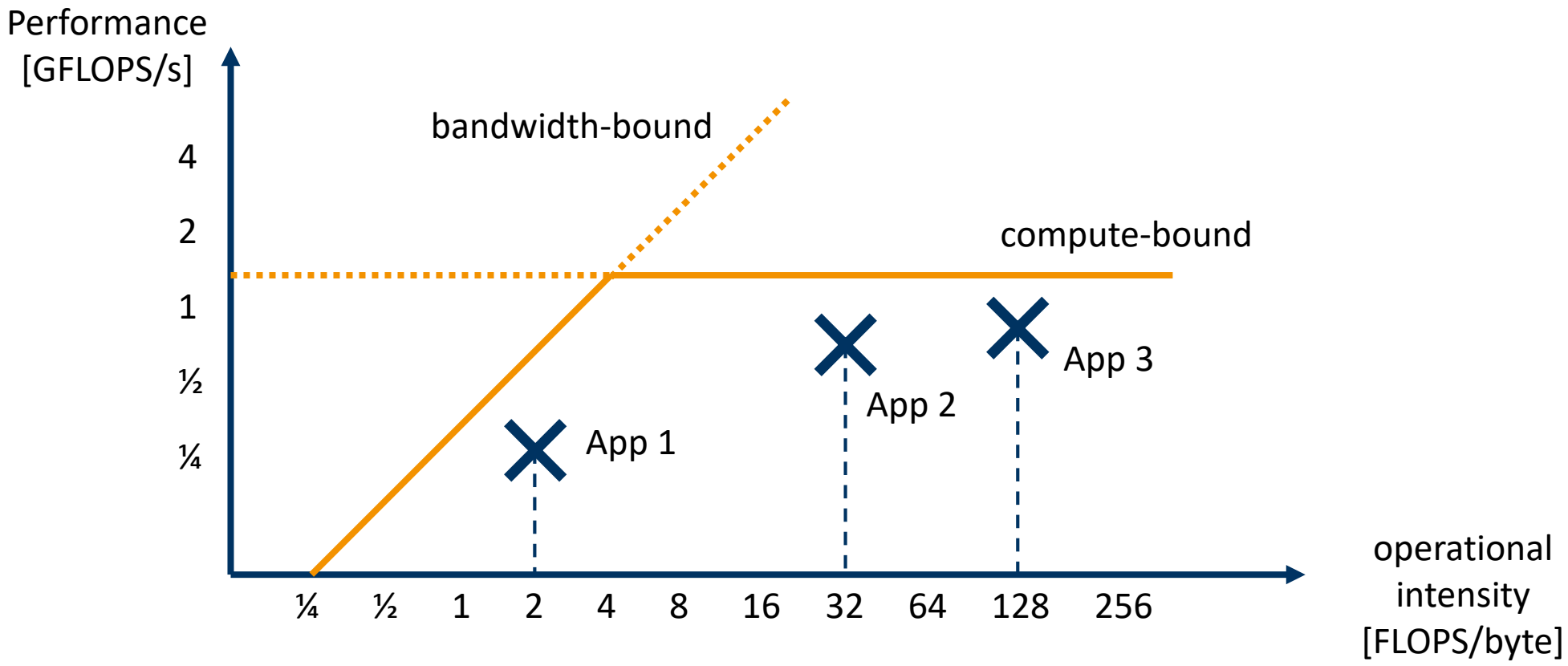
- law: $\text{speedup}_p = \frac{1}{r_s + \frac{r_p}{n}} = \frac{1}{\alpha + \frac{1-\alpha}{n}}$
- severely limits potential speedup, even for infinite parallelism
- example: $\alpha = 0.2$ (=20 % sequential)
  - ideal speedup on 8 processors is 3.33
  - ideal speedup on ∞ processors is 5



Strong vs. Weak Scalability

- *scalability* is (sort-of) a synonym for (good) speedup and efficiency
  - “program scales (linearly)” = program achieves linear speedup
- strong scalablity
  - how the program scales with a fixed problem size
- weak scalability
  - how the program scales when keeping the problem size proportional
  - important: how to scale the problem in proportion?

Roofline Model



How to Report Metrics? (Non-Exhaustive)

- How many repetitions of an experiment do I need to run?
  - A single run is enough, right?
  - Three runs is enough for averaging, right?
- Should I run on multiple systems?
  - Should their architecture differ?
- Should my experiments be reproducible?
  - What information is required to enable reproducibility?
- Do I show all collected data?
  - Do I select data, and if so, how?
  - Do I aggregate data, and if so, how?
  - Do I report absolute or relative values?
- Do I quantify the reliability of my data?
  - If so, how?
- …

Sequential Equivalence

- strong sequential equivalence
  - bitwise identical results
  - potentially big impact on performance
    (associativity, collective communication patterns, …)
  - requires preserving the order of computations compared to sequential version

- weak sequential equivalence
  - mathematically equivalent but not bitwise identical
    (IEEE 754 float arithmetic is neither associative, nor commutative)
  - does not require preserving the order of computations

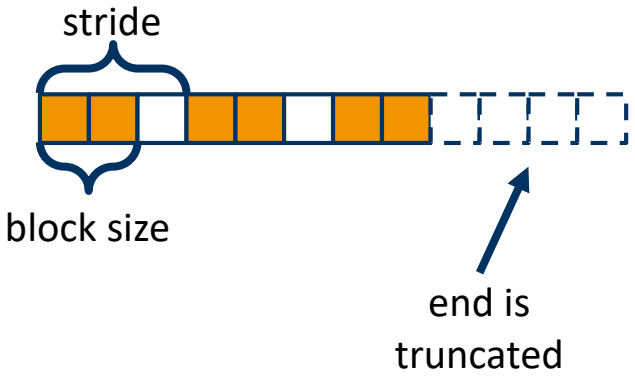
- Always check your requirements!
  - If your algorithm doesn’t require a specific order, why should its implementation?

MPI Derived Datatypes and Virtual Topologies

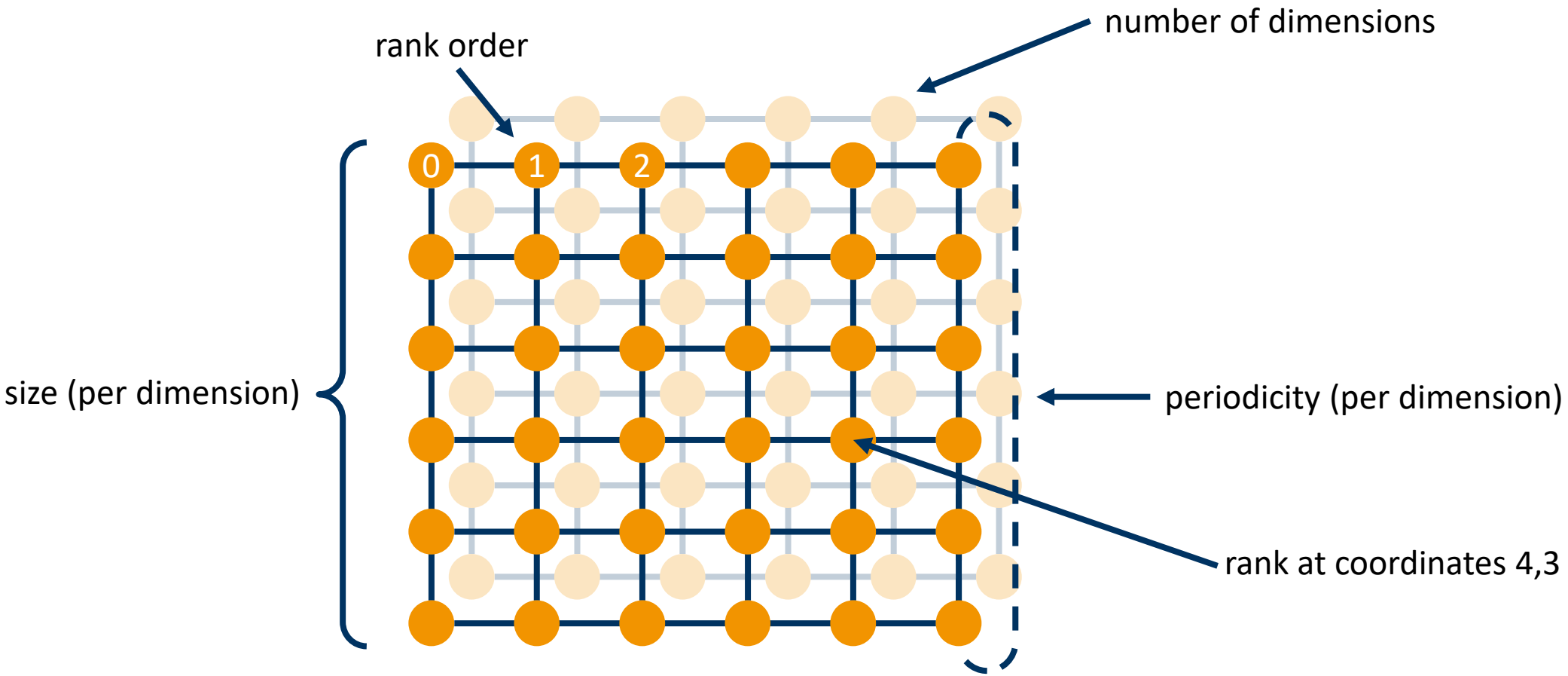
- derived datatypes
  - allows to send user-specific datatypes
- virtual topologies
  - adds semantic position information to ranks
- tales from the proseminar
  - off-topic topics

Selection of MPI Derived Datatype Facilities

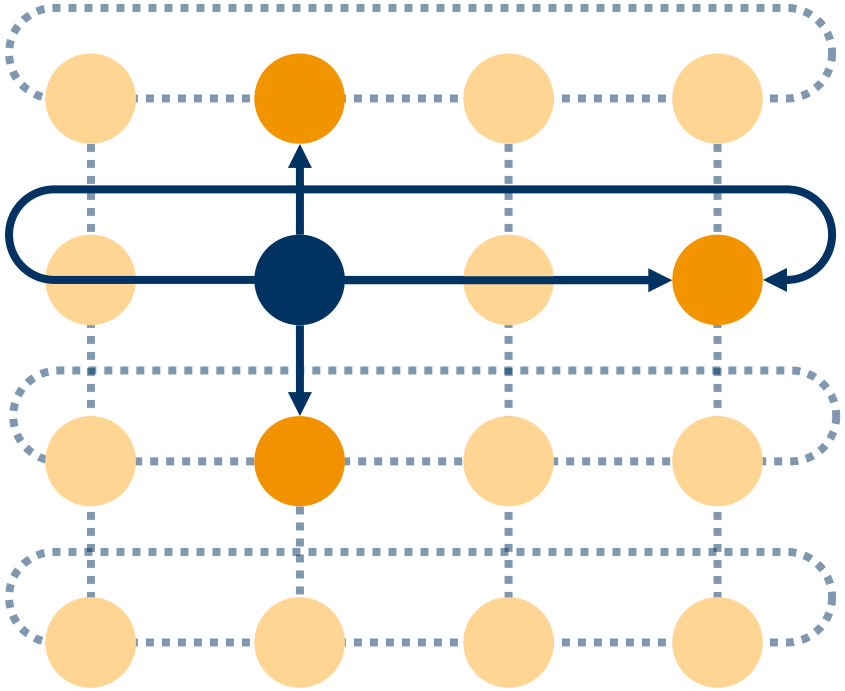
- `MPI_Type_create_struct(…)`
  - specifies the data layout of user-defined structs (or classes)
- `MPI_Type_vector(…)`
  - specifies strided data, i.e. same-type data with missing elements
- `MPI_Type_create_subarray(…)`
  - specifies sub-ranges of multi-dimensional arrays
- `MPI_Type_contiguous(…)`
  - specifies a user-defined contiguous type comparable to C arrays

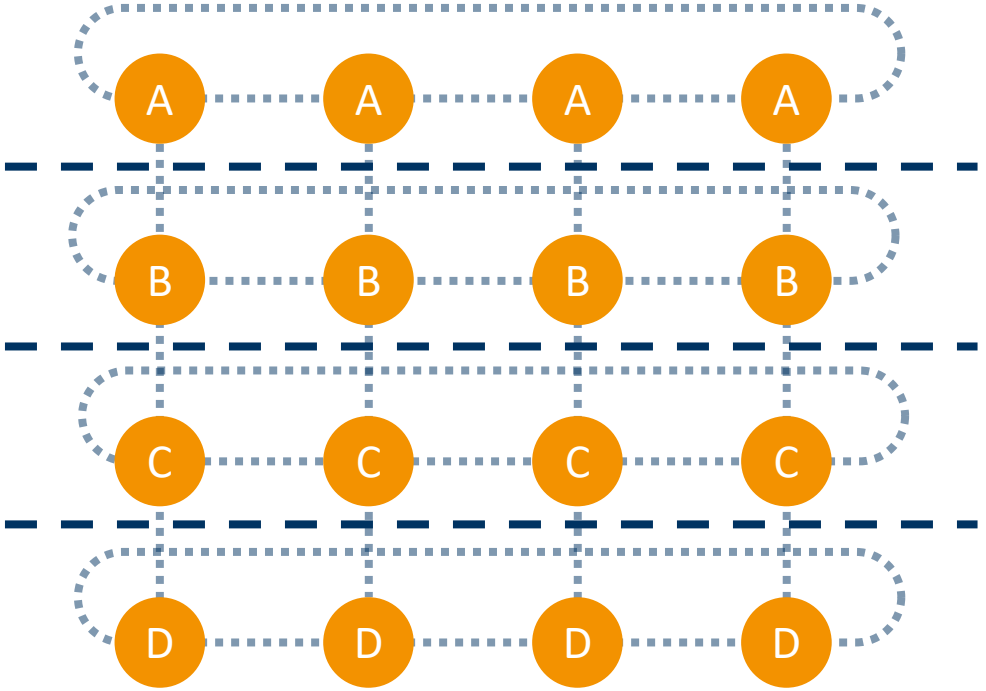


Properties of Cartesian Topologies



Shifting & Slicing

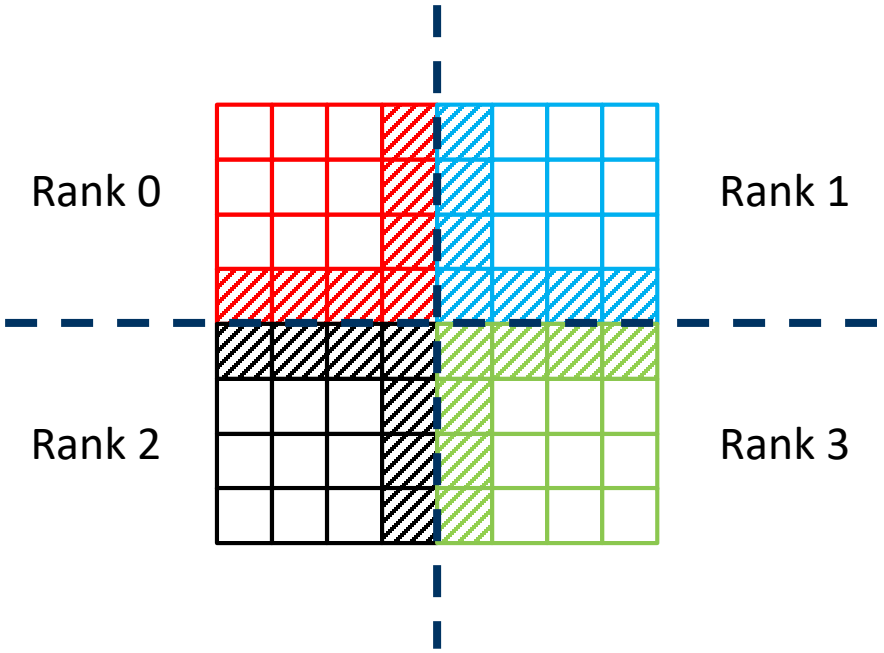


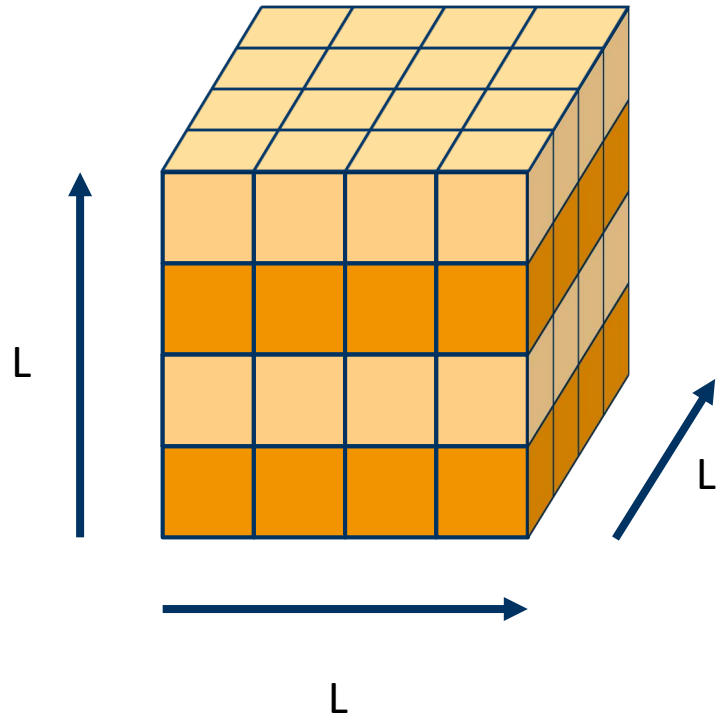


Verification & Validation

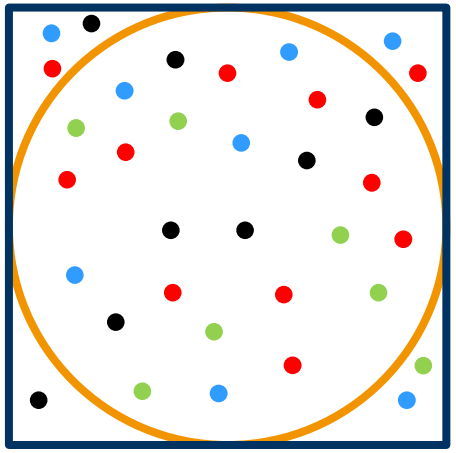
- absolutely not the same thing, though often used synonymously
- verification means checking your implementation
  - ensure that implementation meets the specification
  - check that software output is correct
- validation means checking your specification
  - ensure that the specification meets requirements
  - check that software output serves the use case purpose

Domain Decomposition & Ghost Cell Exchange

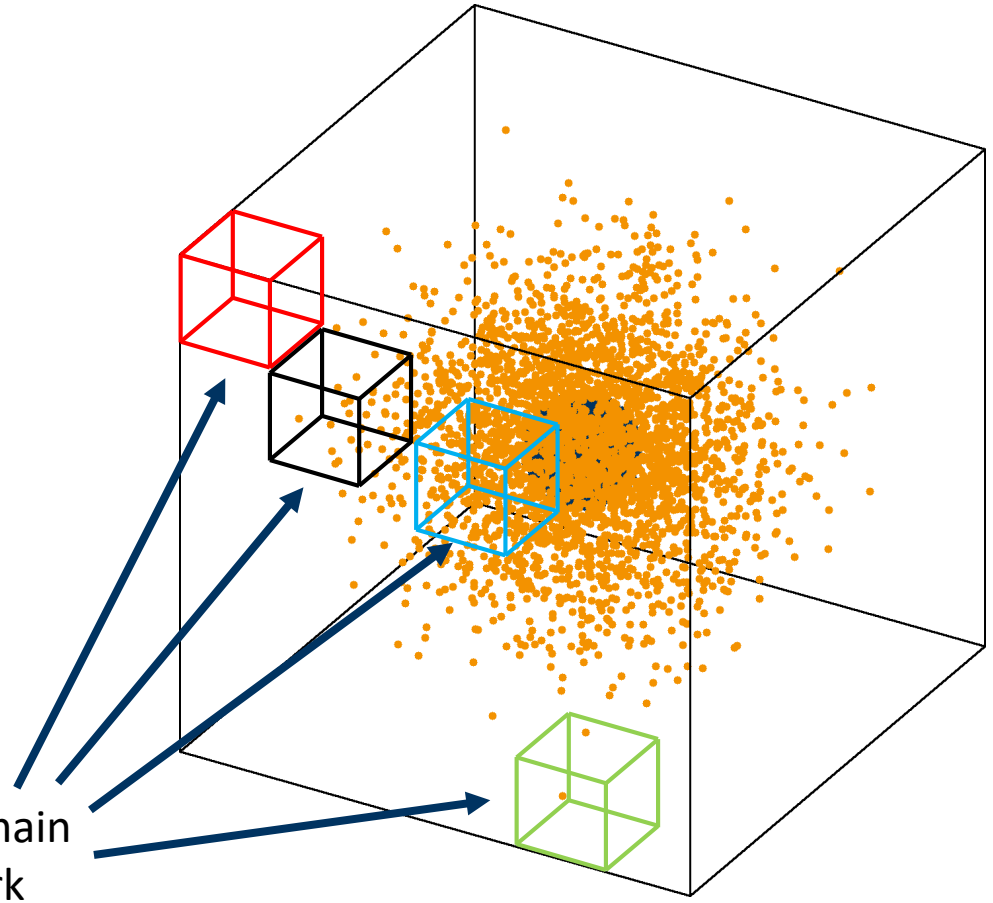




Domain Decomposition & Load Balancing



- Rank 0
- Rank 1
- Rank 2
- Rank 3



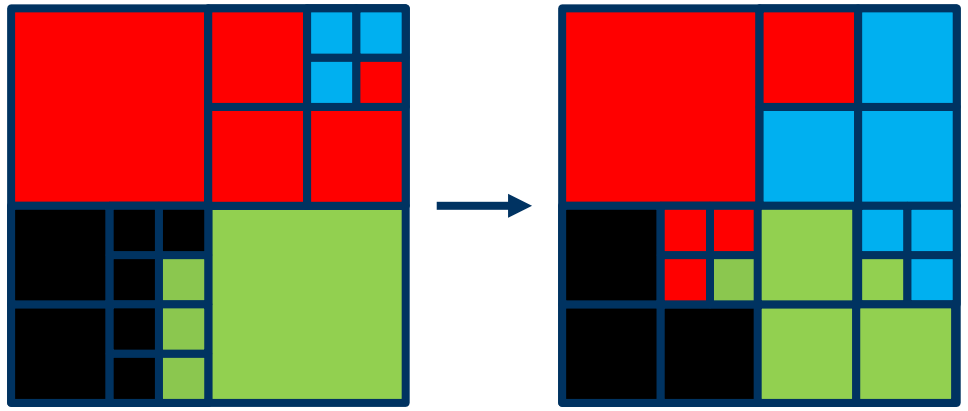
same-sized sub-regions of domain
but different amount of work

Static vs. Dynamic Load Imbalance

- static load imbalance
  - caused by initial conditions, e.g.
    - mountains vs. plains
    - ocean shore vs. open sea
    - remainder in integer division
  - does not change during application execution
  - mitigation usually incurs no runtime overhead after initial setup
- dynamic load imbalance
  - caused by application execution
    - moving particles (e.g. galaxy clusters)
    - partial availability of sensor data
    - convergence of iterative algorithms
  - does change during application execution
  - requires rebalancing (e.g. at fixed intervals, when reaching limits, …)
    - definitely incurs runtime overhead

Dealing with Load Imbalance: Dynamic Case

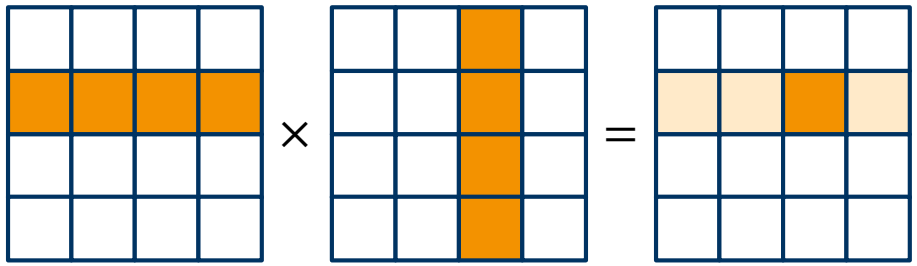
- dynamic
  - some form of repeated balancing required, e.g.
    - at certain intervals
    - when reaching certain thresholds
  - use e.g. worker queues and work stealing

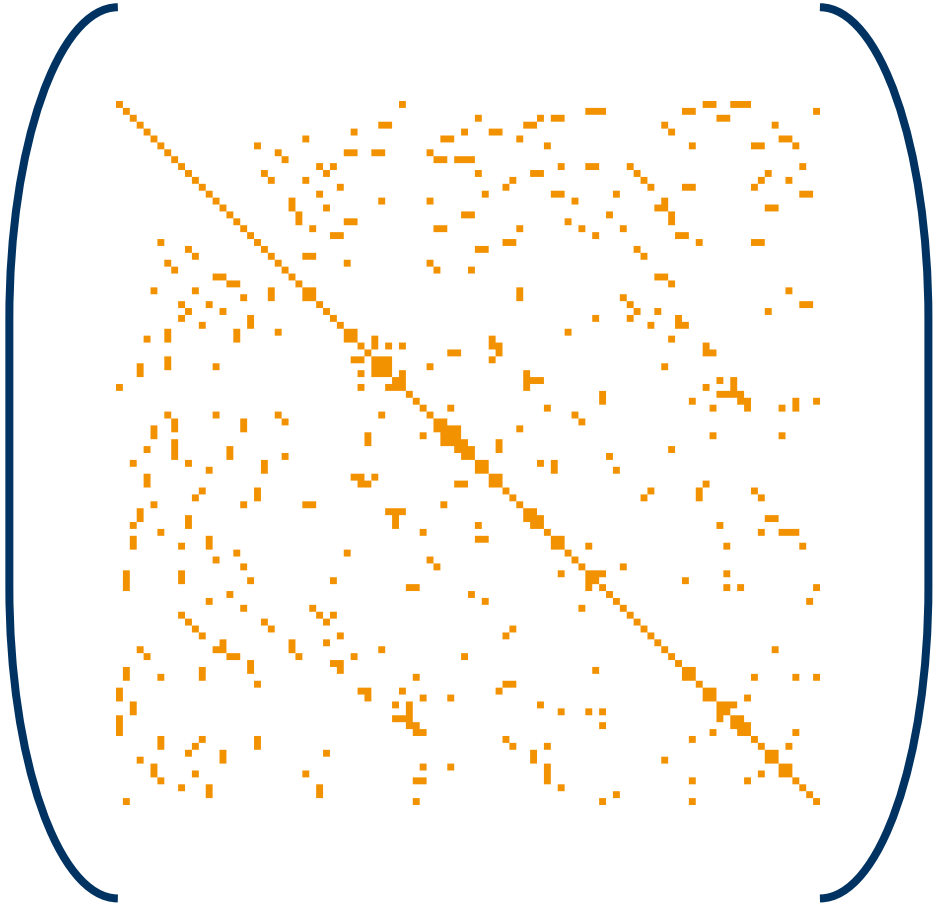


13 Dwarfs of HPC

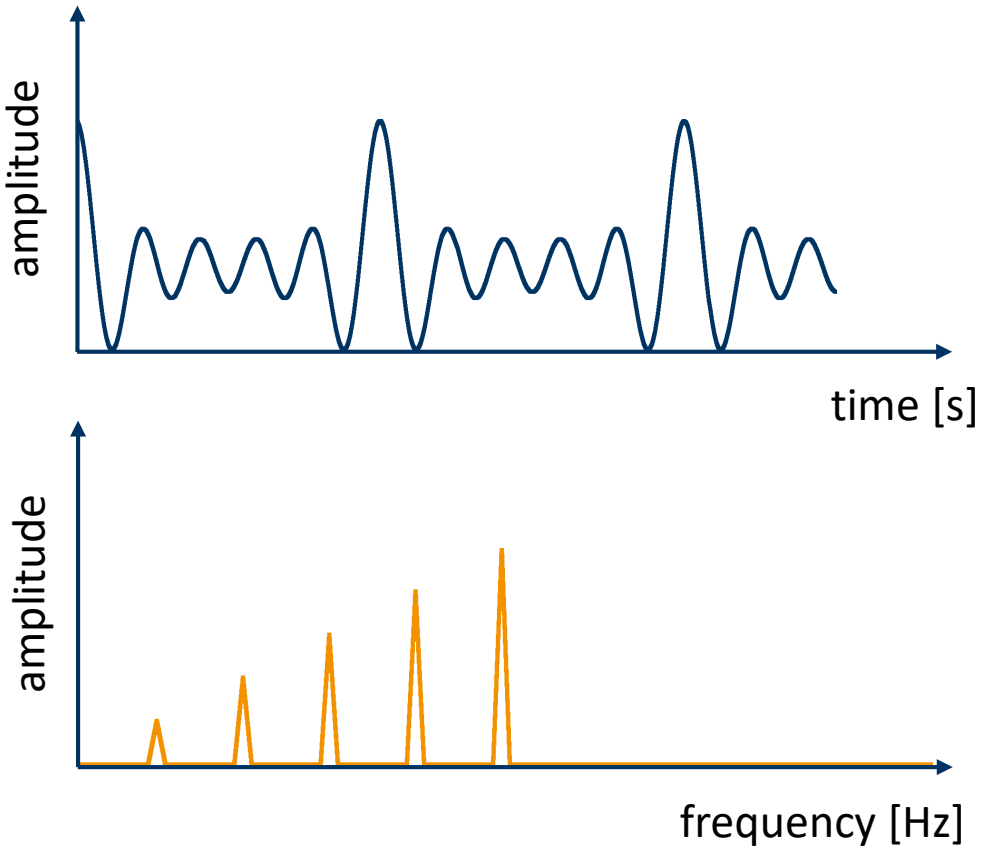
- 1. Dense Linear Algebra
- 2. Sparse Linear Algebra
- 3. Spectral Methods
- 4. N-body Methods
- 5. Structured Grids
- 6. Unstructured Grids
- 7. Monte Carlo Methods
- 8. Combinational Logic
- 9. Graph Traversal
- 10. Dynamic Programming
- 11. Backtrack & Branch+Bound
- 12. Graphical Models
- 13. Finite State Machine

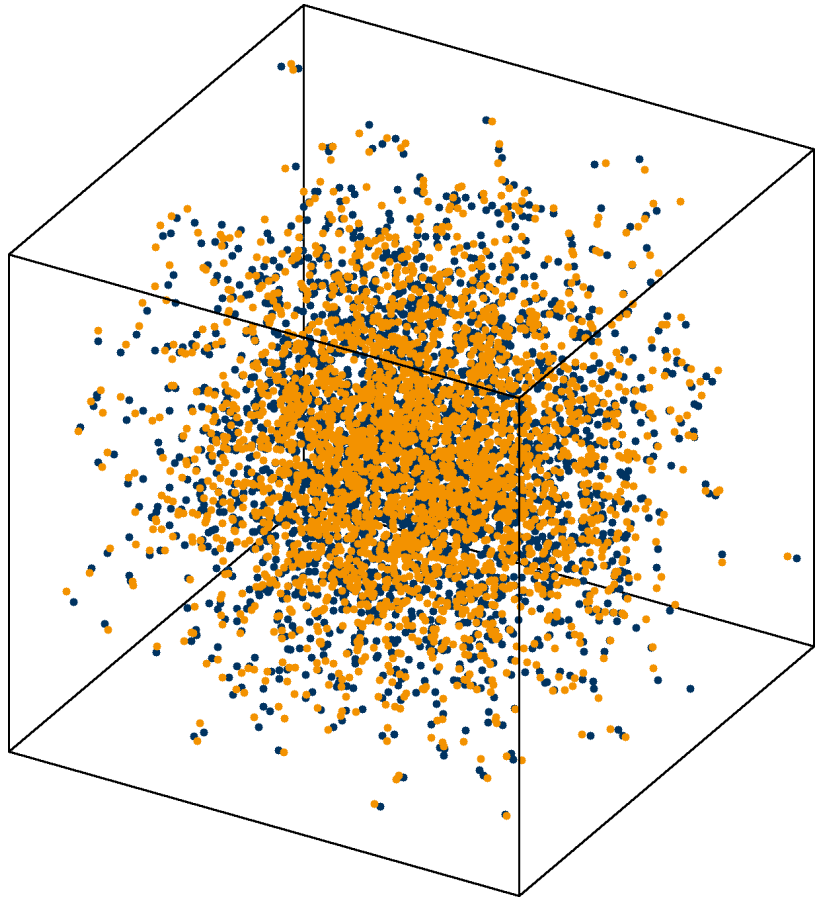
Dense and Sparse Linear Algebra



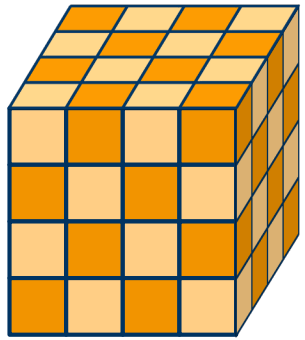


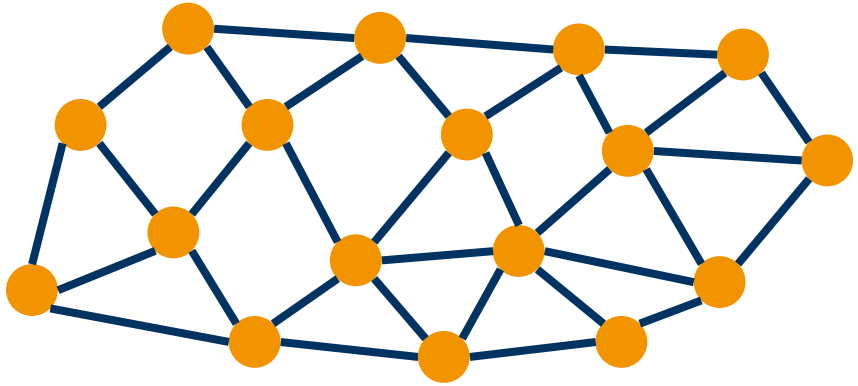
Spectral & N-body Methods

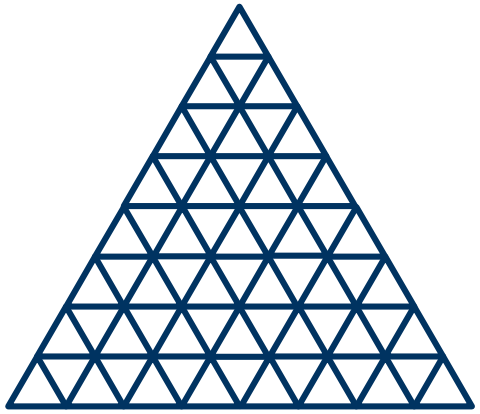


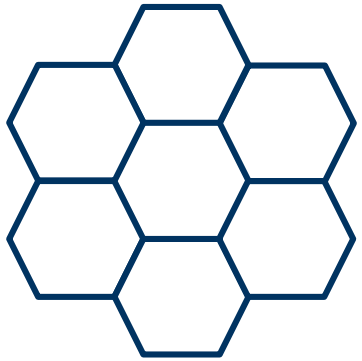


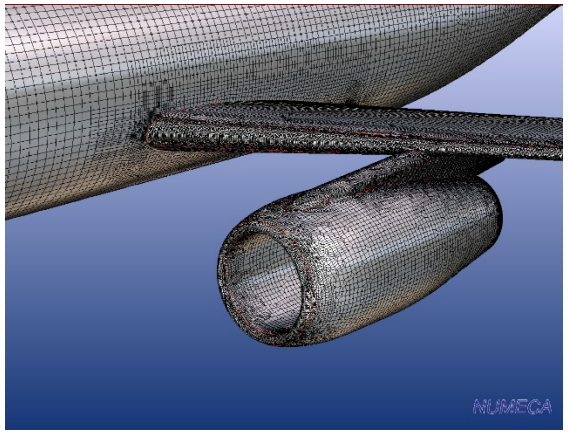
Structured & Unstructured Grids



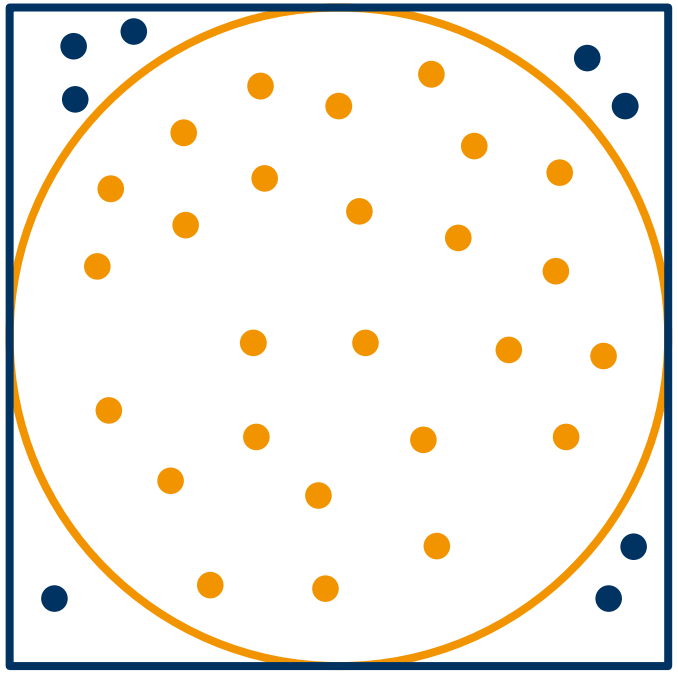


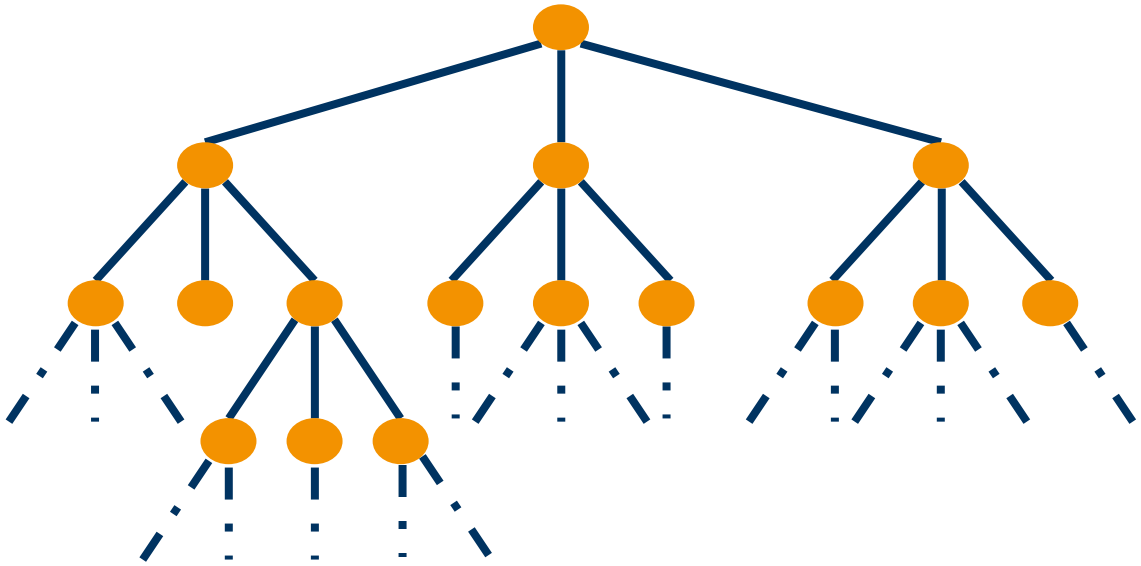






Monte Carlo Methods & Graph Traversal



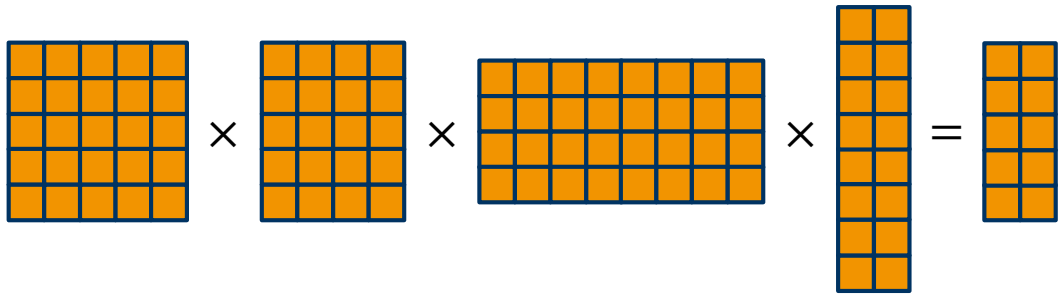


Combinational Logic

- generally involves performing simple operations on large amounts of integer data
  - e.g. computing cyclic redundancy codes (CRC)
- often parallelizable on multiple levels
  - bit-level parallelism
  - block-level parallelism

```
uint8_t compute(uint8_t const msg[], int n) {
    uint8_t rem = 0;
    for (int byte = 0; byte < n; ++byte) {
        rem ^= (msg[byte] << (WIDTH - 8));
        for (uint8_t bit = 8; bit > 0; --bit) {
            if (rem & TOPBIT) {
                rem = (rem << 1) ^ POLYNOMIAL;
            } else {
                rem = (rem << 1);
            }
        }
    }
    return (rem);
}
```

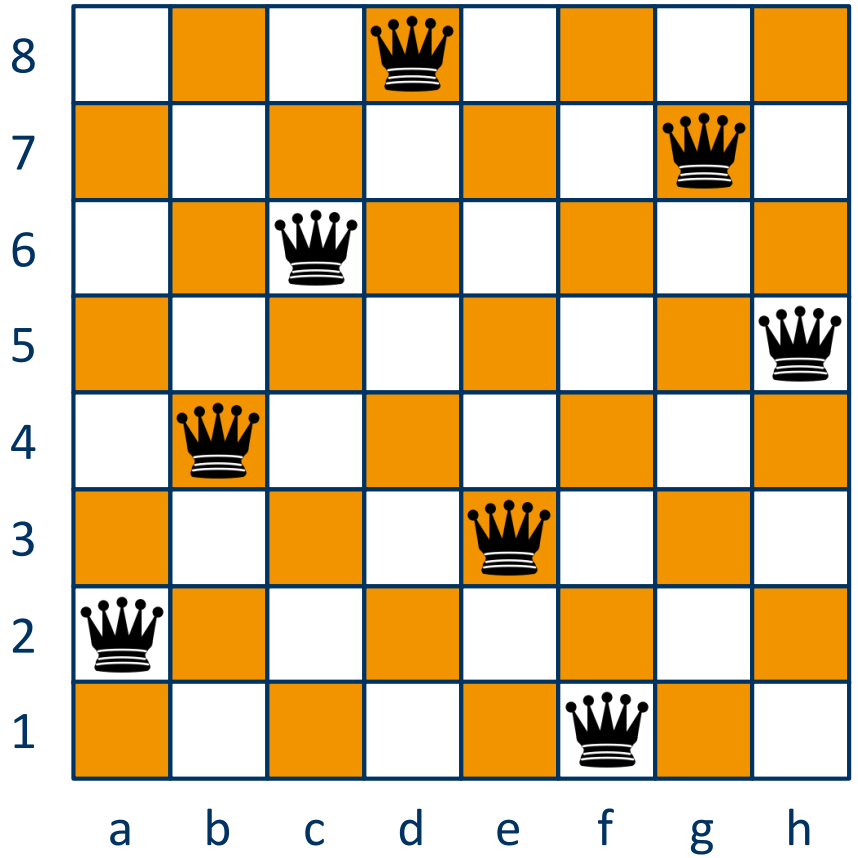
Dynamic Programming & Backtracking/Branch+Bound



$$A \times \big(B \times (C \times D)\big) = X \quad \text{154 ops}$$

$$(A \times B) \times (C \times D) = X \quad \text{204 ops}$$

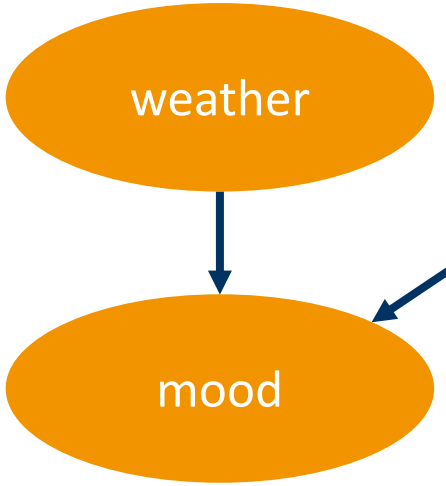
$$\big((A \times B) \times C\big) \times D = X \quad \text{340 ops}$$



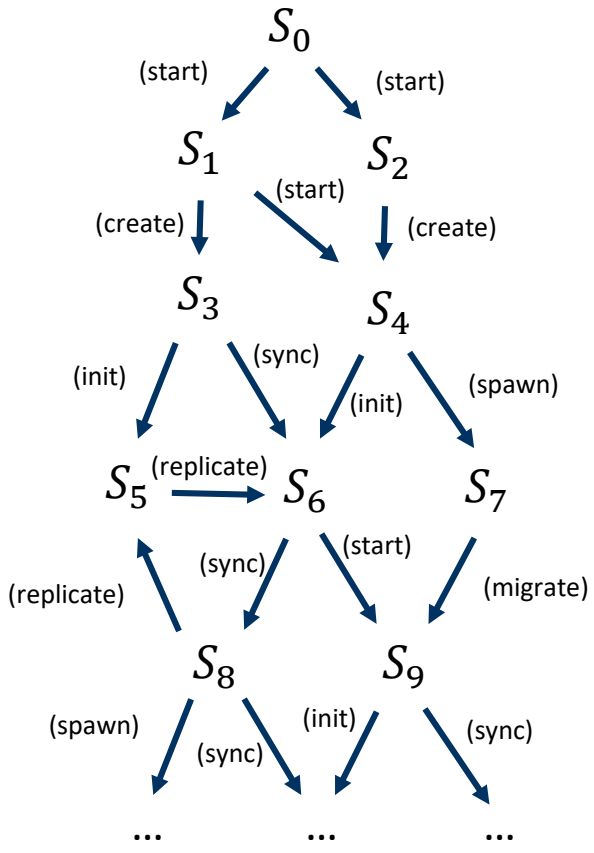
Graphical Models & Finite State Machines

sun	rain
0.40	0.60

edible	inedible
0.90	0.10

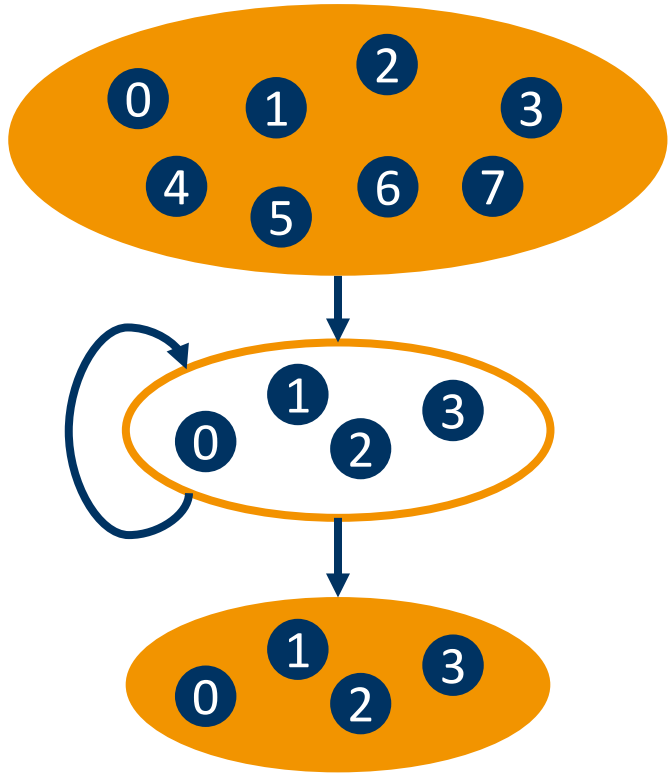


	good	bad
sun, edible	0.95	0.05
sun, inedible	0.70	0.30
rain, edible	0.75	0.25
rain, inedible	0.10	0.90

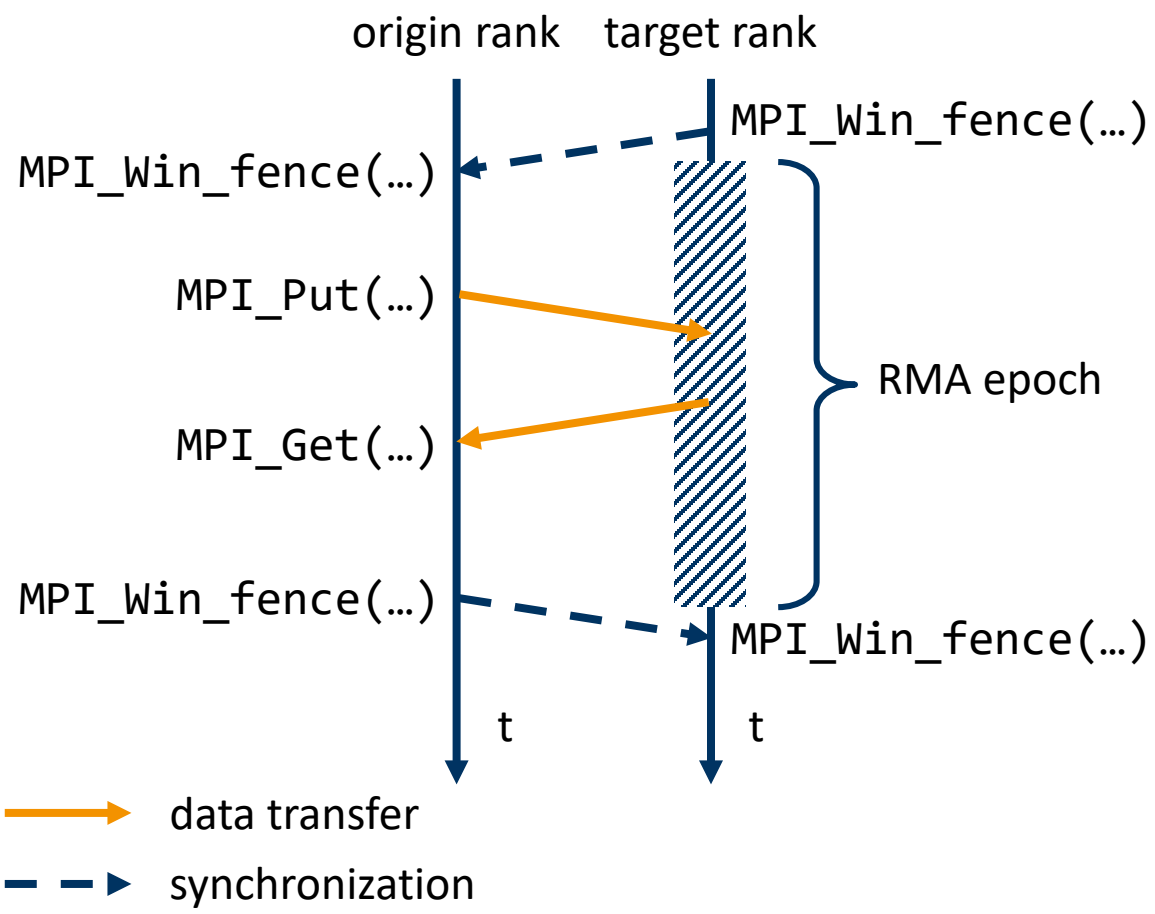


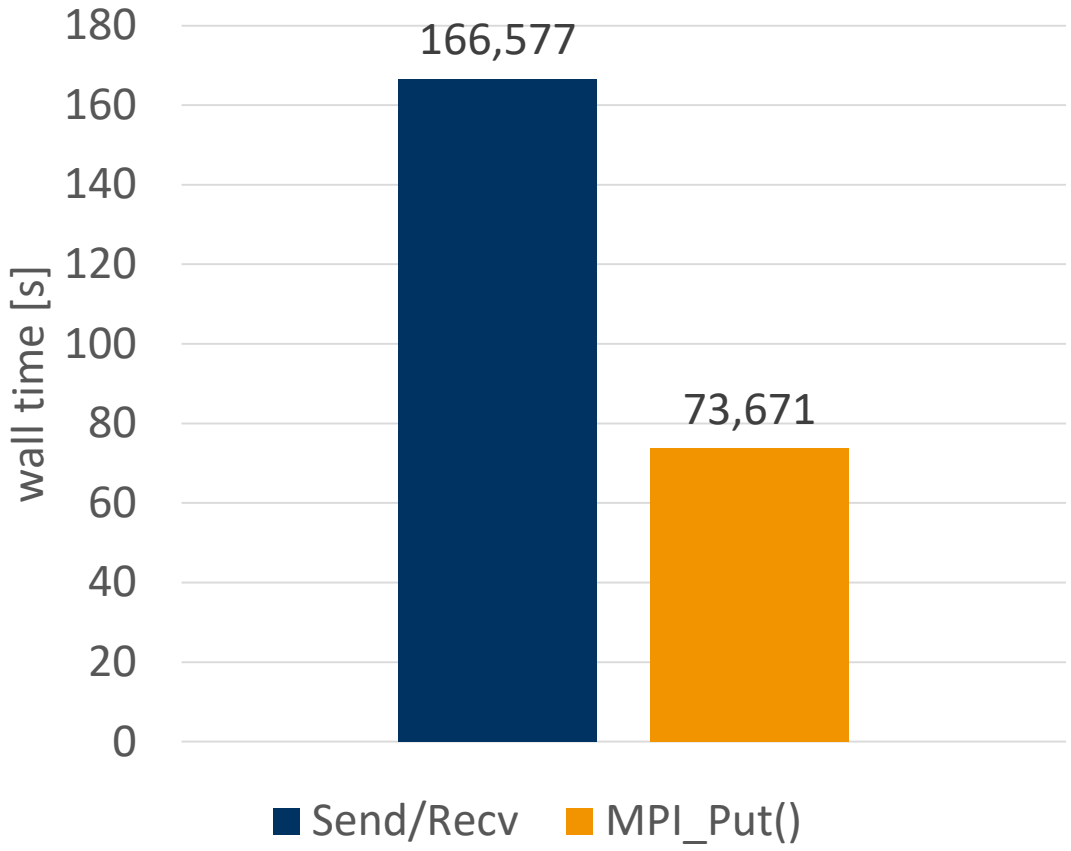
MPI Groups, Communicators and One-Sided Communication

- communicators and groups
- one-sided communication
- error handling



MPI One-sided Communication

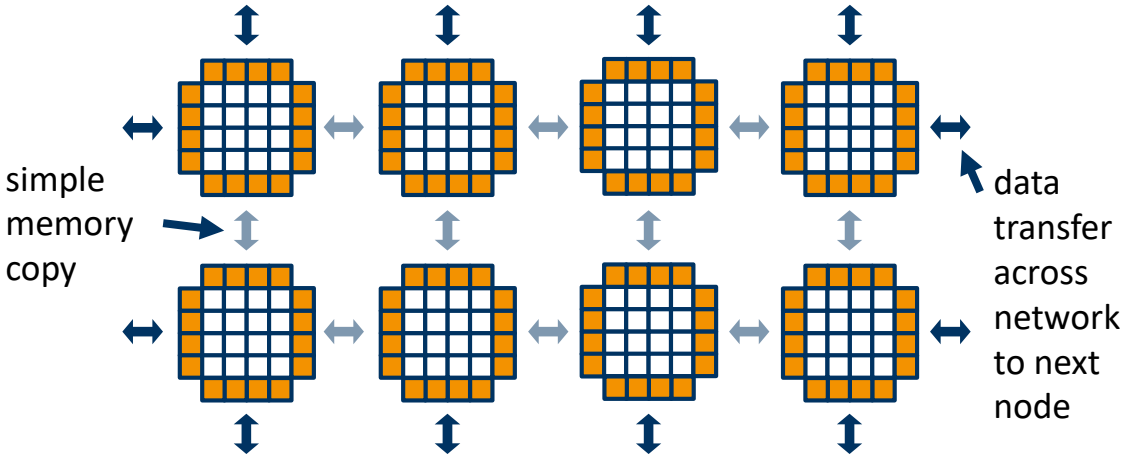


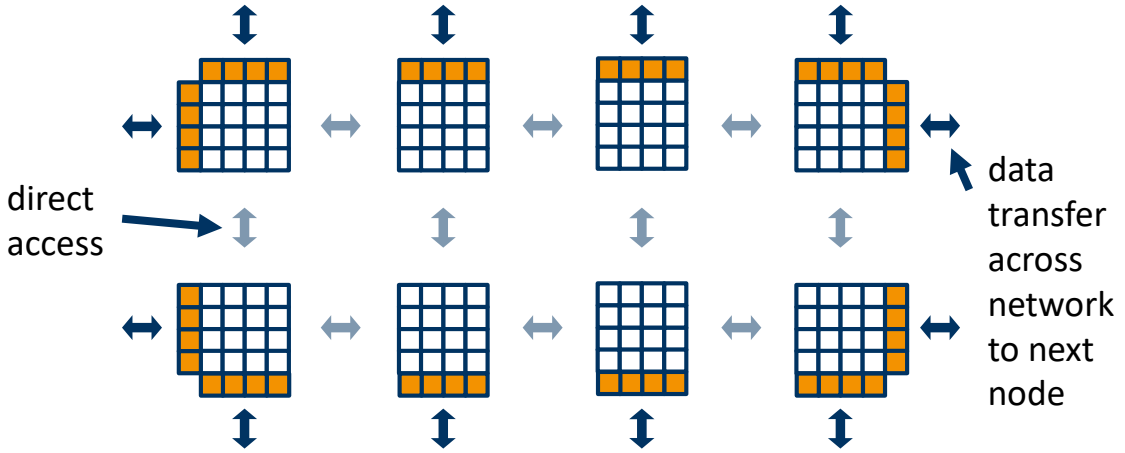


Implications of One-sided Communication

- several benefits
  - allows dynamic access patterns (e.g. when target rank does not know number and ranks of origins)
  - reduce synchronization overhead for multiple data transfers
  - reduce management overhead on receiver side (e.g. tag matching)
  - performance gain
  - reduce coding effort on receiver side
- drawbacks
  - no send/receive matching
  - operations are not explicitly visible on the receiver side
  - user often responsible for correct order of reads/writes (race conditions)
  - only non-blocking communication

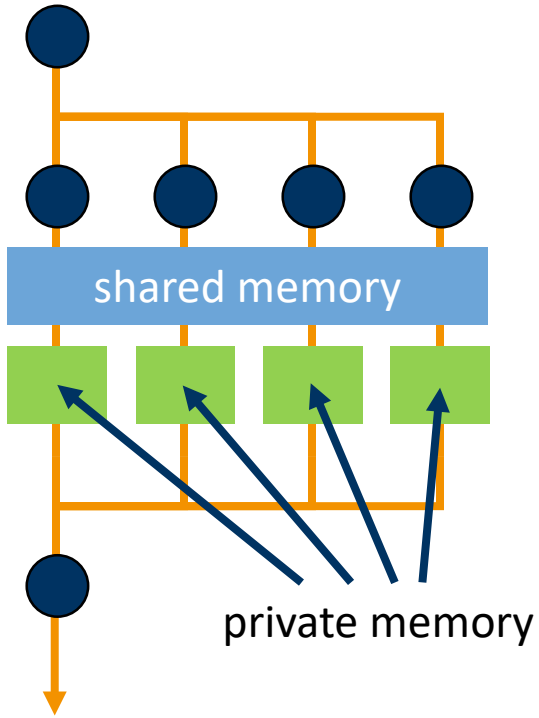
Ghost Cell Exchange (Message Passing vs. Shared Memory Access)





OpenMP Basics

- main characteristics
- programming, execution and memory models
- directives
- Tales From the Proseminar



OpenMP API cont'd

- pragmas
  - control constructs
    - parallelism & work sharing
  - data sharing
    - private & shared variables, initialization
  - synchronization
    - critical & atomic sections, barriers
- library functions
  - querying/controlling environment
  - timing
  - locking
- environment variables
  - degree and nesting of parallelism
  - loop scheduling
  - thread mapping and binding

Work Sharing Directives

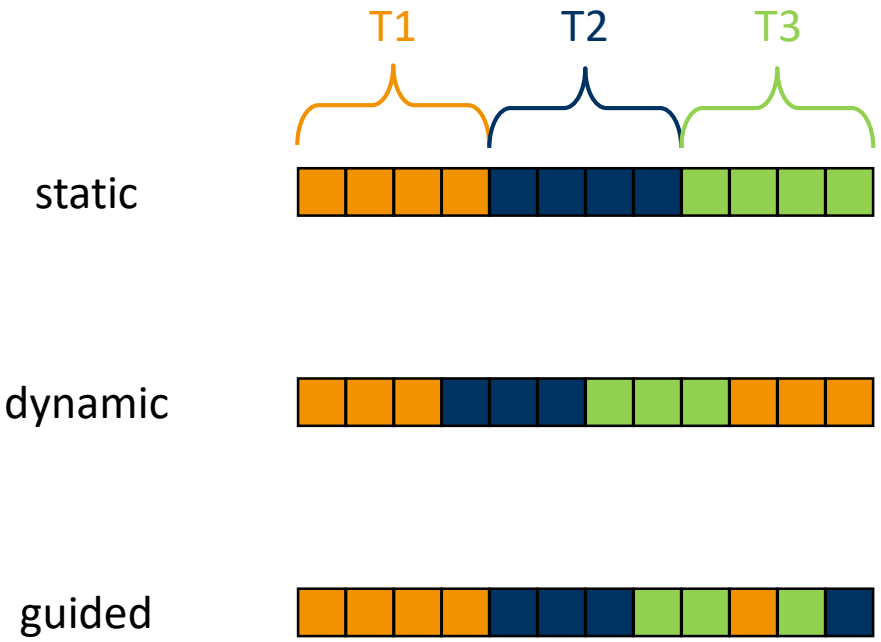
- distribute execution of following code region among existing threads of the team
- must be enclosed in parallel region
- cannot be directly nested
- do not launch new threads
- no barrier on entry
- implicit barrier on exit
  - unless `nowait` clause

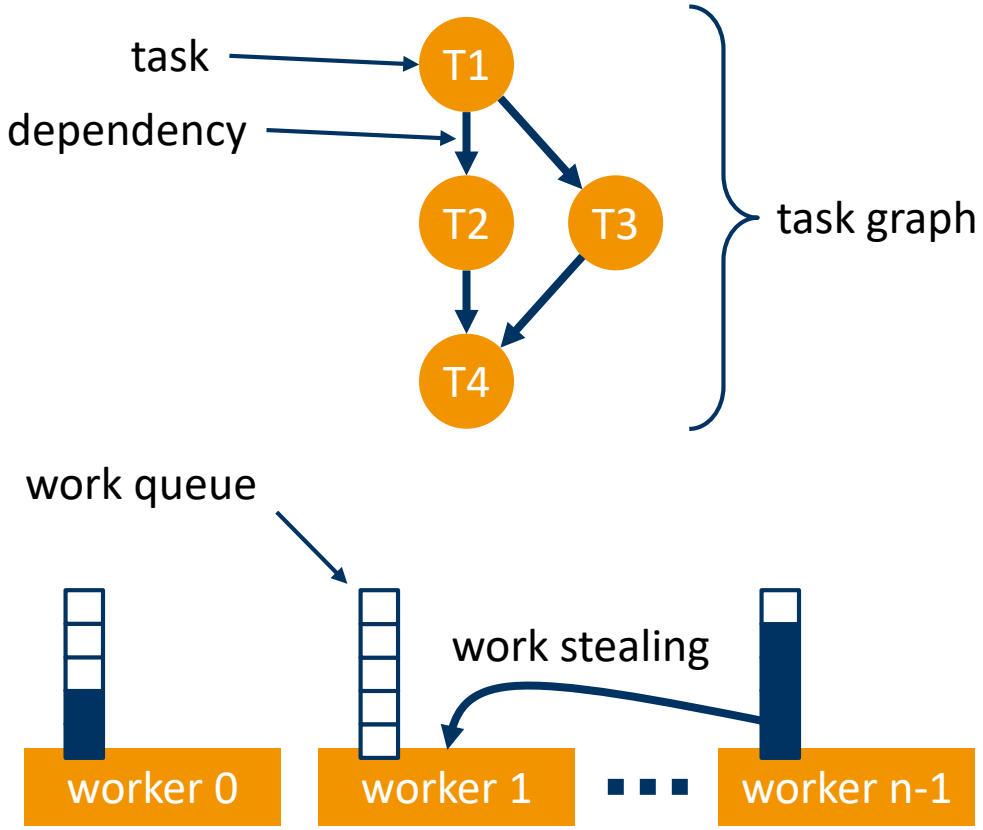
- `for`
- `sections`
- `single`
- `task`
- `simd`

OpenMP Advanced

- task-based parallelism
- modern OpenMP features
  - affinity
  - vectorization
  - accelerators
- common OpenMP pitfalls

Data & Task-based Parallelism

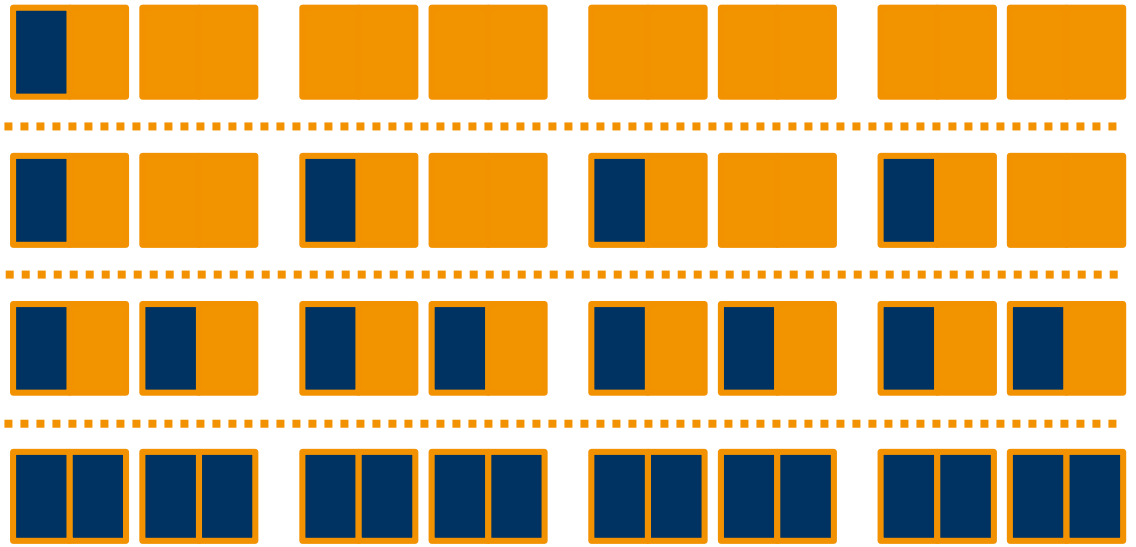




OpenMP Nested Affinity

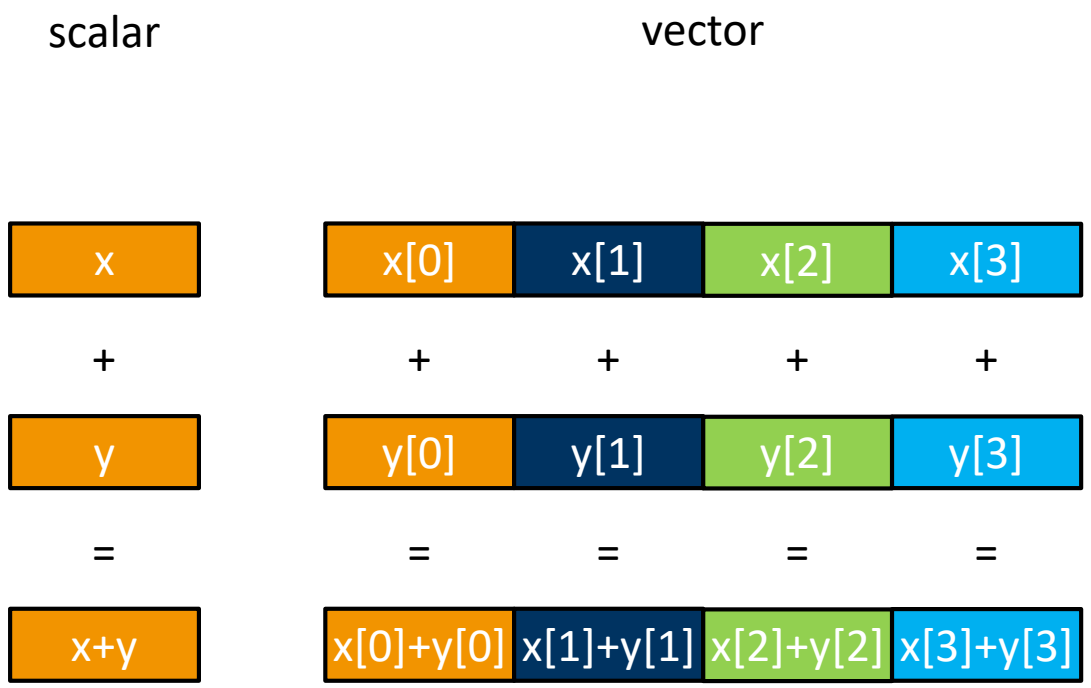
```
// OMP_PLACES=threads
// OMP_NUM_THREADS=4,2,2
// OMP_PROC_BIND=spread,spread,close

#pragma omp parallel
{
  #pragma omp parallel
  {
    #pragma omp parallel
    {
      // work ...
}}}
```

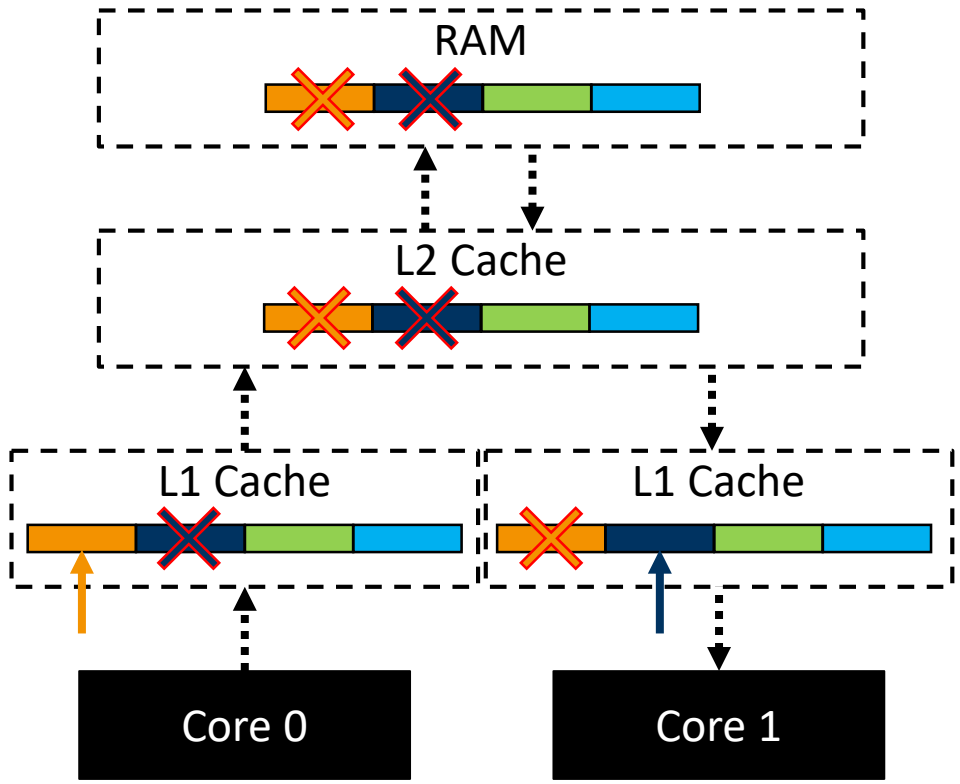


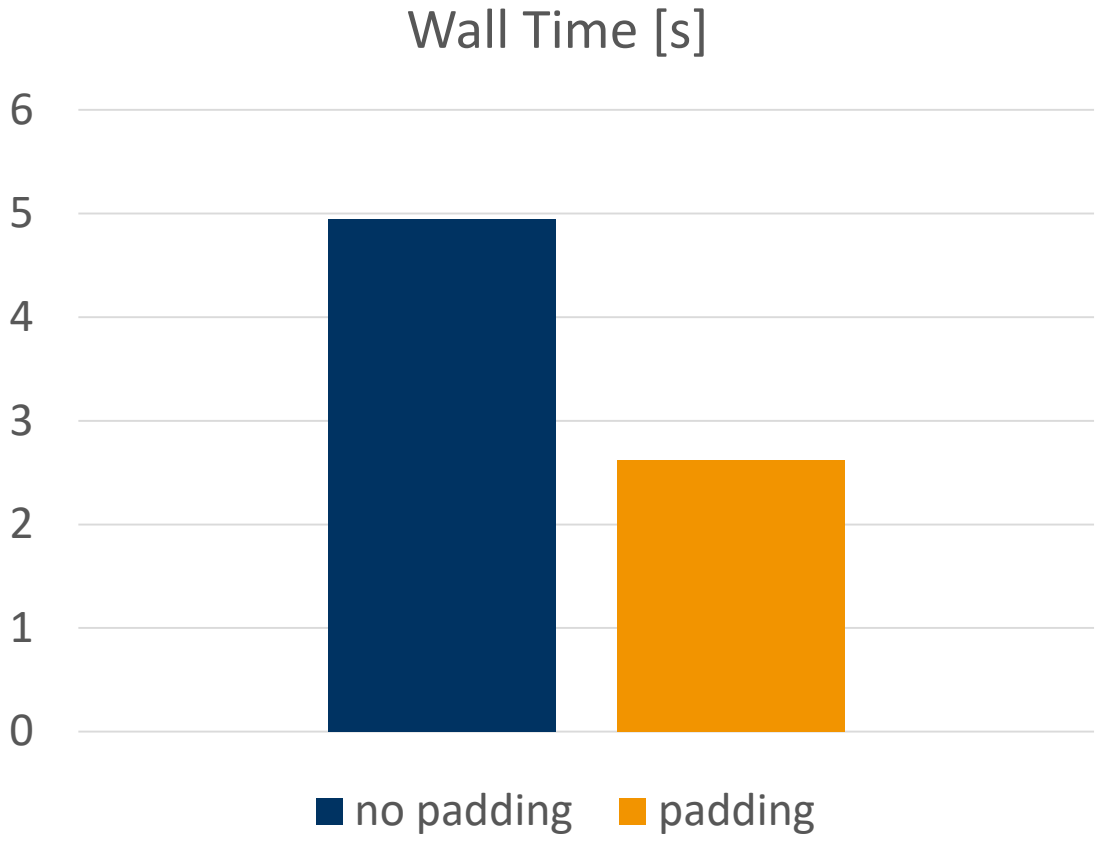
Vectorization

- modern CPUs have vector units
  - allow multiple operands per operation
  - performance gains of up to e.g. 4x without any thread- or process-based parallelism
  - Intel/AMD MMX/SSE/AVX, ARM NEON, IBM AltiVec, …
- available operations and number of operands (“vector width”) depend on your software/hardware stack
  - hard to code manually (compiler intrinsics or assembler)

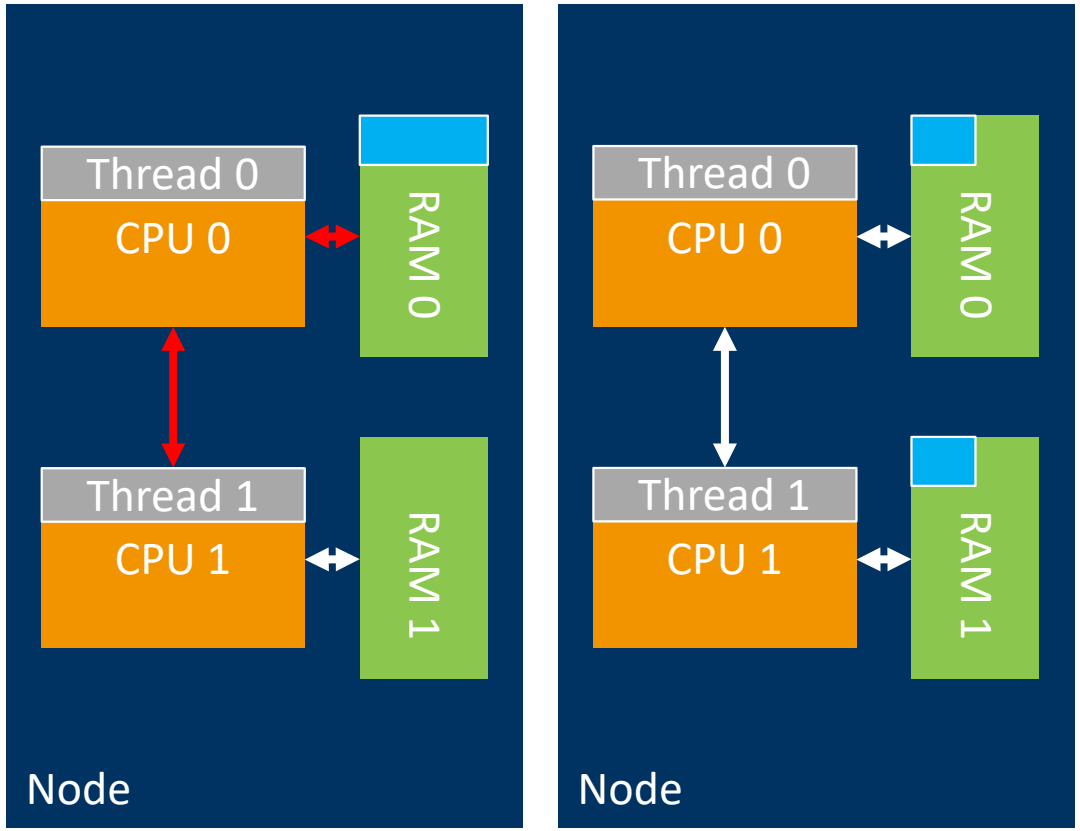


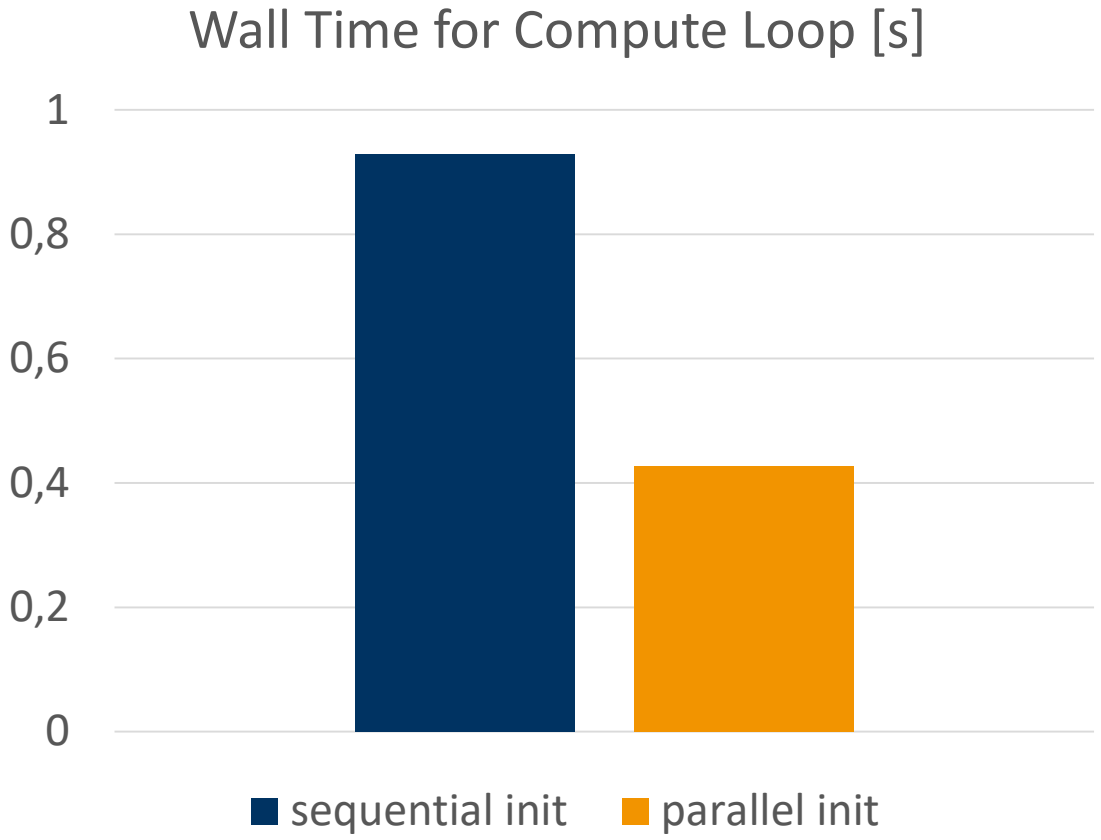
False Sharing





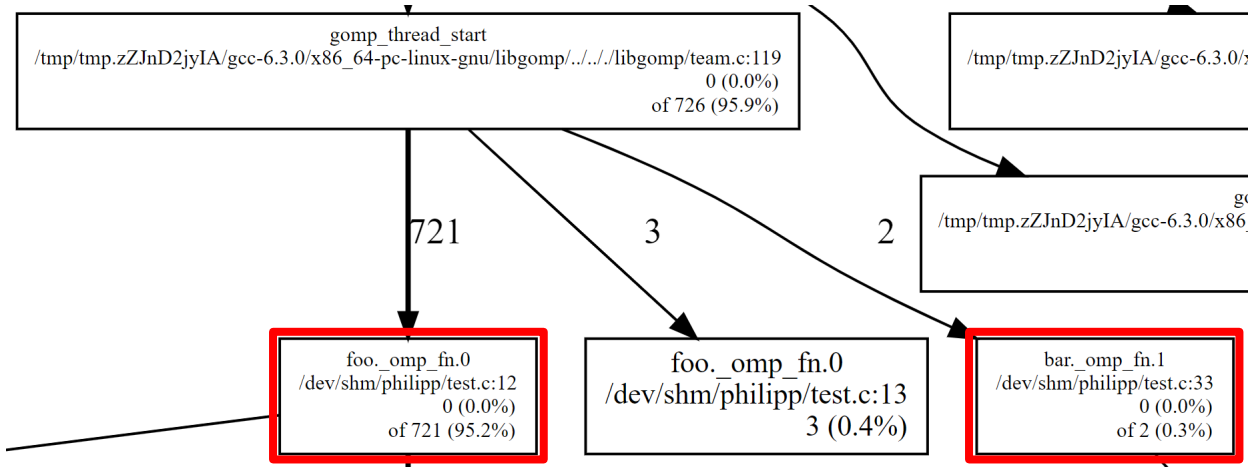
Sequential vs. Parallel Initialization on NUMA (“First Touch”)





Debugging Parallel Programs

- functional debugging
  - generic guidelines
  - serial debugging
  - parallelism-specific debugging
- performance debugging
  - generic guidelines
  - serial debugging
  - parallelism-specific debugging



Coding Guidelines

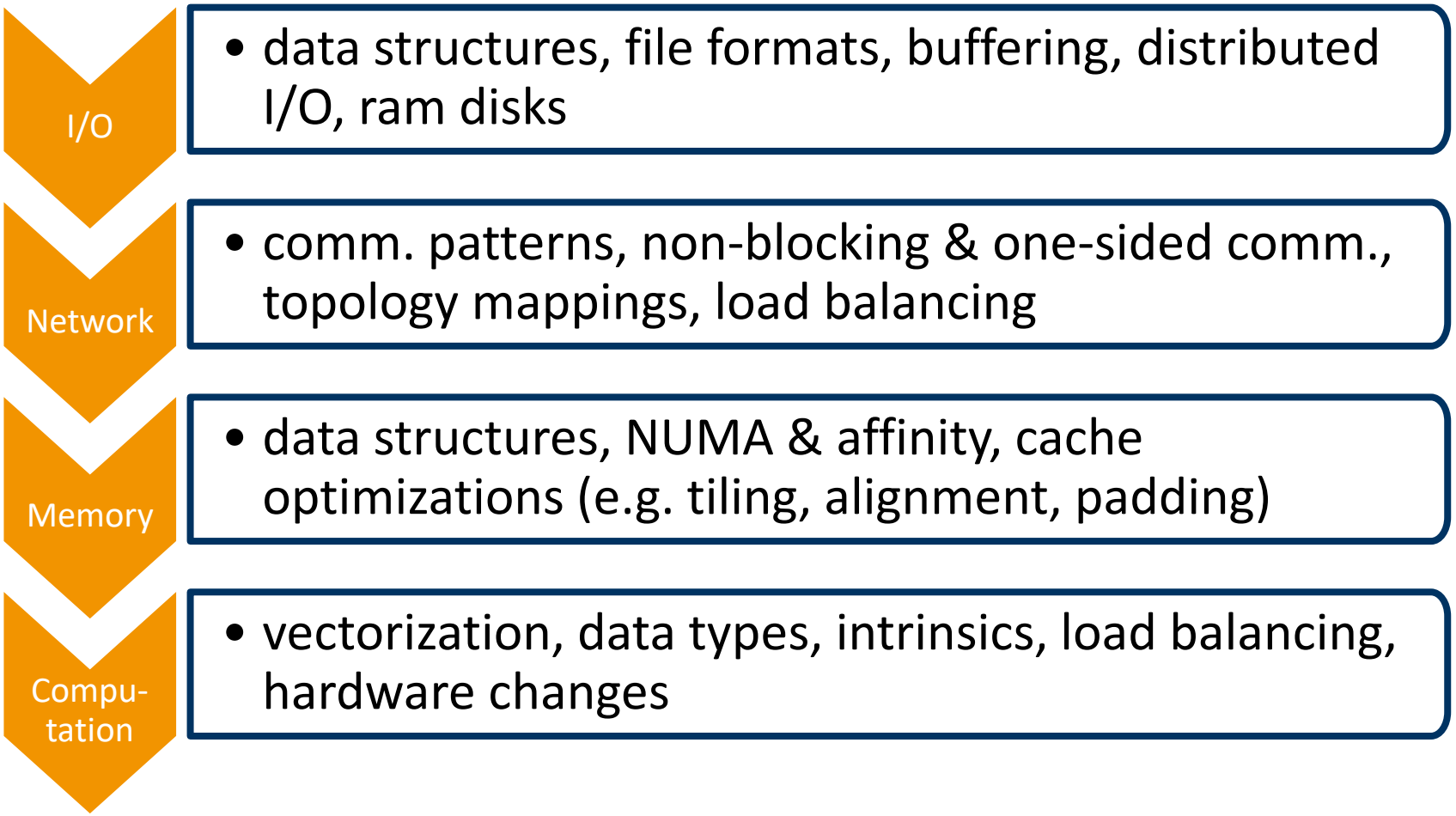
- write clean code that prevents bugs or facilitates their detection, e.g.
  - use meaningful identifiers
  - minimize vertical distance of variables
  - don't use OpenMP's `private`
  - follow the **D**on't **R**epeat **Y**ourself (DRY) principle (single component per feature)
  - …
- The toolchain you must use!
  - read & heed compiler warnings
  - write and regularly run unit and/or integration tests, especially aimed at (varying degrees of) parallelism
  - use code coverage tests
  - use continuous integration
  - use source version control

Generic Debugging Guidelines

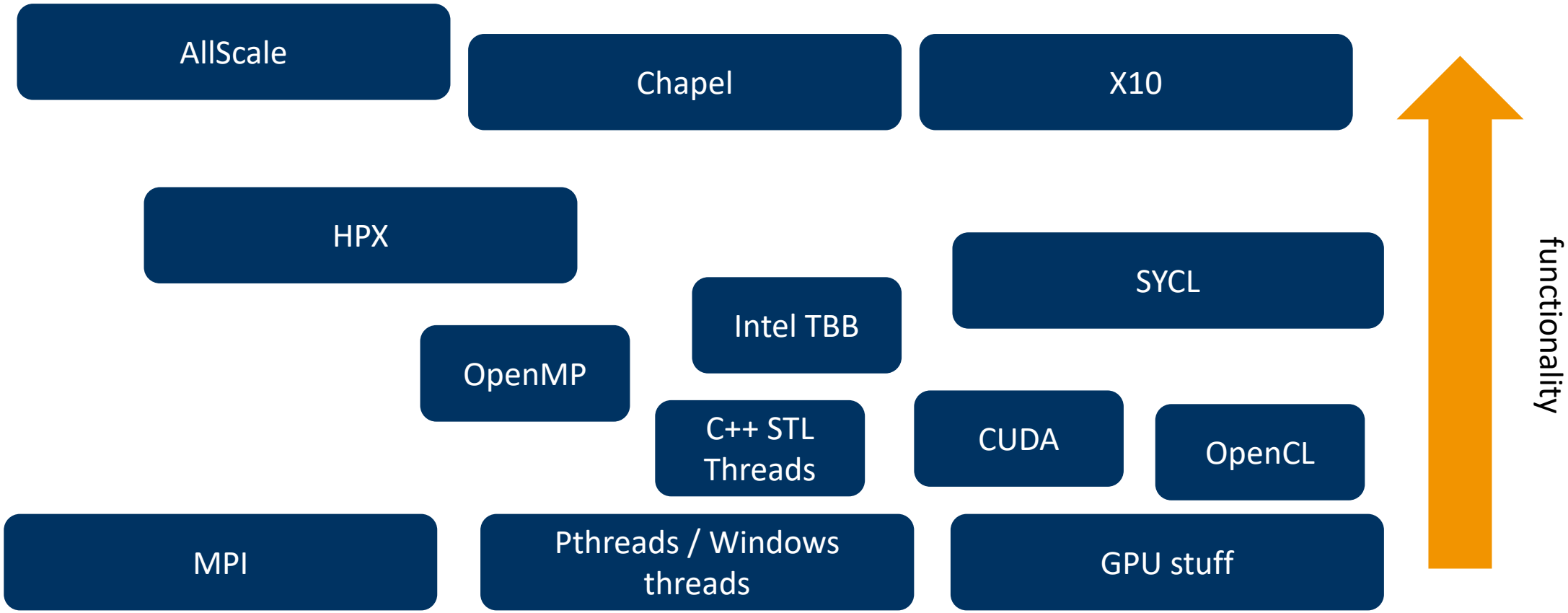
- create a **M**inimal **W**orking **E**xample (MWE)
  - minimize problem size
  - minimize software components/features involved
  - ensure/increase reproducibility
  - if parallel
    - minimize machine size (number of threads and/or ranks)
    - minimize complexity of parallel interaction (e.g. communication patterns, …)

- minimizes debugging feedback cycles times, amount of memory to inspect, amount of code to consider, overall degree of complexity of component & parallel interaction
  - sounds simple, but don't underestimate this
  - every change along the way to an MWE gives you more information about the problem

Points of Attack in Order of Benefit



Expanding Horizons: Additional Programming Models



How to Categorize Programming Models

- type of API/user interface
  - language? language extension? library?
- domain specificity
  - single use-case only? generic?
- target platform
  - distributed memory? shared memory? accelerators?
- features
  - intra-node scheduling? inter-node scheduling? data decomposition? data distribution? work decomposition? work distribution? fault tolerance? nested parallelism? …

Overview of State-of-the-art Programming Models

	Architectural			Task System				Management				Eng.	
	Communication Model	Distributed Memory	Heterogeneity	Graph Structure	Task Partitioning	Result Handling	Task Cancellation	Worker Management	Resilience Management	Work Mapping	Synchronization	Technological Readiness	Implementation Type
C++ STL	smem	×	×	dag	×	i/e	×	i	×	i/e	e	9	Library
TBB	smem	×	×	tree	×	i	✓	i	×	i	i	8	Library
HPX	gas	i	e	dag	✓	e	✓	i/e	×	i/e	e	6	Library
Legion	gas	i	e	tree	✓	e	×	i	×	i/e	e	4	Library
PaRSEC	msg	e	e	dag	×	e	✓	i	✓	i/e	i	4	Library
OpenMP	smem	×	i	dag	×	i	✓	e	×	i	i/e	9	Extension
Charm++	gas	i	e	dag	✓	i/e	×	i	✓	i/e	e	6	Extension
OmpSs	smem	×	i	dag	×	i	×	i	✓	i	i/e	5	Extension
AllScale	gas	i	i	dag	✓	i/e	×	i	✓	i	i/e	3	Extension
StarPU	msg	e	e	dag	✓	i	×	i	×	i/e	e	5	Extension
Cilk Plus	smem	×	×	tree	×	i	×	i	×	i	e	8	Lang.
Chapel	gas	i	i	dag	✓	i	×	i	×	i/e	e	5	Lang.
X10	gas	i	i	dag	✓	i	×	i	✓	i/e	e	5	Lang.

Considerations for Application Developers

- Which platform should I target?
  - shared memory, distributed memory, accelerators, ...
- Which features and degree of control do I need?
  - automatic work and data decomposition and distribution, scheduling, fault tolerance, ...
  - porting legacy codes might constrain you to using libraries (or possibly language extensions)
- How much support do I need?
  - consider maturity and long-term support of the language/extension/library
- Which programming language do I know?
  - check libraries or embedded DSLs of languages you already master

Considerations for System Developers

- Which features do I want to offer?
  - e.g. automatic data decomposition & distribution often requires data flow analysis and hence a compiler
- What do I require my application developers to master?
  - build upon widely-used programming languages
  - choosing a library-based model often decreases adoption barriers
- How much effort can I spend on this? How many people are in my workforce?
  - languages and their toolchains take an immense (!) amount of effort if done properly
  - only move away from library-only solutions if necessary
  - even then, consider language extensions first

Questions?

Image Sources

- Unstructured Mesh: https://resourcearea.cpu-24-7.com/en/numeca_welcome
- Yoda: https://www.deviantart.com/biggiepoppa/art/Master-Yoda-Star-Wars-395511111
- State-of-the-Art Overview: https://link.springer.com/content/pdf/10.1007%2Fs11227-018-2238-4.pdf