Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. **Parallelization and optimization strategies**
   - Types of parallelism
     - In hardware: processor, node & system level parallelism
     - In software: data & task parallelism
     - In algorithms: dwarfs, parallel patterns, design spaces
   - Parallel patterns & design spaces
     - Algorithm structure
       - Divide and conquer
       - Geometric decomposition
       - Adaptive mesh refinement
     - Supporting structures
       - Load balancing
       - SPMD
       - Master/ worker
       - Loop parallelism
       - Fork/ join
   - Case study: Molecular dynamics
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Concurrency vs. Parallelism

- **Parallel computation is about concurrency**
- **Concurrency**
  - → Property of a program
  - → Goal: make program more usable
  - → Execution order of 2 tasks is not predetermined (*could* occur at the same time)
    - → Can start, run and complete in overlapping time periods, not necessarily at the same instant
  - → Tasks are scheduled by the OS
    - → Can also run on one single-core processor
    - → e.g. multitasking
- **Parallelism**
  - → Property of the machine
  - → Goal: make program faster
  - → Performing computations in parallel (simultaneously)
  - → Parallel hardware units are needed (e.g. multicore processor)

Thread vs. Process

- **Process**
  - → Two programs are executed as separate processes (not necessarily vice versa)
    - →They have memory protection = separate address spaces
    - →Processes have their own program counter, stack, heap
  - → Communication between processes possible e.g. sockets, message passing
- **Threads**
  - → Lighter weight than a process (less flexibility, but faster initialization because fewer resources needed)
    - →Threads exist within a process (at least one)
  - → Threads have no separate address space
    - →If several threads within program: all share single memory space & heap

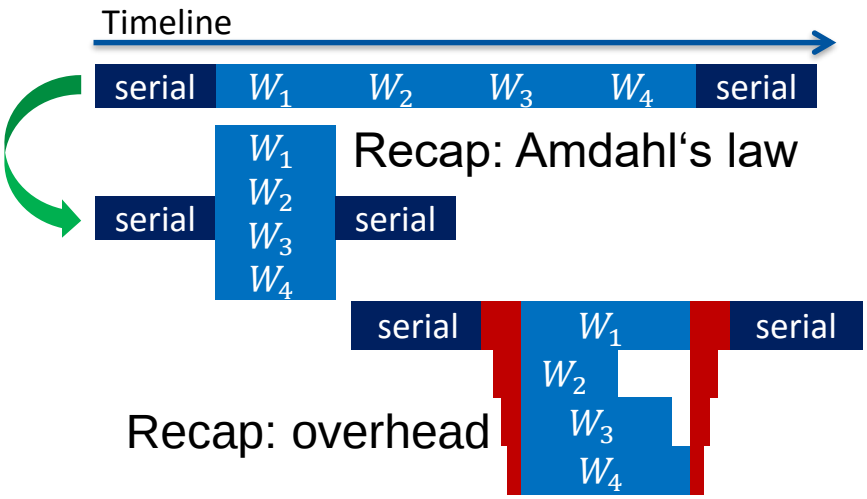
Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. **Parallelization and optimization strategies**
   - **Types of parallelism**
     - **In hardware: processor, node & system level parallelism**
     - In software: data & task parallelism
     - In algorithms: dwarfs, parallel patterns, design spaces
   - Parallel patterns & design spaces
     - Algorithm structure
       - Divide and conquer
       - Geometric decomposition
       - Adaptive mesh refinement
     - Supporting structures
       - Load balancing
       - SPMD
       - Master/ worker
       - Loop parallelism
       - Fork/ join
   - Case study: Molecular dynamics
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Levels of parallelism in hardware

- **A modern supercomputer may contain multiple levels of parallelism**
  - → Processor level parallelism: Superscalar, SIMD
  - → Node/Chip level: Several cores/processors run in parallel with access to the same memory
  - → System level: Several nodes run in parallel and are communicating over a network interconnect

- **Parallelism introduces overhead**
  - → Additional computational costs (cycles)
  - → Implementation (hours of work)
- **Overhead increases from processor to system level**



Processor level parallelism

- **Parallelism at processor/ instruction level**
  - → Pipelining (overlap in execution: load, decode, execute)
  - → Superscalar (redundant arithmetical units: Multiplication, Addition, …)
  - → SIMD execution (e.g. 256 bit registers)

- **Example: Multiplication of two vectors**

```
for(i=0; i<N; ++i)
{
  C[i] = A[i] * B[i];
}
```

- **Programming techniques**
  - → Exploitation of data or instruction level parallelism
  - → Code modifications: Unrolling, Cache reuse
  - → Compiler optimizations

Processor level parallelism – Example

- **Example: 4-way unrolled loop (256-bit AVX SIMD DP execution)**

```
int end=(N/4)*4;
for(i=0; i<end; i+=4)
{
  C[i] = A[i] * B[i];
  C[i+1] = A[i+1] * B[i+1];
  C[i+2] = A[i+2] * B[i+2];
  C[i+3] = A[i+3] * B[i+3];
}
```

Remainder loop

```
for(; i<N; ++i)
{
  C[i] = A[i] * B[i];
}
```

- **Overhead:**
  - → Remainder loop
  - → No costs if N is multiple of 4, otherwise O(10) cycles
- **SIMD vectorization already pays off at short loop length**

Node level parallelism

- **Parallelization approach: threading**
  - → Thread administration/ synchronization overhead: O(100) – O(1000) cycles
  - → Workload should exceed O(100) – O(1000) cycles in computation time
- **Example: Matrix-Vector Multiplication (MVM)**
  - → Costs:
    - →inner loop: O(200)-O(1000)
    - →Parallelize outer loop!
- **Programming techniques**
  - → Auto-parallelization (by compiler)
    - →Works only in most simple cases
  - → Compiler directives (OpenMP)
  - → Explicit parallelization (pthreads, Win32 Threads)

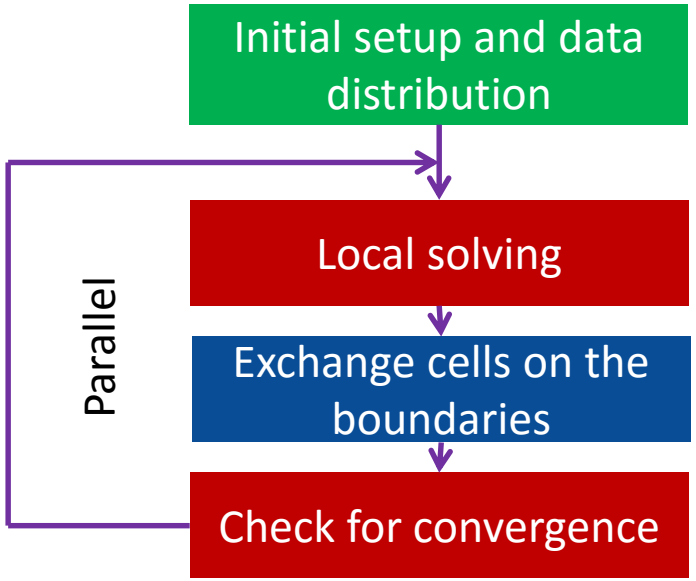
```
for(i=0; i<N; ++i)
{
  C[i] = 0;
  for(j=0; j<N; ++j)
  {
    C[i] += A[i][j] * B[j];
  }
}
```

System level parallelism

- **System level parallelism**
  - → Overhead of parallelization: typically O(10,000+) cycles
  - → Workload needs to exceed O(10,000+) in computation time

- **Example: Domain decomposition in CFD: Mapping of 3D mesh to the processors**

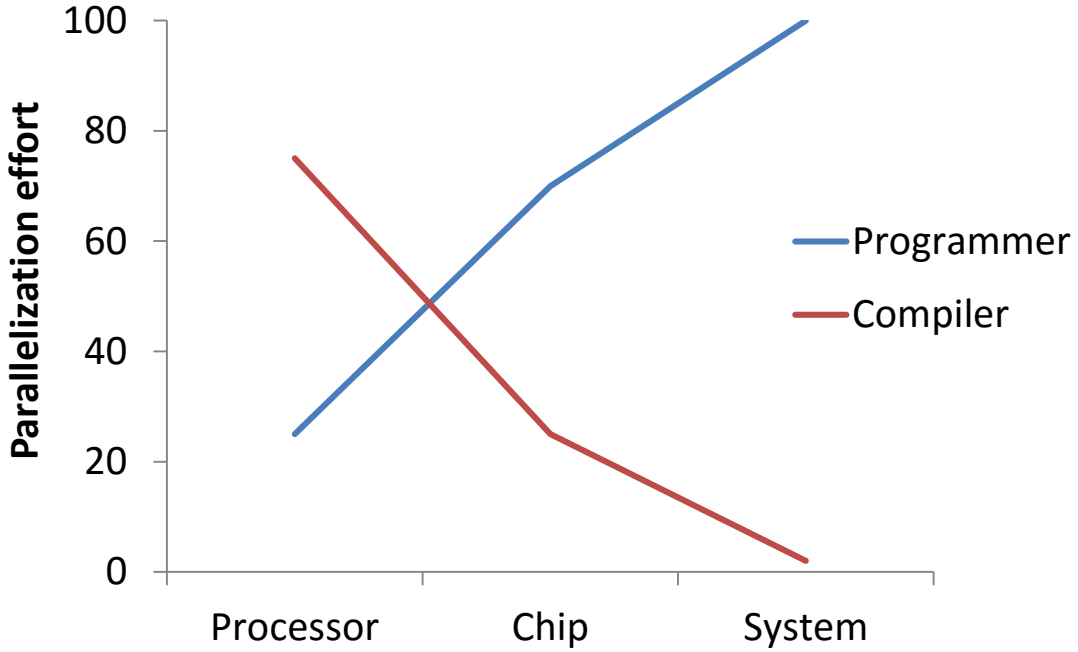
- **Programming techniques**
  - → Data parallel approach
    - → Distribute data structures
  - → Parallel algorithms
  - → Explicit data exchange (MPI)



Expenses on parallelization

- **Task of parallelizing is done by**
  - → Compiler
    - →On the Processor and Node/Chip level (fine and intermediate level)
  - → Programmer
    - →On the Processor, Node/Chip and System level (fine, intermediate and coarse level)

Goal: Exploit all available levels of parallelism!



Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. **Parallelization and optimization strategies**
   - **Types of parallelism**
     - In hardware: processor, node & system level parallelism
     - **In software: data & task parallelism**
     - In algorithms: dwarfs, parallel patterns, design spaces
   - Parallel patterns & design spaces
     - Algorithm structure
       - Divide and conquer
       - Geometric decomposition
       - Adaptive mesh refinement
     - Supporting structures
       - Load balancing
       - SPMD
       - Master/ worker
       - Loop parallelism
       - Fork/ join
   - Case study: Molecular dynamics
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Types of parallelism in software

- **Programmer can choose to parallelize his code by**
  - → Task parallelism
  - → Data parallelism

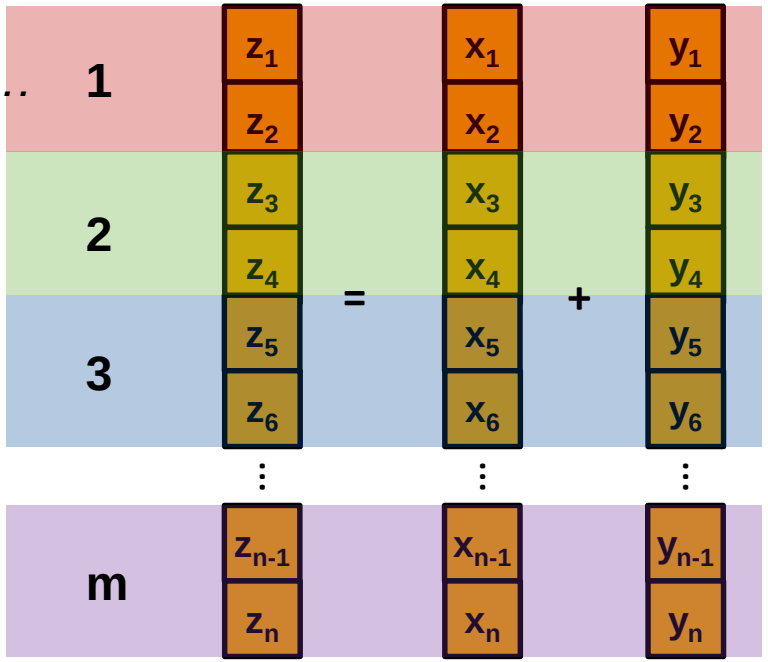
→ **Mapping of software parallelization to parallel hardware units**

Data parallelism

- **Use case**
  - → Items are processed in a collection of data (often in a loop)
- **Idea**
  - → "Distribute" data across different processing elements/ compute units
    - →Often: domain decomposition
  - → Process same activity on different items/ subsets of (distributed) data at the same time
- **Example**

```
for(i=0; i<n; i++)
{
  z[i] = x[i] + y[i];
}
```

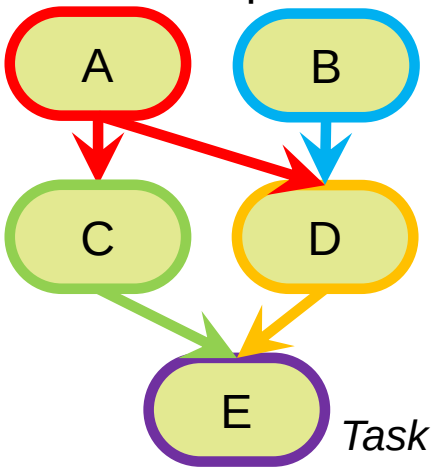
  - → Thread/ process 1-m can work in parallel on different data



Task parallelism

- **Use case**
  - → Work is divided into several activities (which you cannot parallelize individually)
- **Idea**
  - → Task = indivisible sequential unit of computation
  - → Create multiple distinct task bodies performing different activities at the same time (e.g. in recursive algorithms/ divide-and-conquer approaches)
  - → Dependencies might occur between different tasks
  - → Data parallelism can often be expressed as task parallelism (depending on definition)
- **Example**

```
A=a();
B=b();
C=c(A);
D=d(A,B);
E=e(C,D);
```



Task dependency graph

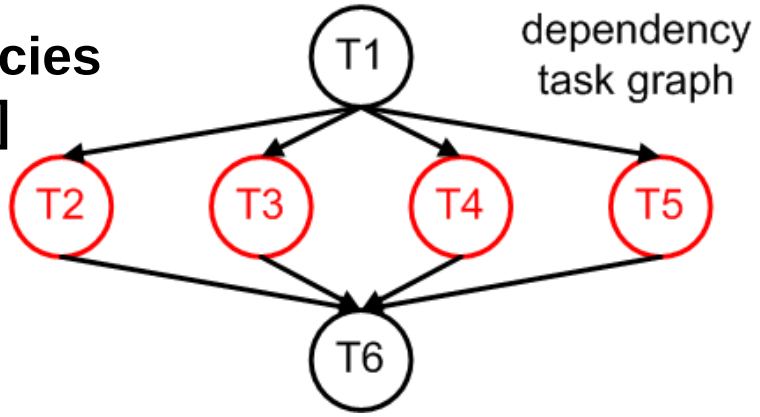
→ Potential parallelism

→ Task A & B

→ Task C & D

Dependency Graphs

- **= directed graph representing dependencies of several objects towards each other [1]**
  - → Needed: DAG = directed acyclic graph
  - → Node = task
  - → Edge = (control or data) dependency

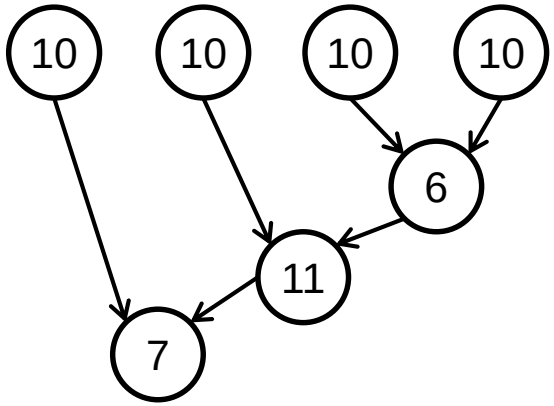


- **Metrics**
  - → Directed path: sequence of tasks must be processed one after the other
  - → Critical path: longest such path (from start to end node)
    - → Shortest time in which program can be executed in parallel
  - → Critical path length: length of longest path (sum of weights)
  - → Degree of concurrency: #tasks that can execute in parallel
    - → Maximum degree of concurrency: largest # of concurrent tasks at any point of execution
    - → Average degree of concurrency: ratio of total amount of work to critical path length (average # of tasks to execute in parallel)

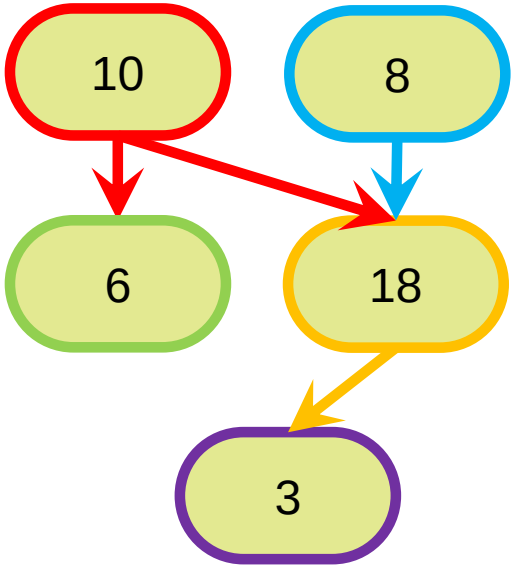
[1] Wikipedia, http://en.wikipedia.org/wiki/Dependency_graph

Dependency Graphs – Examples

Total work: 64
Critical path length: 34
Avg. concurrency: ~1.9



Total work: 45
Critical path length: 31
Avg. concurrency: ~1.5

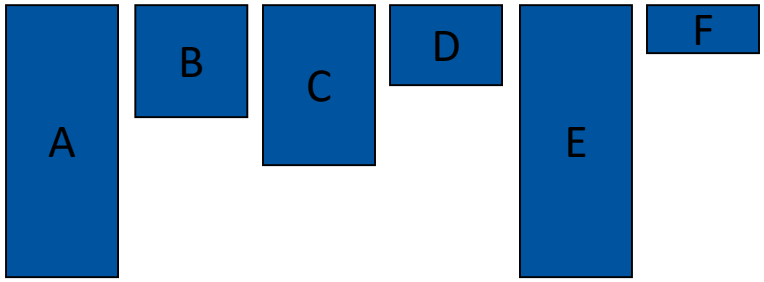


Granularity of task decomposition

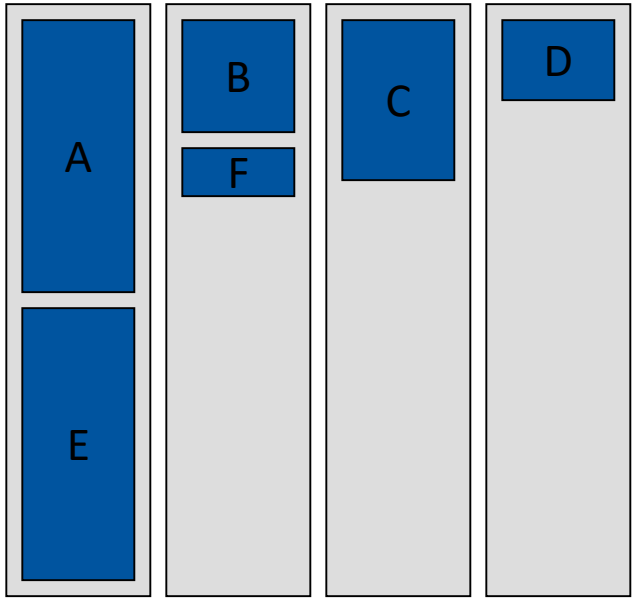
- **Tasks can be of different size (=granularity)**
  - → Fine-grained decomposition
    - →Large number of small tasks
  - → Coarse-grained decomposition
    - →Small number of large tasks
  - → Both affect the critical path length and the degree of concurrency
- **Fine-grained vs. coarse-grained decomposition**
  - → Fine-grained: better chance for load balance (see later)
    - →Smaller tasks have more flexibility to be distributed evenly
  - → Coarse-grained: less management overhead
    - →Reduces overhead of (too) many small tasks

Task scheduling

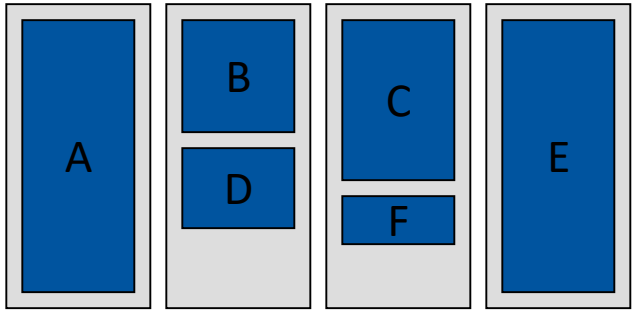
Independent tasks



Poor mapping to 4 execution units



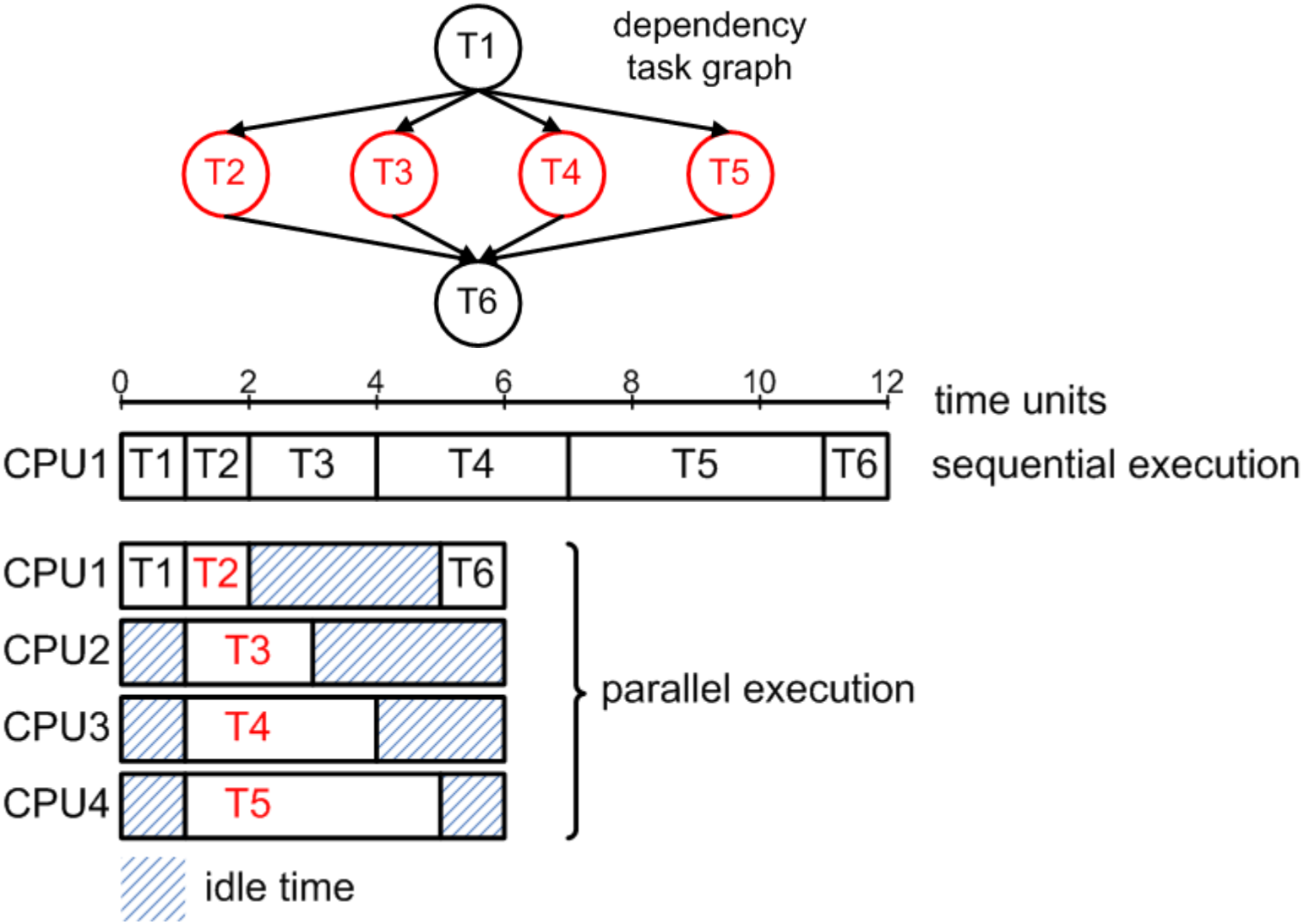
Good mapping to 4 execution units



Also called “load balance” → see later chapter

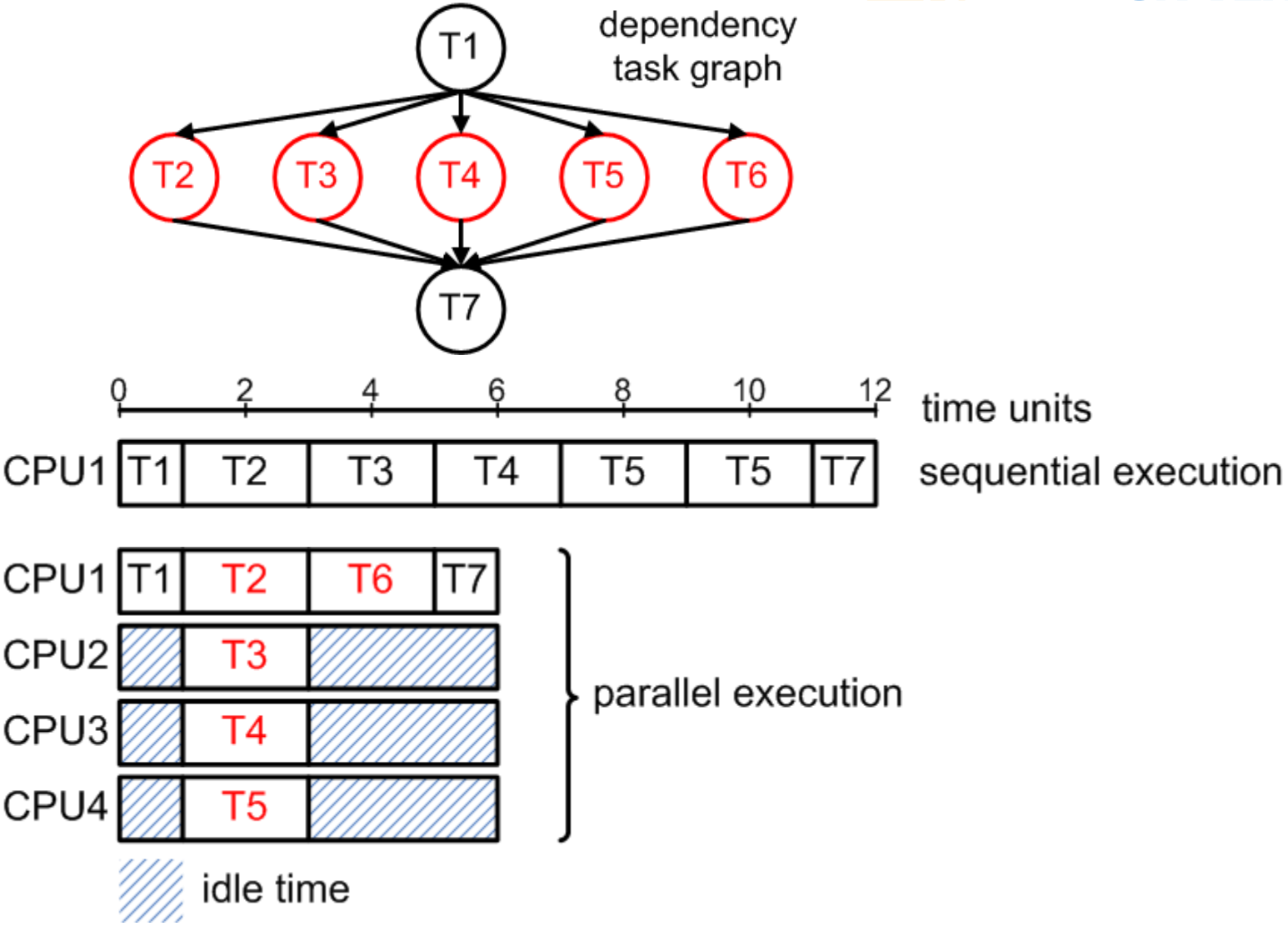
Courtesy of Edgar Gabriel

Task scheduling with dependencies – Examples



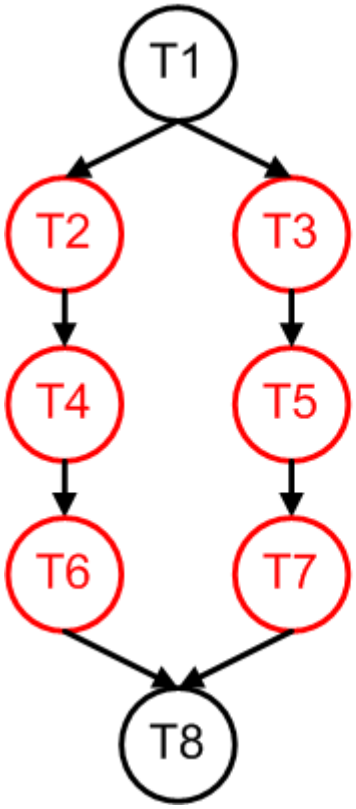
Courtesy of Wolfgang Nagel

Task scheduling with dependencies – Examples

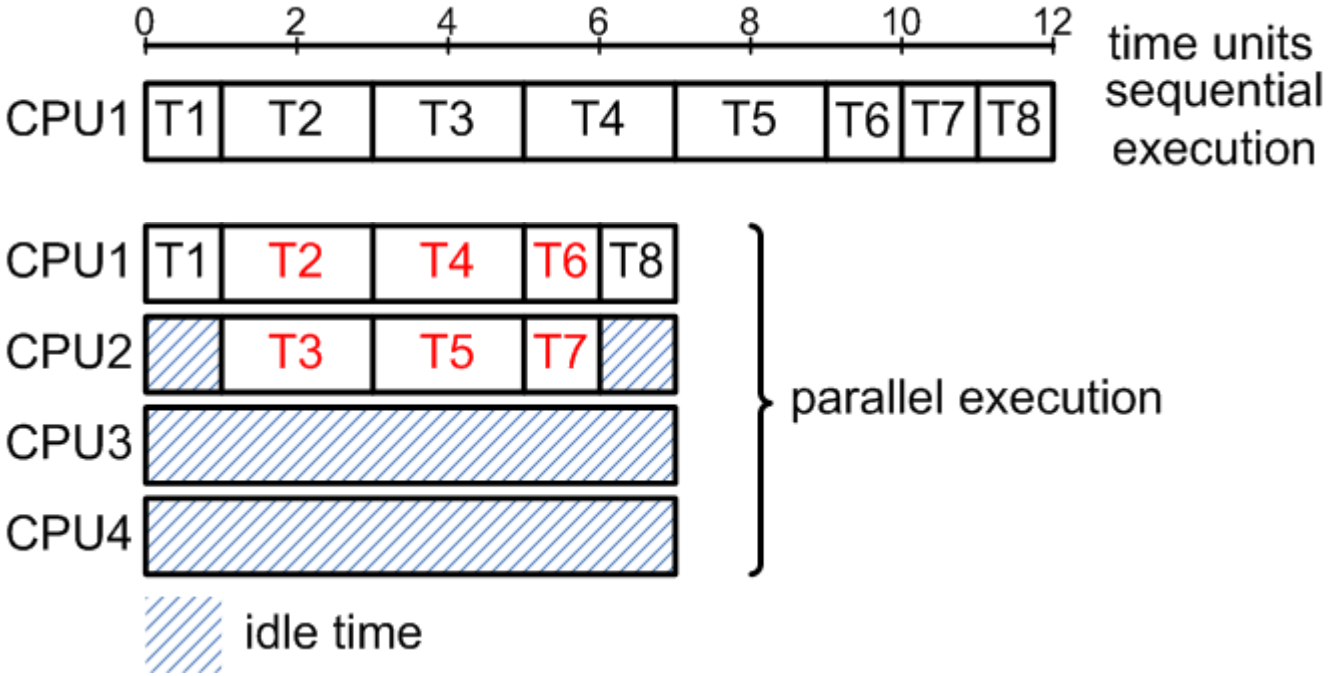


Courtesy of Wolfgang Nagel

Task scheduling with dependencies – Examples



dependency task graph



Courtesy of Wolfgang Nagel

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. **Parallelization and optimization strategies**
   - **Types of parallelism**
     - In hardware: processor, node & system level parallelism
     - In software: data & task parallelism
     - **In algorithms: dwarfs, parallel patterns, design spaces**
   - Parallel patterns & design spaces
     - Algorithm structure
       - Divide and conquer
       - Geometric decomposition
       - Adaptive mesh refinement
     - Supporting structures
       - Load balancing
       - SPMD
       - Master/ worker
       - Loop parallelism
       - Fork/ join
   - Case study: Molecular dynamics
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

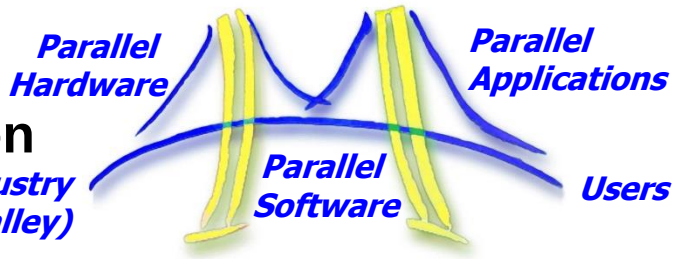
Parallel patterns

- **Parallel patterns**
  - → “Best practices” for solving specific problems
  - → Support a particular “algorithmic structure” (maybe with efficient implementation)
  - → May future architectures/ programming models support these patterns?
  - → Several different approaches are in practice

- **Approaches**
  - → Dwarfs [1]
  - → Structured parallel patterns [2]/ algorithmic skeletons
  - → Parallel design patterns [3]

[1] Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The Landscape of Parallel Computing Research: A View from Berkeley. Tech. Rep. UCB/EECS-2006-183 (2006)
[2]McCool, M., Reinders, J., Robison, A.: Structured Parallel Programming: Patterns for Efficient Computation. Morgan Kaufmann, first edn. (2012)
[3] Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming. Addison-Wesley Professional, first edn. (2004)

Berkeley's Dwarfs



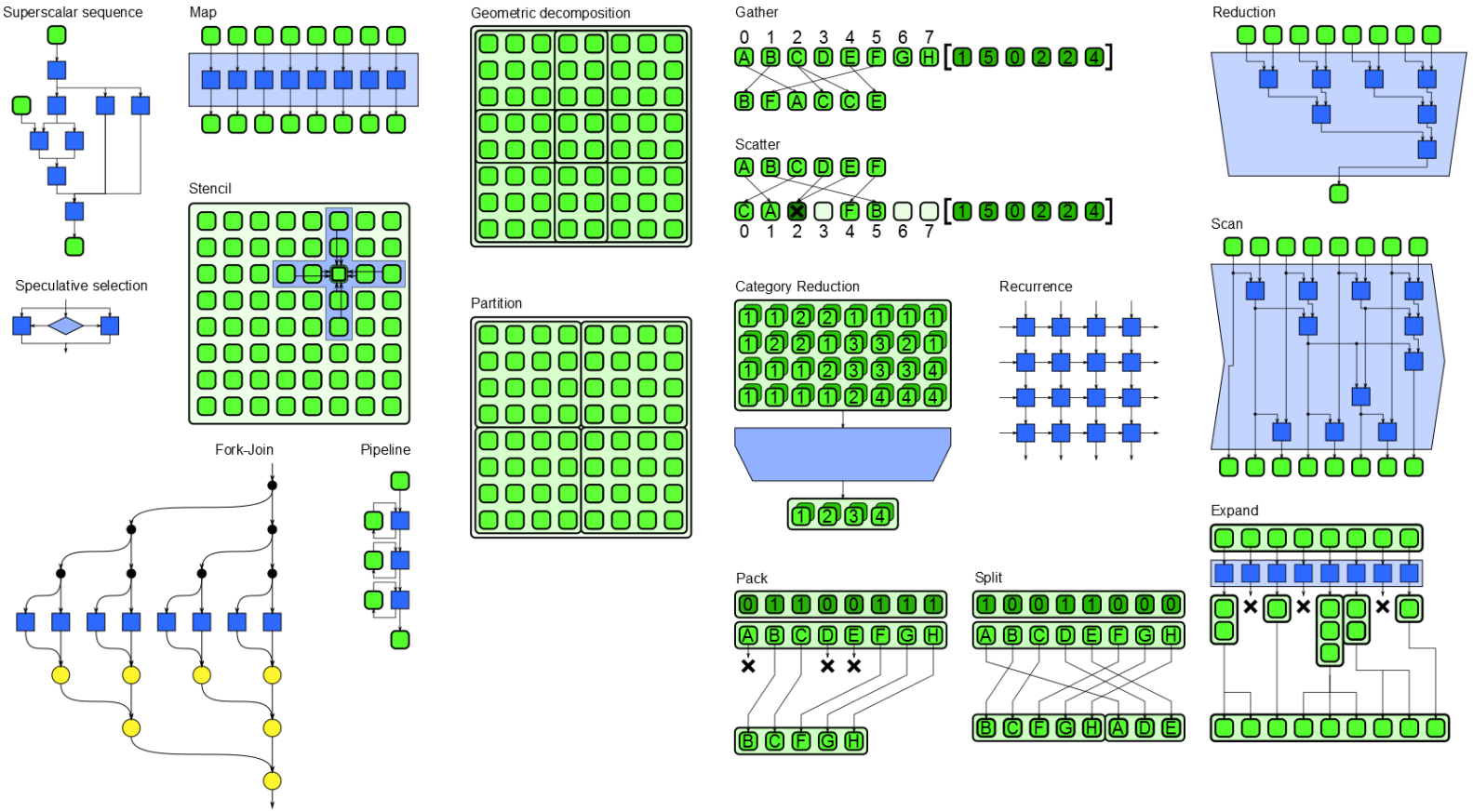
- **Dwarf/ motif: algorithmic method that captures pattern of computation/ communication**
- **Goal: basis for design/ evaluation of parallel programming models & architectures**

- **Definition of 7 dwarfs of HPC (Phil Colella)**
  - → Dense linear algebra: e.g. dense matrix-matrix multiplication (often with BLAS)
  - → Sparse linear algebra :e.g. sparse matrix-vector multiplication
  - → Spectral methods (FFT): solving differential equations & Eigenvalue problems
  - → N-body methods: particle-oriented simulation (interaction between points)
  - → Structured grid methods: discretization of problem space into regular parts
  - → Unstructured grid methods: irregular data structures
  - → Monte Carlo methods: include statistical results (with repetitions)
- **Extension to 13 dwarfs (Lawrence Berkeley National Lab)**
  - → Combinational logic, graph traversal, dynamic programming, backtrack/ branch & bound, graphical models, finite state machines

Source: Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The Landscape of Parallel Computing Research: A View from Berkeley. Tech. Rep. UCB/EECS-2006-183 (2006)

Structured parallel patterns

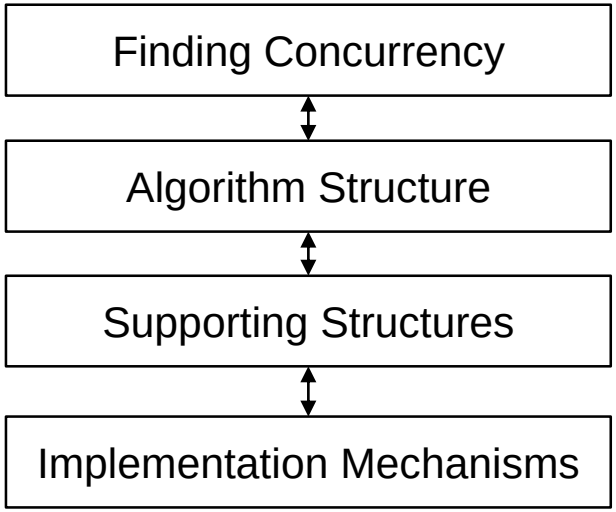
- **Pattern by McCool et al (Intel): “algorithmic structure” with an efficient implementation**
  - → Also called “algorithmic skeletons” [2]
  - → Good parallel prog. model should support useful pattern + efficient implementation → maintainable software



[1] McCool, M., Robison, A., Reinders, J.: Structured Parallel Programming: Patterns for Efficient Computation. Morgan Kaufmann, first edn. (2012)
[2] Murray Cole: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, Cambridge, MA, USA (1991)
[3] Hebenstreit, Reinders, Robison, McCool: Structured parallel Programming with Patterns, SC13 Tutorial

Design spaces & pattern

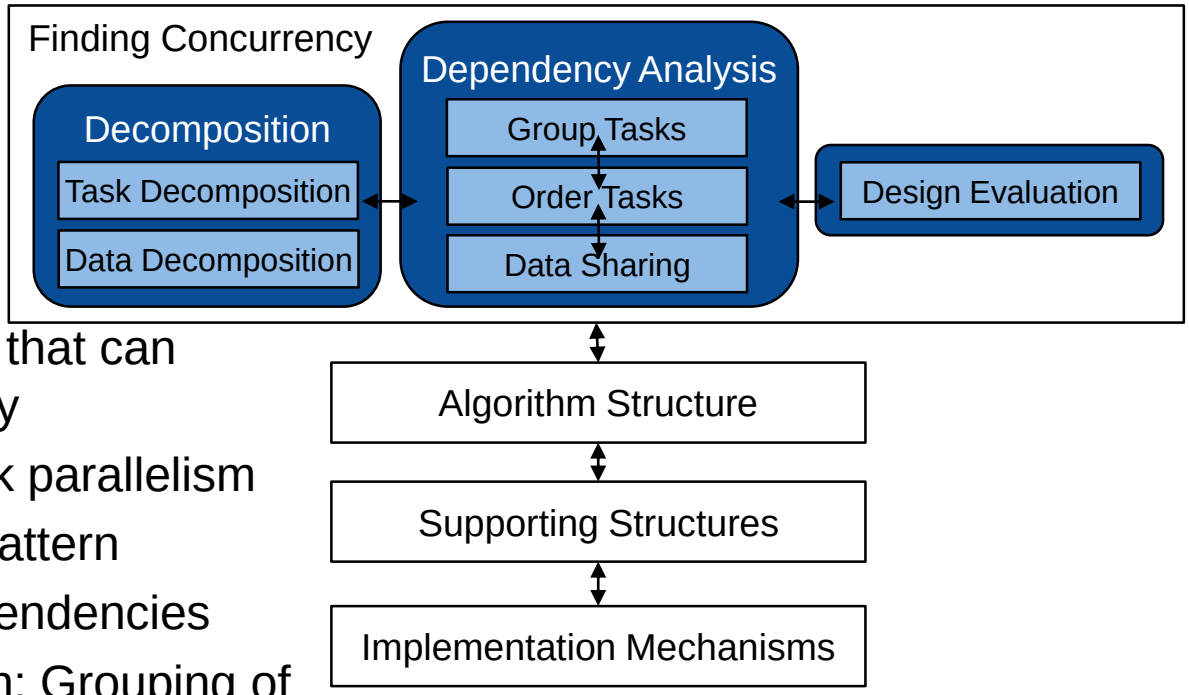
- **Mattson et al. define a pattern language (design spaces and design patterns)**
- **Design spaces**
  - → Finding concurrency
    - →Programmer works in problem domain to identify available concurrency
  - → Algorithm structure
    - →Programmer works with high-level structures for organizing a parallel algorithm
  - → Supporting structure
    - →Shift from algorithms to source code: organization of parallel program
  - → Implementation mechanisms
    - →Programmer looks at specific software constructs for implementing parallel program



Source: Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming. Addison-Wesley Professional, first edn. (2004)

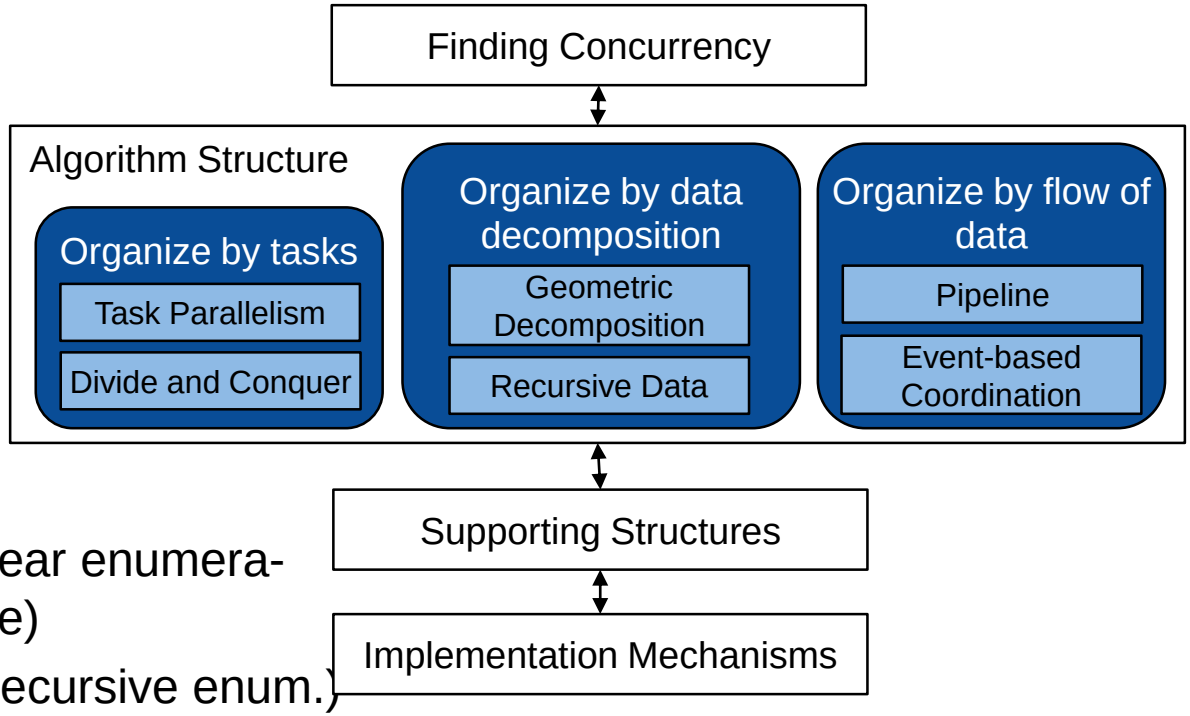
Design space – Finding concurrency

- **Problem domain**
- **Design patterns**
  - → Decomposition patterns
    - → Composition of problem into pieces that can execute concurrently
    - → Also see data & task parallelism
  - → Dependency Analysis Pattern
    - → Identification of dependencies
    - → Group Tasks Pattern: Grouping of tasks to simplify the managing dependencies (e.g. because a group shares a (temporal) constraint such as file reading or shared data access)
    - → Order Tasks Pattern: Ordering of (group) tasks according to dependencies
    - → Data Sharing Pattern: Mapping of shared data among tasks
  - → Design Evaluation Patterns
    - → Guidance what have been done so far → good enough to move on?



Source: Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming. Addison-Wesley Professional, first edn. (2004)

Design space – Algorithm Structure

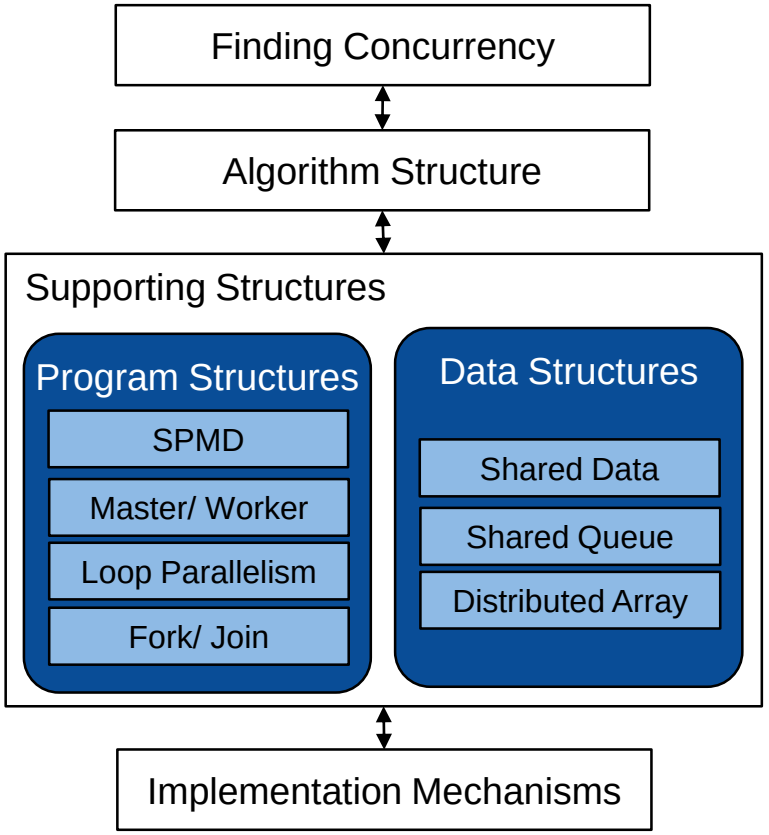


- **Which pattern is the best for the problem (algorithm, machine, prog. environment)?**
- **Design patterns**
  - → As decision tree
  - → Organize by tasks
    - → Task parallelism (linear enumeration of tasks possible)
    - → Divide & Conquer (recursive enum.)
  - → Organize by data decomposition
    - → Geometric decomposition (linear decomposition) into discrete subspaces
    - → Recursive data (recursive decomposition), e.g. for searching in binary tree
  - → Organize by flow of data: imposes an ordering on the groups of tasks
    - → Pipeline: flow among task groups is regular/ one-way/ static
    - → Event-based coordination: flow is irregular/ dynamic/ unpredictable

Source: Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming. Addison-Wesley Professional, first edn. (2004)

Design space – Supporting Structures (1/2)

- **Between problem domain and specific implementation**
- **Design patterns**
  - → Program Structures
    - → SPMD (Single Program, Multiple Data): all UEs execute same program in parallel on own data
    - → Master/ Worker: Master manages worker pool with bag of tasks
    - → Loop parallelism: iterations of loop are executed in parallel
    - → Fork/ Join: one UE forks off some other UEs & joins them at the end

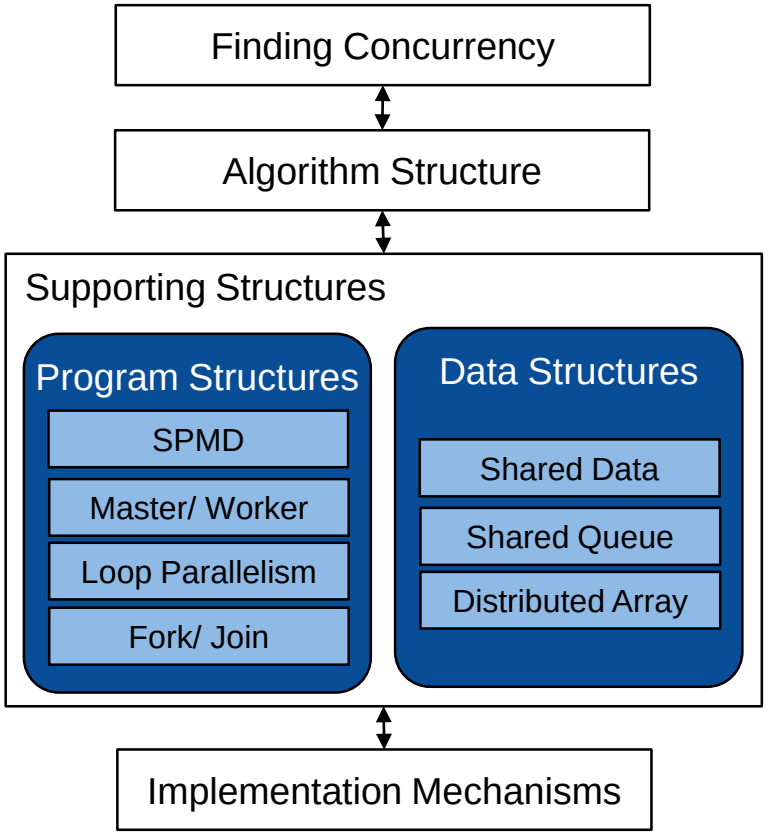


UE: unit of execution, e.g. process/ thread

Source: Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming. Addison-Wesley Professional, first edn. (2004)

Design space – Supporting Structures (2/2)

- **Design patterns**
  - → Data Structures
    - → Shared Data: Handles problem of using shared data
    - → Shared Queue: “thread-safe” implementation of the queue abstract data type (ADT)
    - → Distributed Array: Arrays of 1 or more dimensions that are decomposed into sub-arrays & distributed among processes/ threads



UE: unit of execution, e.g. process/ thread

Source: Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming. Addison-Wesley Professional, first edn. (2004)

Design space – Supporting Structures

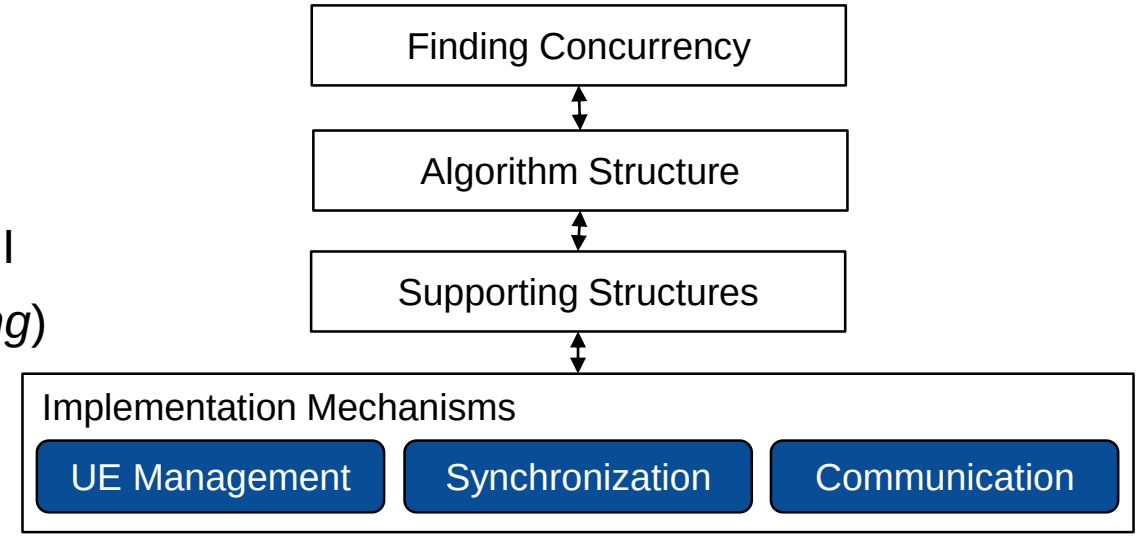
- **Relationship between Supporting Structures & Algorithm Structure patterns**
  - → Stars: indication of likelihood that the given supporting structures pattern is useful in the implementation of the algorithm structure pattern

	Task parallelism	Divide & conquer	Geometric decomposition	Recursive data	Pipeline	Event-based coordination
SPMD	★★★★	★★★	★★★★	★★	★★★	★★
Loop parallelism	★★★★	★★	★★★			
Master/ Worker	★★★★	★★	★	★	★	★
Fork/ Join	★★	★★★★	★★		★★★★	★★★★

- **Supporting structures patterns are flexible in their application in algorithm structure patterns**
  - → E.g. SPMD can be used to implement all defined algorithm structures
  - → The programming environment can narrow the choice

Source: Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming. Addison-Wesley Professional, first edn. (2004)

Design space – Implementation Mechanisms



- **Source code domain**
  - → Write parallel program
  - → E.g. with OpenMP or MPI (see *parallel programming*)
- **Design patterns**
  - → UE management
    - →Creation, destruction, management of the processes/ threads used in parallel computation
  - → Synchronization
    - →Enforcing constraints on the ordering of events occurring in different Ues
    - →Usually to ensure correct result when accessing shared data
  - → Communication
    - →Exchange of information between UEs

UE: unit of execution, e.g. process/ thread

Source: Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming. Addison-Wesley Professional, first edn. (2004)

Design spaces in the following

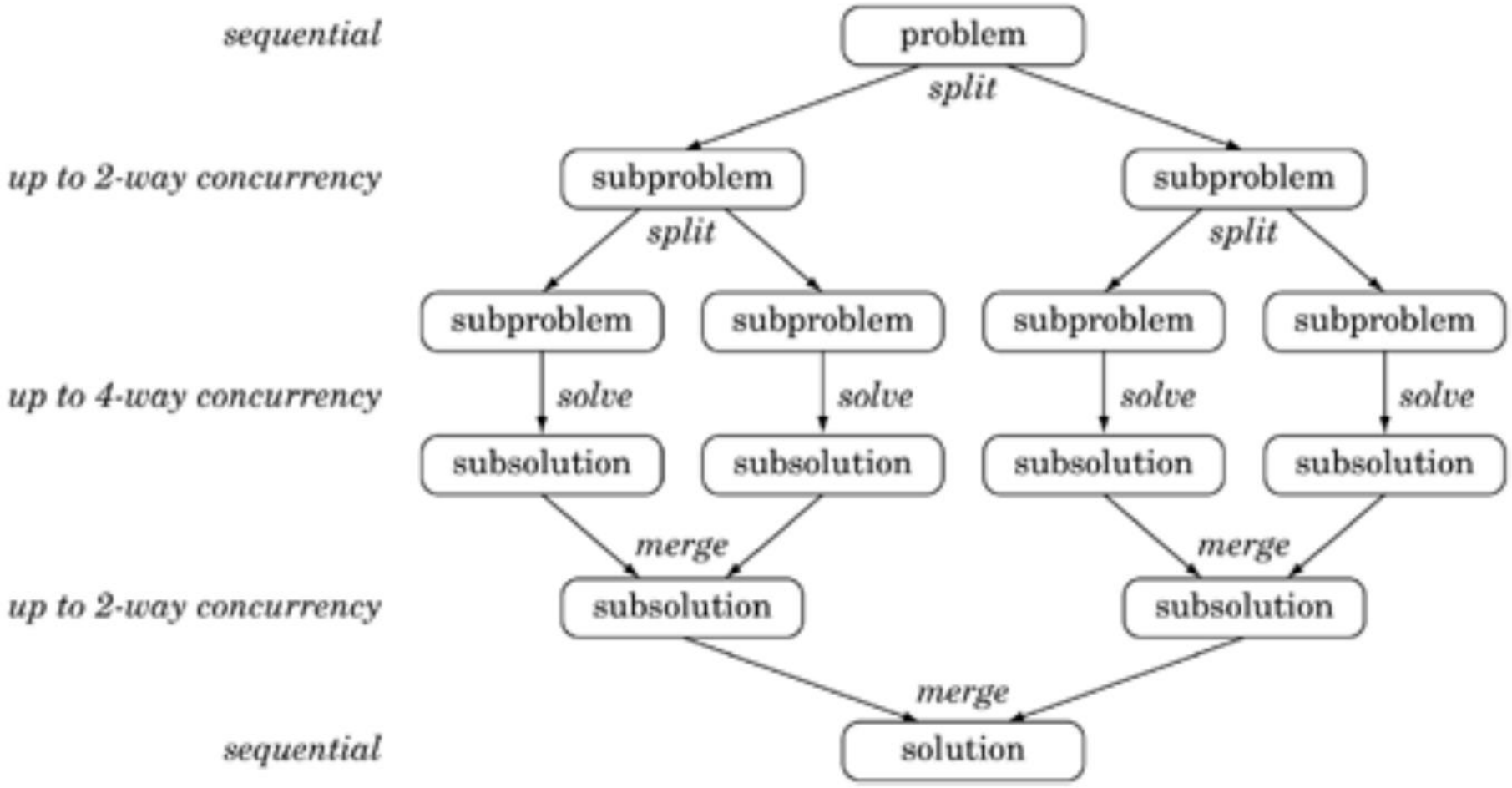
- **As seen: different approaches for structuring parallel problems**

- **Here, we follow the design spaces by Mattson et al**
  - → Focus on the most common patterns
  - → Additionally, explanation of some basics on load imbalances and some enhanced patterns (e.g. adaptive mesh refinement)

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. **Parallelization and optimization strategies**
   - Types of parallelism
     - In hardware: processor, node & system level parallelism
     - In software: data & task parallelism
     - In algorithms: dwarfs, parallel patterns, design spaces
   - **Parallel patterns & design spaces**
     - **Algorithm structure**
       - **Divide and conquer**
       - Geometric decomposition
       - Adaptive mesh refinement
     - Supporting structures
       - Load balancing
       - SPMD
       - Master/ worker
       - Loop parallelism
       - Fork/ join
   - Case study: Molecular dynamics
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Divide and Conquer



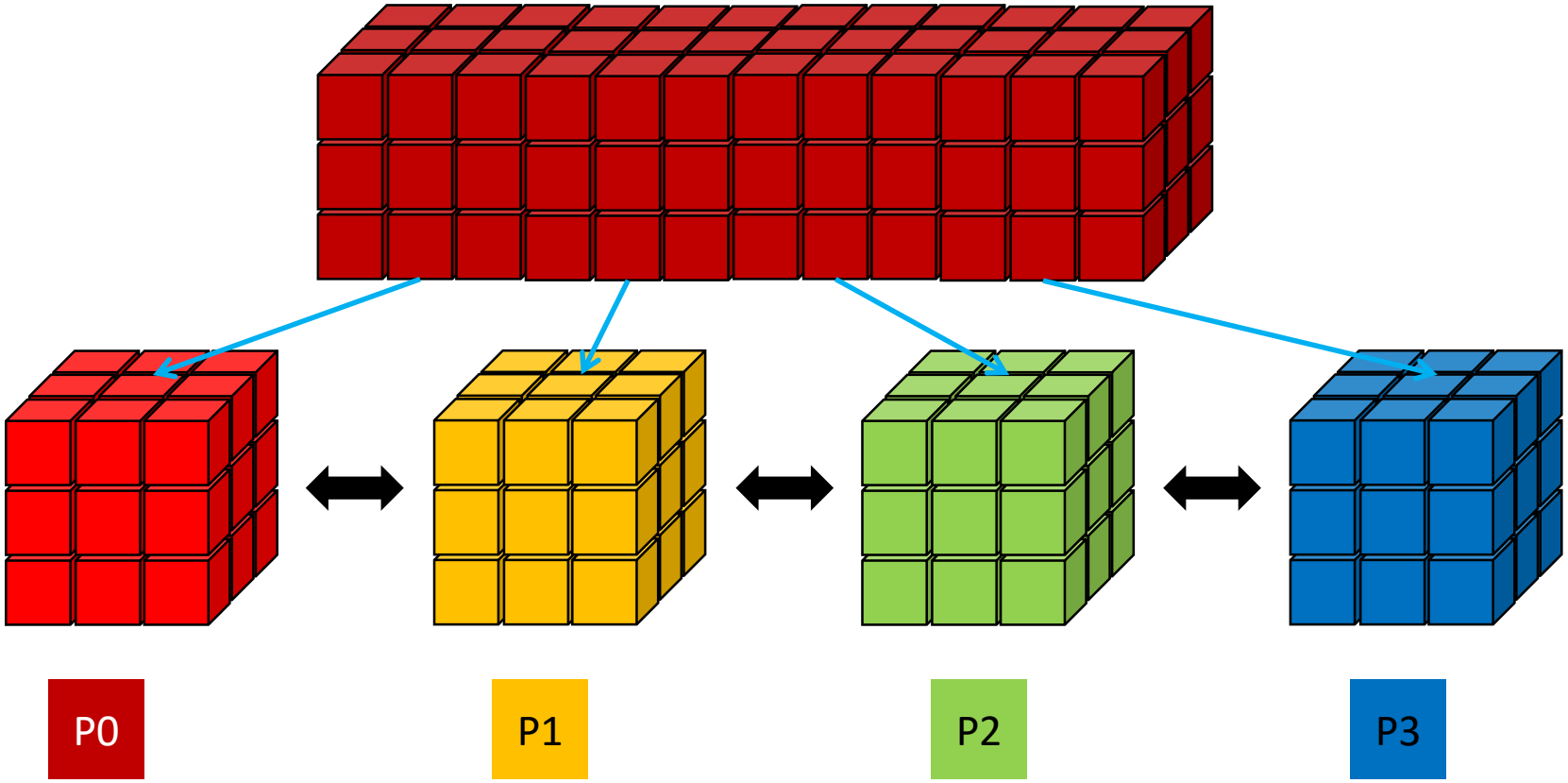
Source: Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming. Addison-Wesley Professional, first edn. (2004)

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. **Parallelization and optimization strategies**
   - Types of parallelism
     - In hardware: processor, node & system level parallelism
     - In software: data & task parallelism
     - In algorithms: dwarfs, parallel patterns, design spaces
   - **Parallel patterns & design spaces**
     - **Algorithm structure**
       - Divide and conquer
       - **Geometric decomposition**
       - Adaptive mesh refinement
     - Supporting structures
       - Load balancing
       - SPMD
       - Master/ worker
       - Loop parallelism
       - Fork/ join
   - Case study: Molecular dynamics
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Geometric decomposition

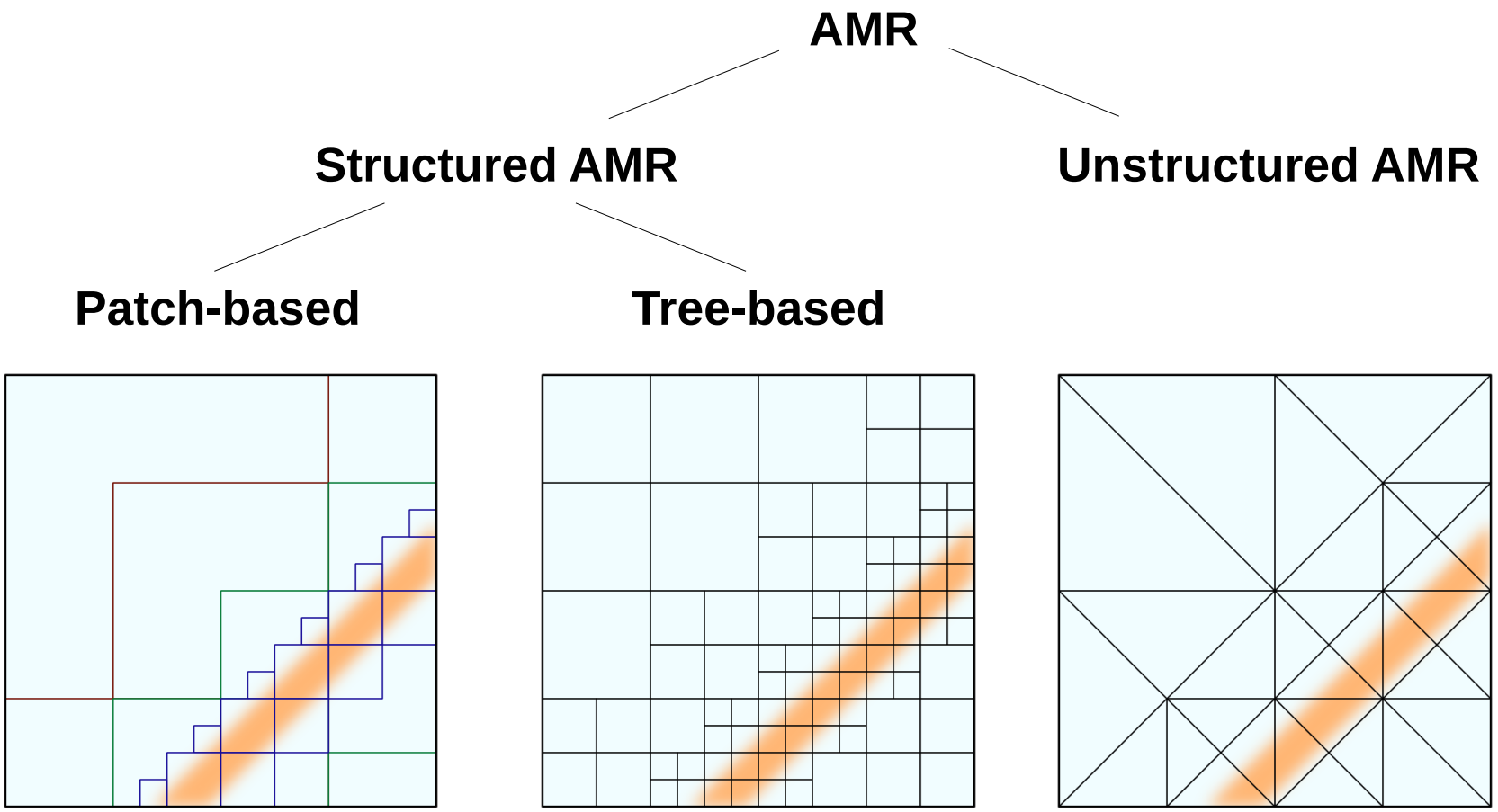
- **Example: Domain decomposition of a 3D-mesh**
  - → Simple communication and load balancing



Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. **Parallelization and optimization strategies**
   - Types of parallelism
     - In hardware: processor, node & system level parallelism
     - In software: data & task parallelism
     - In algorithms: dwarfs, parallel patterns, design spaces
   - **Parallel patterns & design spaces**
     - **Algorithm structure**
       - Divide and conquer
       - Geometric decomposition
       - **Adaptive mesh refinement**
     - Supporting structures
       - Load balancing
       - SPMD
       - Master/ worker
       - Loop parallelism
       - Fork/ join
   - Case study: Molecular dynamics
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

AMR Approaches



Matthias Lieber (TU Dresden)

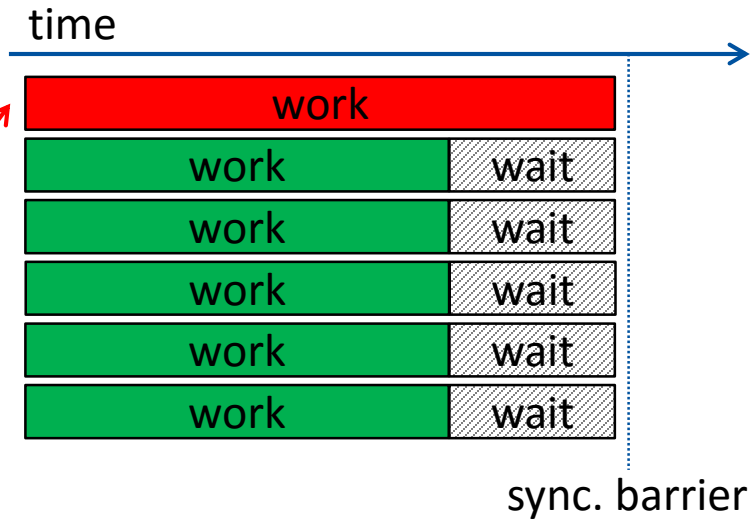
Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. **Parallelization and optimization strategies**
   - Types of parallelism
     - In hardware: processor, node & system level parallelism
     - In software: data & task parallelism
     - In algorithms: dwarfs, parallel patterns, design spaces
   - **Parallel patterns & design spaces**
     - Algorithm structure
       - Divide and conquer
       - Geometric decomposition
       - Adaptive mesh refinement
     - **Supporting structures**
       - **Load balancing**
       - SPMD
       - Master/ worker
       - Loop parallelism
       - Fork/ join
   - Case study: Molecular dynamics
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Load imbalance

- **Common problem: load imbalance**
  - → May be a reason for bad performance and scalability
  - → Occurs when some workers reach synchronization barriers earlier than others

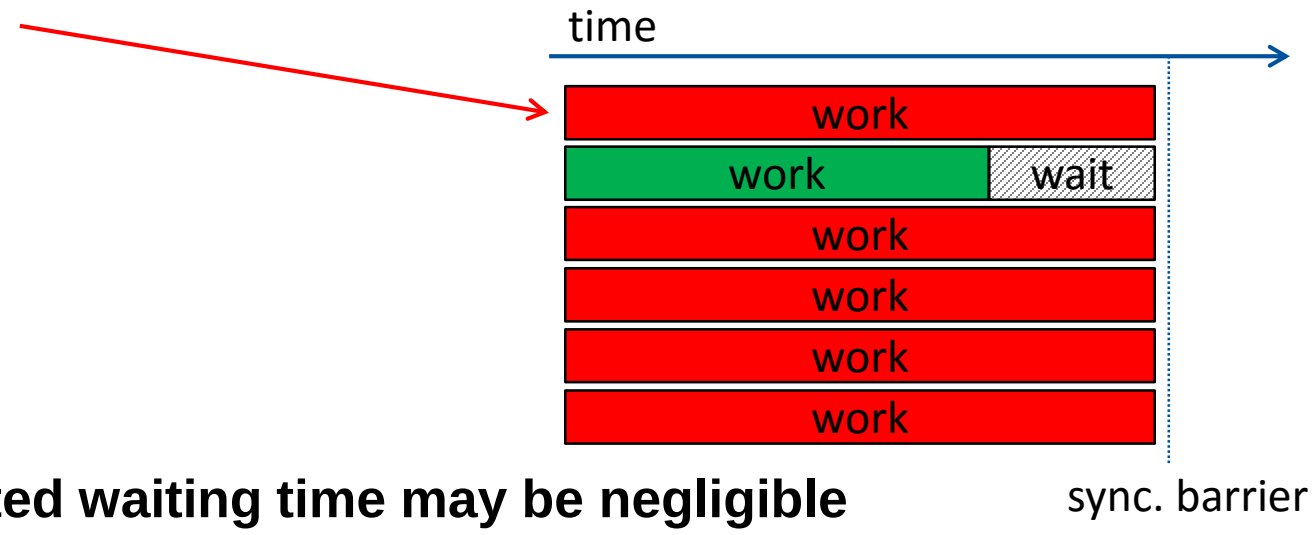
Few „laggers“ waste a lot of resources when concurrently running workers have to wait at a synchronization barrier



- → **Try to tackle with different supporting structures (in the next chapters): “load balancing”**
  - → E.g. dynamic task parallelism or master/ worker pattern

Load imbalance – Example

- **A few “speeders” on the other side may be harmless**



- **Accumulated waiting time may be negligible**

  → Turning “laggers” into “speeders” may induce significant performance boost

- **Reasons for load imbalance are diverse**

  → Optimization problems

  → Algorithmic issues

Challenges

- **Method for distributing work among the workers may not be compatible with the structure of the problem**
  - → Cyclic or dynamic work distribution may improve load balancing
- **Load balance may not be known at compile time**
  - → i.e. how much time a “chunk” of work actually takes
  - → e.g. iterative solvers may require different numbers of iterations to a solution when executed in parallel among multiple workers
- **Parallelism may be limited by coarse granularity of the problem**
  - → Happens usually when the number of workers is not significantly smaller than the number of work packages

Challenges

- **Other reasons may account for load imbalance as well**
  - → e.g. wait times for shared resources (I/O, communication devices)
  - → OS jitter (next slides)

- **If load balancing is identified as a major performance problem, different strategies for work distribution should be considered**
  - → If completely even work distribution is impossible, getting rid of “laggers” may already substantially improve performance and scalability
  - → Overlapping I/O and communication with useful work can avoid load imbalances

Impact of the operating system (OS)

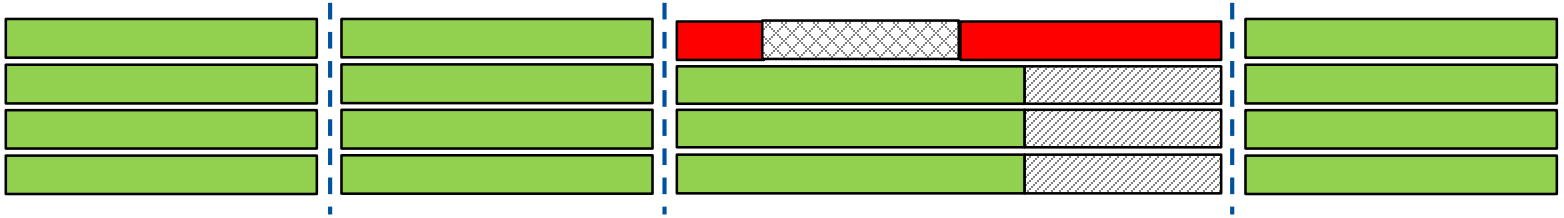
- **OS activity is a peculiar and very unexpected source of load imbalances**

- **OS has many routine tasks**
  - → Running userland code is only one of several others
  - → Writing log files
  - → Executing scheduled jobs (Cron jobs)
  - → Flushing disk caches

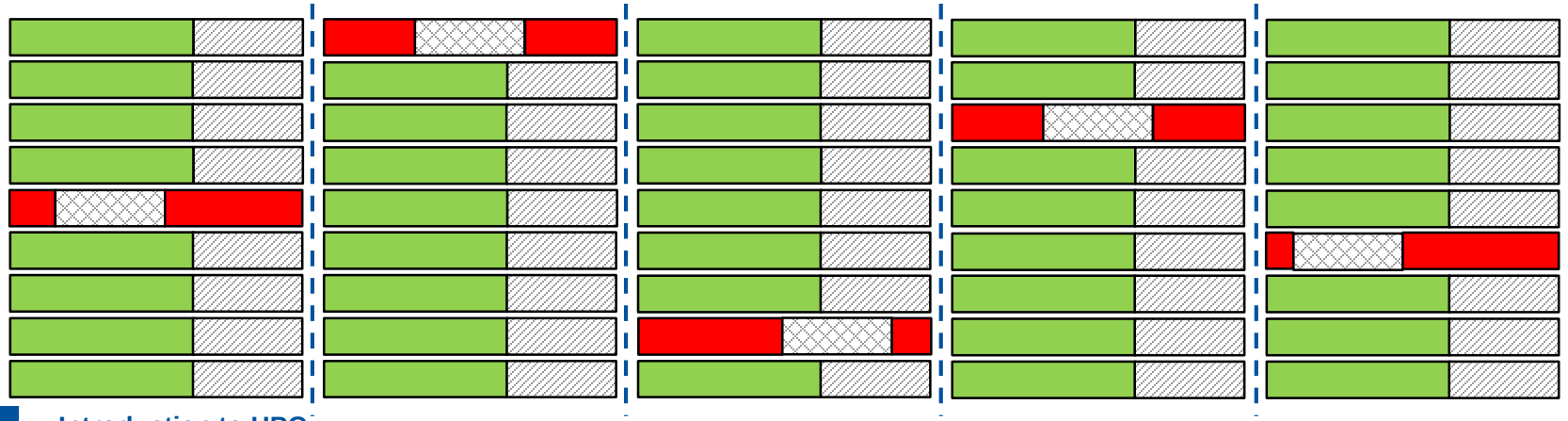
- **Userland code is delayed by some extent via a software interrupt**

OS jitter

- “OS jitter” generates occasional “laggers” at small node counts

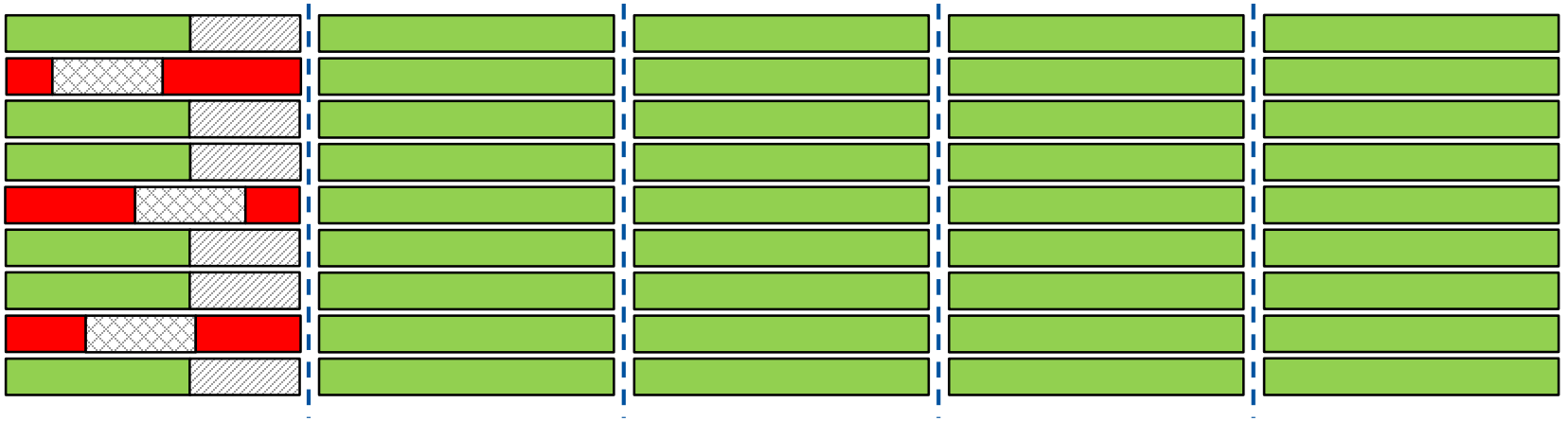


- Impact of “OS jitter” strongly depends on the frequency of synchronization barriers as OS influence is of statistical nature over all workers



OS jitter

- **Goal:**
  - → Reduce OS noise as far as possible
  - → Synchronize OS influences on userland code



- **Approach: Minimized OS**
  - → e.g. CNK (Compute Node Kernel) on Blue Gene architectures

Minimizing OS jitter

- Deactivating/removing unused services
- No logging or polling activities
- Leaving one processor per node free for OS tasks
- Synchronizing OS activities as shown on the previous slide

Optimization/ Algorithmic issues

- **OS jitter is rather complicated to tackle**
  - → May depend on the system architecture

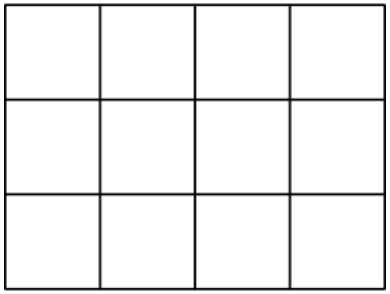
- **But, many load balance approaches exist to tackle optimization/ algorithmic issues**

→ **“Dynamic load balancing”**

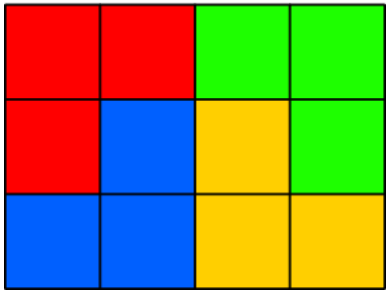
- → Can be implemented by using a special programming construct (e.g. OpenMP tasks) or special algorithmic patterns
- → Often used in combination with
  - → Tasks or
  - → Meshes or grids (see next slides)

Dynamic Load Balancing – Partitioning

- Domain, Mesh, Grid



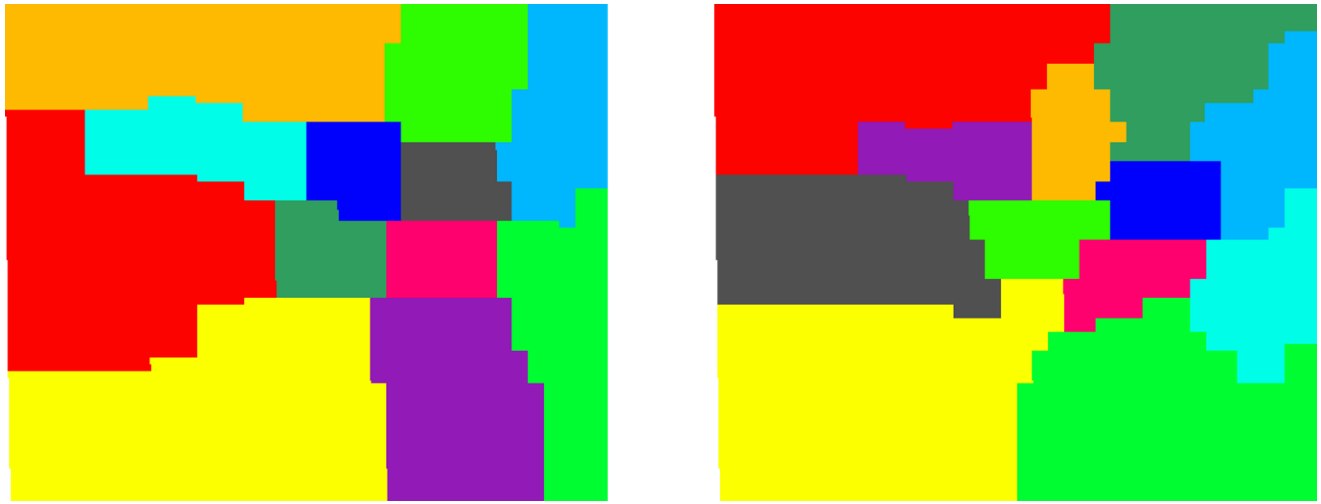
- Decompose into small parts (blocks)



- Assign blocks to processes = Partitioning

Dynamic Load Balancing

- **Static Partitioning**
  - → Partitions do not change during simulation run
  - → Assumes the workload characteristics per partition are constant
  - → Widely used and often sufficient
- **Dynamic Load Balancing**
  - → Partitions are continuously adapted to the workload per grid cell



MUSCAT, Wolke et. al.

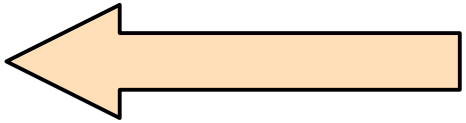
Matthias Lieber (TU Dresden)

Why Dynamic Load Balancing?

- **Adaptive spatial grids**
  - → Number of grid nodes changes locally
- **Computational effort per grid cell varies**
  - → Adaptive time stepping: time step size changes locally
  - → Different equations to be solved at specific events, e.g. ice processes below 0°C

Dynamic Load Balancing Algorithm

predict workload per process
(e.g. from last time step or based on new created adaptive mesh)
if load balance < given tolerance
    predict workload per block *(Weight)*
    compute new partitioning
    redistribute the blocks *(Migration)*
end if
solve equation system



Dynamic Load Balancing Requirements

- **Static Partitioning**
  - → Balance workload on all processes
  - → Minimize communication costs between processes *(Edge-cut)*
    - →i.e. communication due to data exchange between adjacent grid elements in the equation solver

- **Dynamic Load Balancing (additionally)**
  - → Partitioning calculation should execute quickly
  - → Minimize migration costs
    - →i.e. communication due to changes between successive partitions

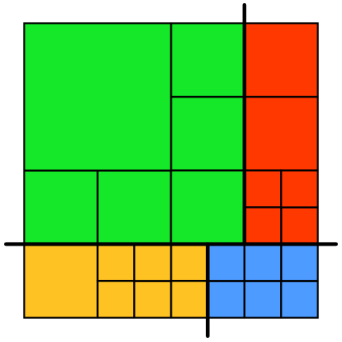
Common Load Balancing Methods

Load Balancing Methods

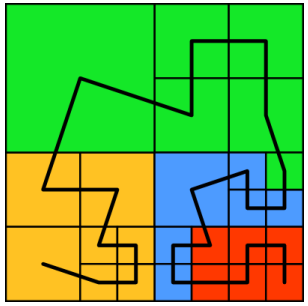
Geometric Methods

- Need spatial coordinates and block weights

Recursive Bisection



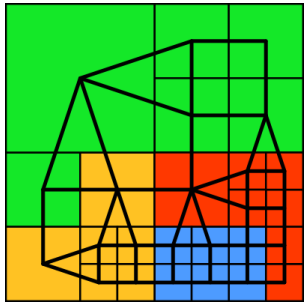
Space-Filling Curve



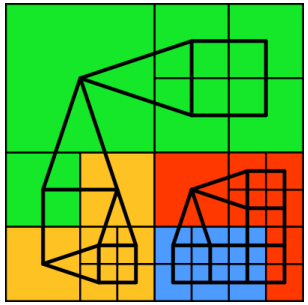
Graph-based

- Consider block decomposition as a weighted graph

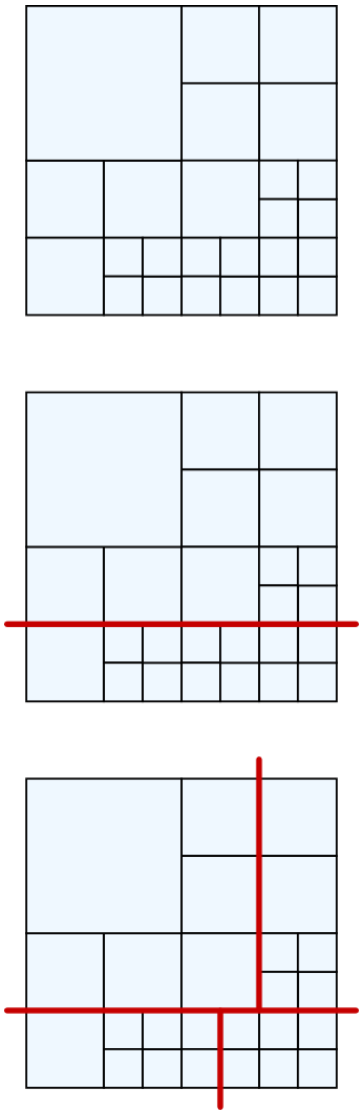
Global Graph-based



Local Graph-based

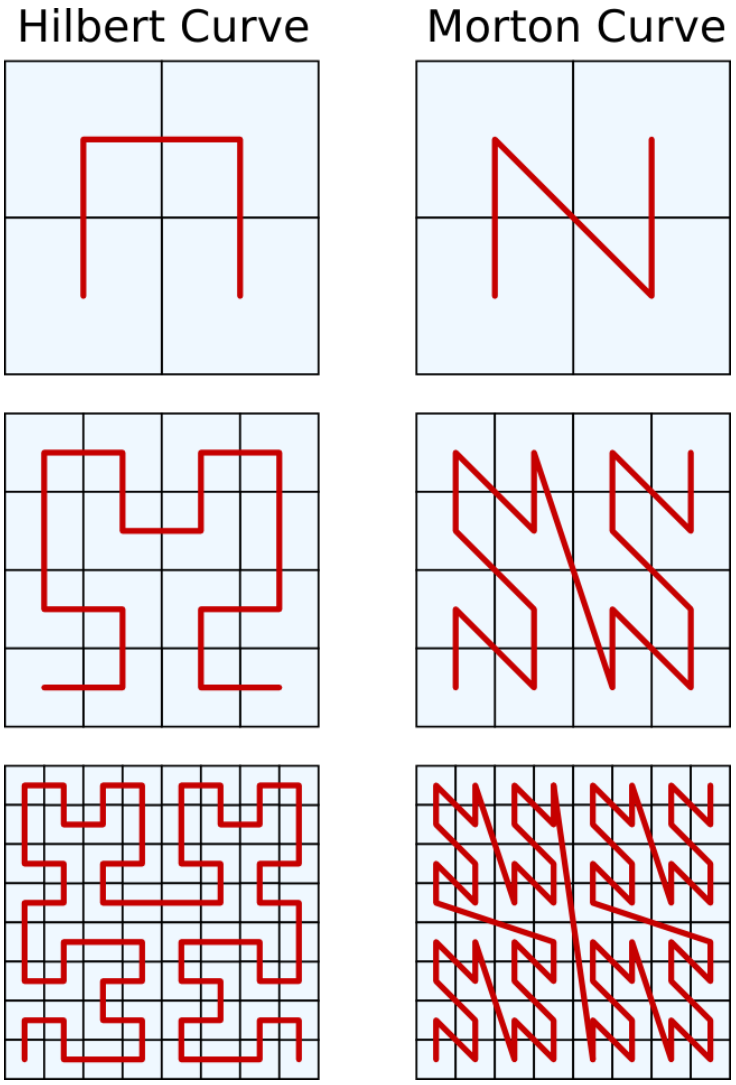


Recursive Bisection Partitioning



- **Cut the grid in two equal weighted parts**
- **Apply this algorithm recursively for each part until number of desired partitions is reached**
- **Processor count ≠ 2^n: cut in more than 2 parts**
- **Very fast algorithm**
- **Versions:**
  - → Recursive Coordinate Bisection (RCB)
  - → Unbalanced Recursive Bisection (URB)
  - → Recursive Inertial Bisection (RIB)
- **Moderate quality partitions**

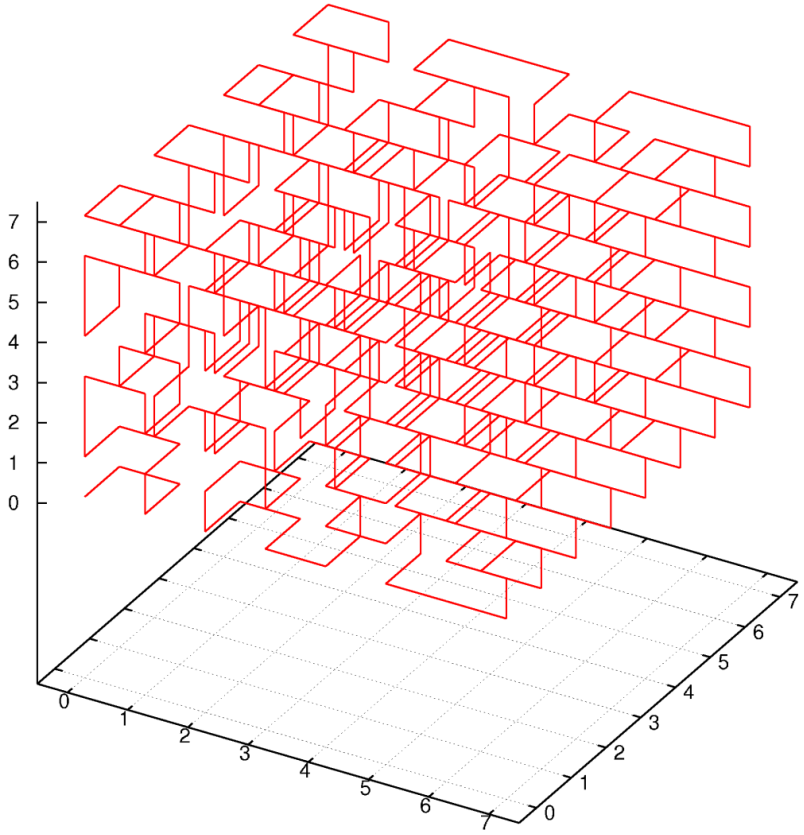
Space-Filling Curves (SFCs)



- **1D traversal of the grid**
- **nD → 1D mapping / ordering**
- **Data locality**
  - → Points close on the curve are also close in the nD grid
- **Self-similarity**
  - → Constructed recursively from a start template in *O*(log n)
- **Most prominent for load balancing:**
  - → Hilbert curve (higher locality)
  - → Morton curve (faster)

Matthias Lieber (TU Dresden)

Space-Filling Curves (SFCs)



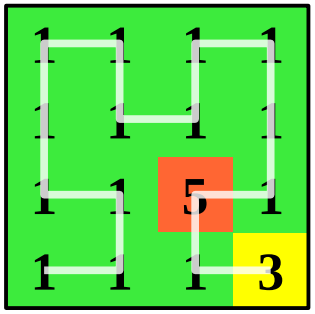
3D Hilbert Curve

- **1D traversal of the grid**
- **nD → 1D mapping / ordering**
- **Data locality**
  - → Points close on the curve are also close in the nD grid
- **Self-similarity**
  - → Constructed recursively from a start template in *O*(log n)
- **Most prominent for load balancing:**
  - → Hilbert curve (higher locality)
  - → Morton curve (faster)

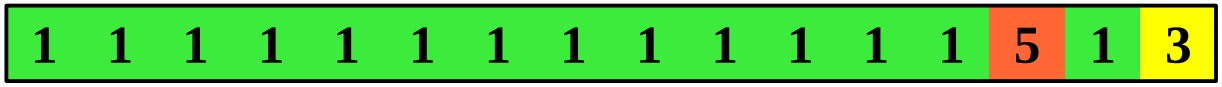
Matthias Lieber (TU Dresden)

1-D Partitioning

- **Reduction of the partitioning problem to dimension one**



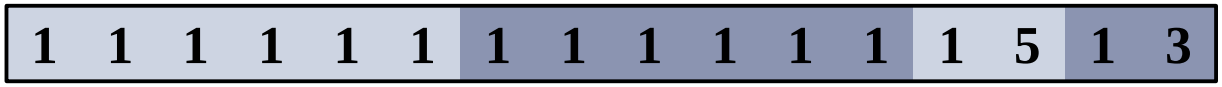
Vector of weights w_i



Total weight $\sum w_i = 22$

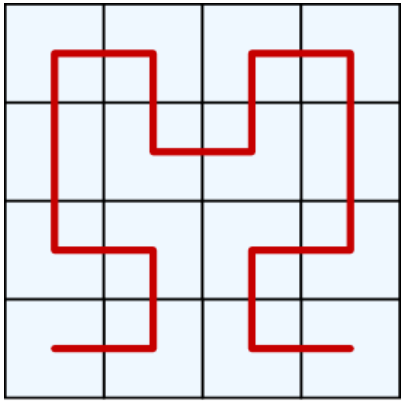
- **1-D partitioning: Create subsequent partitions of a vectors of weights to minimize the maximum weight per partition**

Optimal solution for P=4 partitions

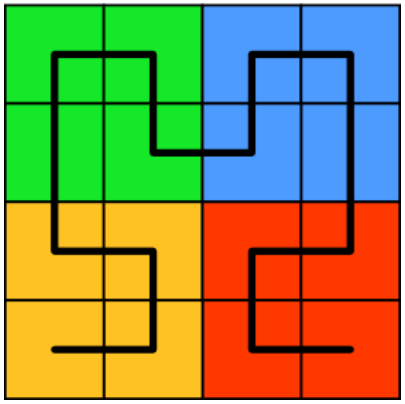


Maximum load (bottleneck) $\beta_{opt} = 6$
Loadbalance $\Lambda_{opt} = (\sum w_i / P) / \beta_{opt} = 0{,}92$

Space-Filling Curve Partitioning

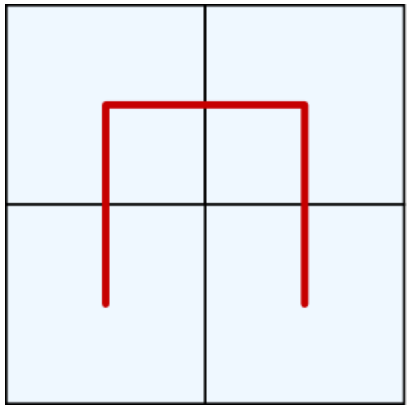


- Cut the SFC into as many equal-weighted parts as desired
- Very fast algorithm
- Moderate quality partitions

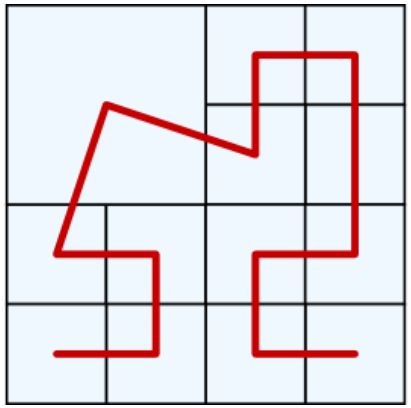


Space-Filling Curve Partitioning for AMR

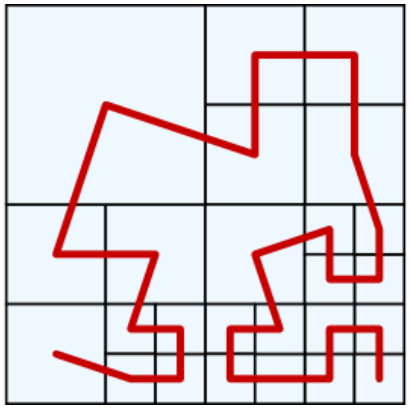
- **Space-Filling Curves are well suited for tree-based AMR due to their self-similarity**



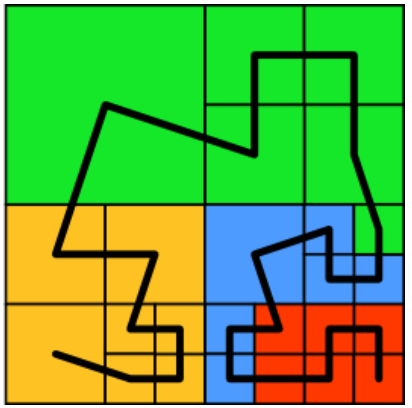
Start template



Refine



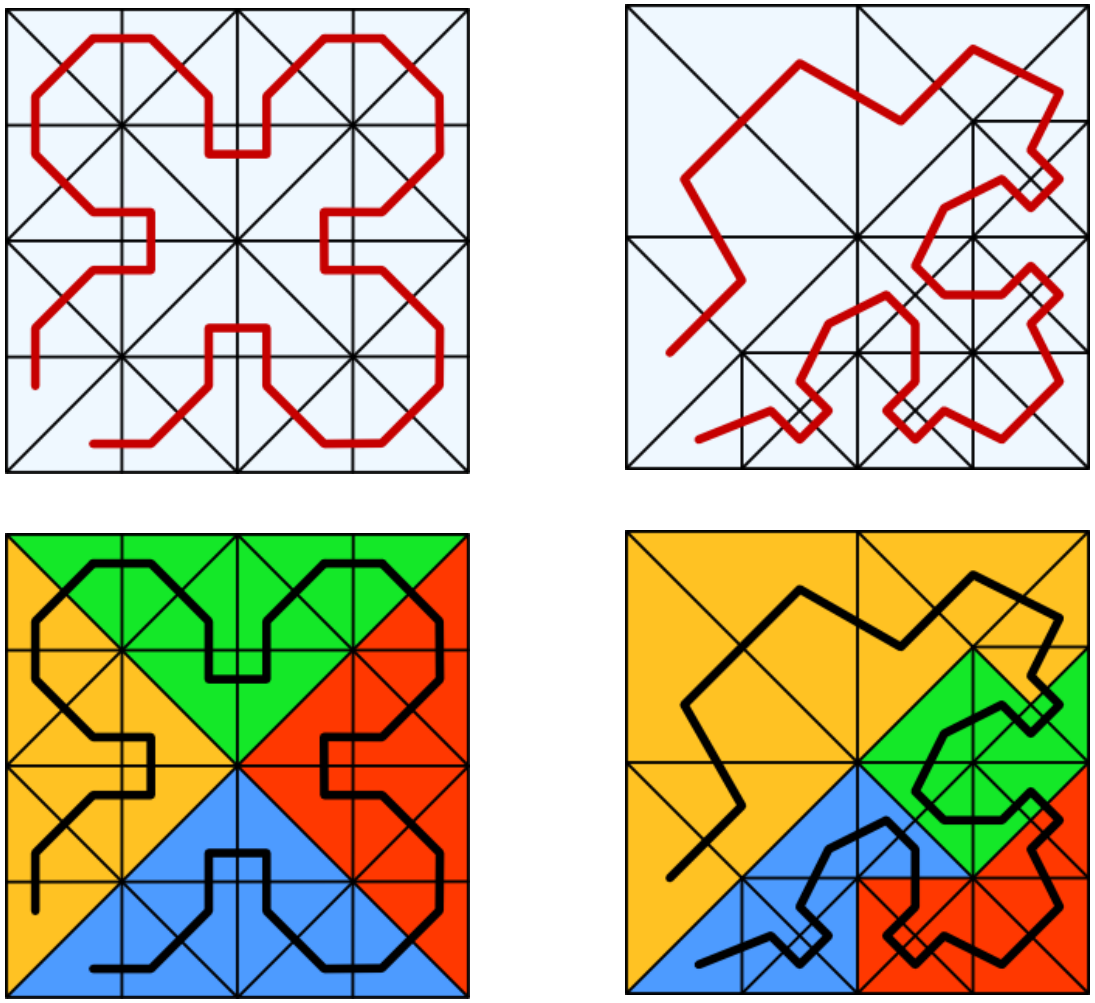
Refine



Partition

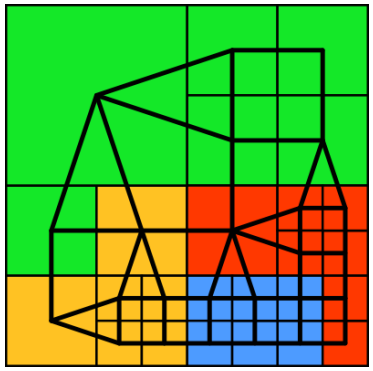
Space-Filling Curve Partitioning for Triangular Meshes

- Sierpinski Curve suited for triangular meshes



Matthias Lieber (TU Dresden)

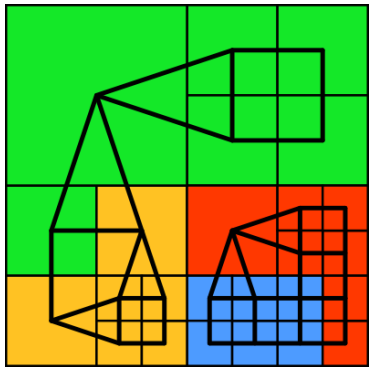
Global Graph Partitioning



- **View the decomposition as a weighted graph**
  - → Vertex weight: block workload
  - → Edge weight: communication costs between blocks
- **Explicit optimization of edge-cut**
- **Works for irregular meshes**
- **Optimal partitioning algorithms (balanced & minimized edge-cut) are NP-complete**
  - → Heuristic algorithms are used
- **Multilevel graph partitioning widely used**
  - → Very high partition quality
  - → Much slower than geometric methods
  - → Difficult to implement efficiently

Matthias Lieber (TU Dresden)

Local Graph Partitioning



- **Global graph-based methods execute slowly**
- **Local methods consider only sets of blocks**
- **Algorithms**
  - → Diffusion algorithms
  - → Demand-based algorithms
- **Faster than global graph-based methods**
- **Requires good start partitioning to reach high quality**
- **Sufficient for small workload changes**

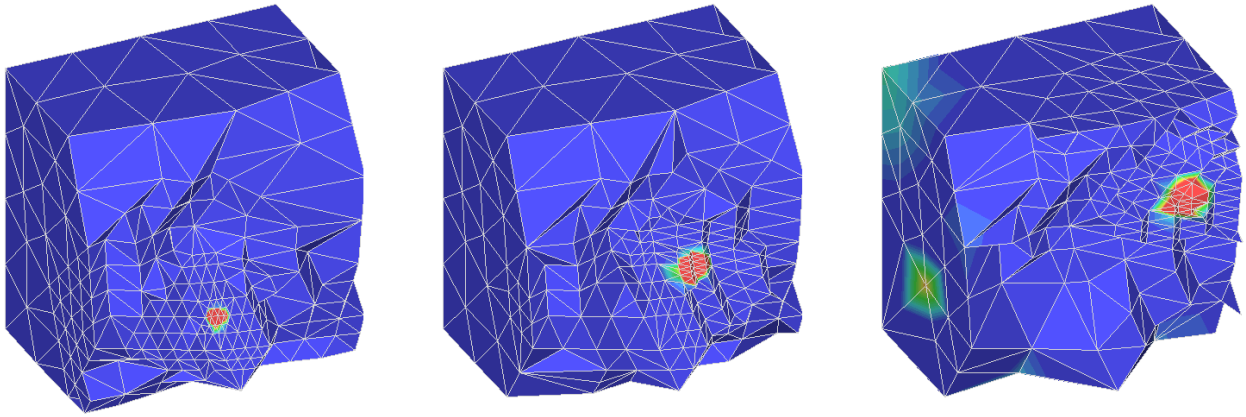
Graph Partitioning Software

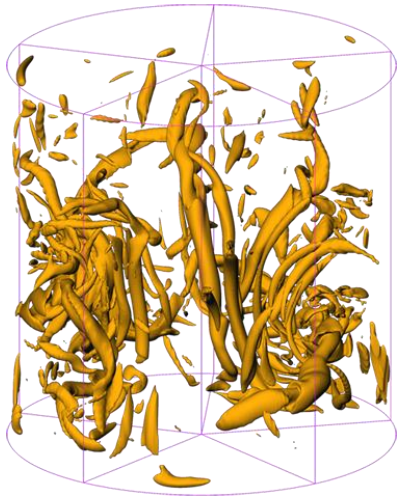
- **METIS (Karypis et al.): http://glaros.dtc.umn.edu/gkhome/software**
  - → Multilevel partitioning
  - → Multiple vertex weights (multi-constraint partitioning)
  - → ParMETIS: parallel repartitioning (multilevel & local diffusion)
- **Jostle (Walshaw et al.): http://staffweb.cms.gre.ac.uk/~c.walshaw/jostle**
  - → Multilevel partitioning and diffusion
  - → PJostle: parallel algorithms
- **Zoltan (Devine et al.): http://www.cs.sandia.gov/Zoltan**
  - → Common interface to several load balancing methods (recursive bisection, SFC, ParMETIS, and PJostle)
  - → Other services like data migration tools

Example: MG Library (Stiller et al., ISM, TU Dresden)

- Library for flow solvers on unstructured grids
- Tetrahedral mesh
- Mesh refinement
- Dynamic load balancing, multilevel partitioning using METIS
- Supports spectral elements

- Used for DNS of magneto-fluiddynamical processes
- Electromagnetic stirring with rotating magnetic fields
- SFB 609

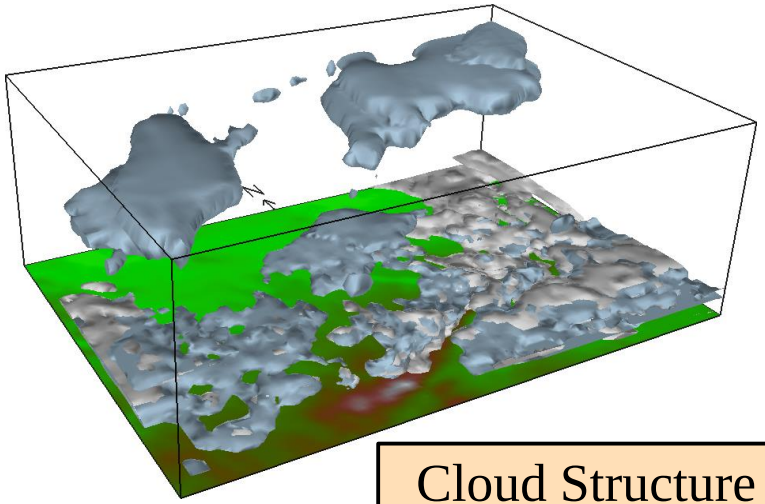




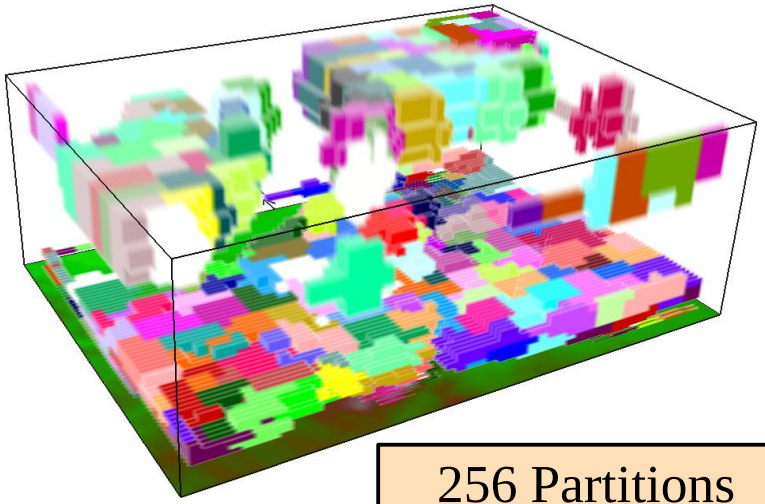
Fladrich, Brunst, Stiller: *Analyzing the Memory Access Pattern of a Spectral Element Method Implementation* (2003)

Matthias Lieber (TU Dresden)

Example: Detailed Cloud Modeling in Atmospheric Models



Cloud Structure



256 Partitions

- **Detailed model of cloud microphysical processes coupled to weather forecast model COSMO (IfT Leipzig)**
- **Microphysics model is *very costly* in terms of runtime and memory requirements**

Adaptive approach:

- **Only allocate / compute grid blocks that contain clouds**
- **Dynamically balance these blocks over all processors during runtime**

Matthias Lieber (TU Dresden)

Further Literature

- J. Teresco, K. Devine, J. Flaherty: *Partitioning and Dynamic Load Balancing for the Numerical Solution of Partial Differential Equations*. In: Numerial Solution of Partial Differential Equations on Parallel Computers, Springer, 2005.
- L.F. Diachin, R. D. Hornung, P. Plassmann, A. M. Wissink: *Parallel Adaptive Mesh Refinement.* In: Parallel processing for scientific computing, Cambridge University Press, 2006
- G. Zumbusch: *Parallel Multilevel Methods*, Teubner, 2003
- C. Burstedde et al: *Scalable Adaptive Mantle Convection Simulation on Petascale Supercomputers.* SC08, 2008
- R. D. Hornung, A. M. Wissink, S. R. Kohn: *Managing complex data and geometry in parallel structured AMR applications.* Engineering with Computers, 2006
- J. L. Vay et al: *Application of adaptive mesh refinement to particle-in-cell simulations of plasmas and beams*, 2004

Matthias Lieber (TU Dresden)

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. **Parallelization and optimization strategies**
   - Types of parallelism
     - In hardware: processor, node & system level parallelism
     - In software: data & task parallelism
     - In algorithms: dwarfs, parallel patterns, design spaces
   - **Parallel patterns & design spaces**
     - Algorithm structure
       - Divide and conquer
       - Geometric decomposition
       - Adaptive mesh refinement
     - **Supporting structures**
       - Load balancing
       - **SPMD**
       - Master/ worker
       - Loop parallelism
       - Fork/ join
   - Case study: Molecular dynamics
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

SPMD = Single Program, Multiple Data

- **(Fortunately) For most algorithmic cases, the computations carried out on each of the individual processors (processing units) are similar**
  - → Data might be different between processors
  - → Even slightly different computations may occur (e.g. boundary conditions in PDE solvers)
- **Bringing all the algorithmic logic together into one source tree is more manageable for the programmer in most scenarios**
  - → Approach is called “Single program, multiple Data”
  - → Dominant way of programming parallelism; MPI for example is especially designed towards this approach
    - → By far most commonly used pattern for structuring parallel programs

Considerations

- **Using similar code for each processing unit is easier for the programmer**

- **Software is used longer than parallel computers**
  - → Programs should be portable

- **High scalability and hence good efficiency is achieved by**
  - → Aligning a program well to the underlying architecture
  - → Controlling details of the parallel system by the programmer (where appropriate)

Structure of most SPMD programs (1/2)

1. **Initialization**
   - → Program is loaded onto different processing units
   - → Program establishes communication channels with other units
2. **Obtaining a unique identifier**
   - → At the top of the source tree, the program usually acquires a unique identifier inside the parallel context
   - → Usually this is the rank of the corresponding MPI group
3. **Distribute data**
   - → Data is either decomposed into chunks and distributed *or*
   - → Data is shared/replicated over several UE's

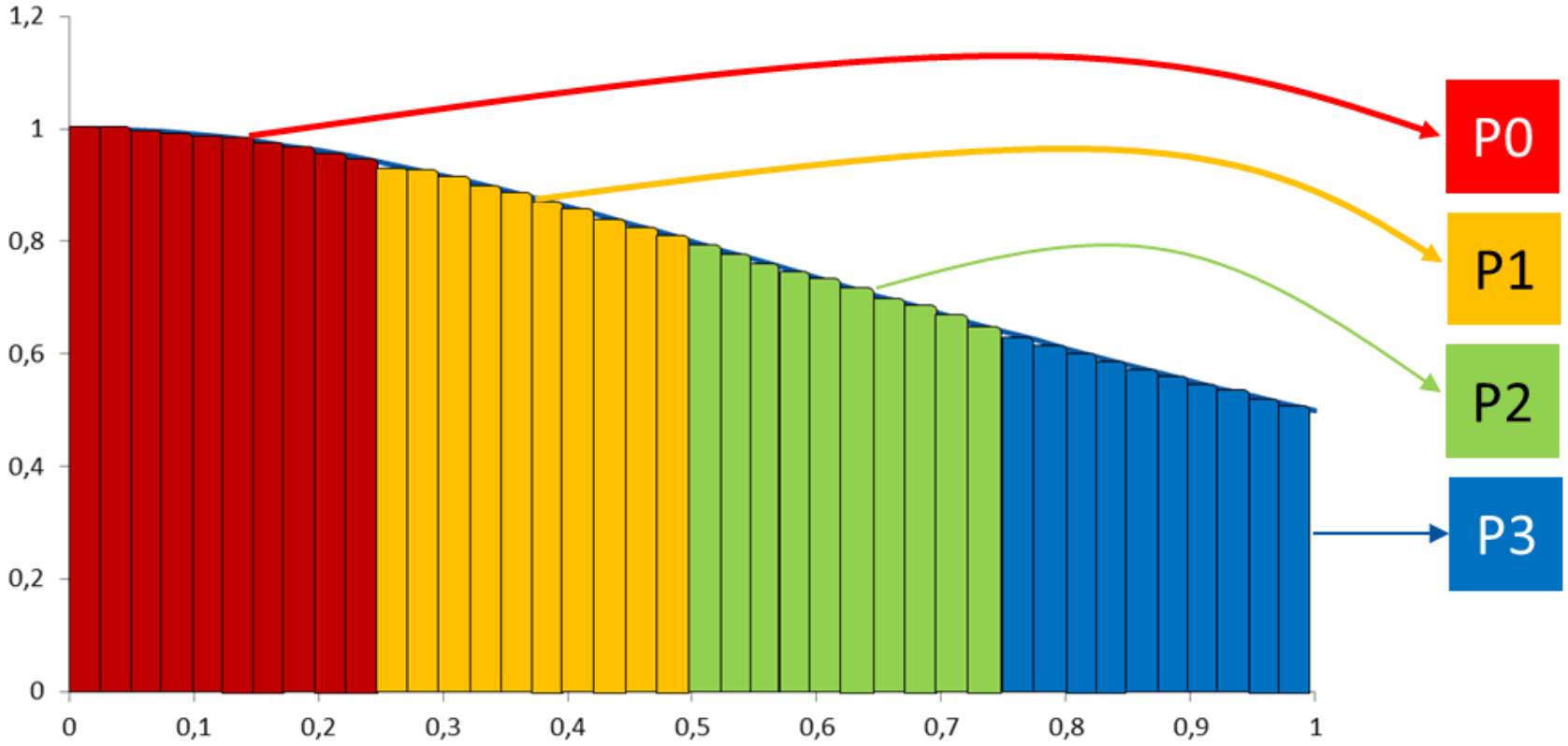
Structure of most SPMD programs (2/2)

4. **Execution**
   - → Run the same program on UE
   - → Use the unique ID to initiate different behavior on different execution units
5. **Finalize**
   - → Program closes by cleaning up the shared context and shutting down the computation
   - → If data needs to be recombined, one or multiple UEs use collective operations to do so

Example

- **Example code for SPMD on next slide**

→ Integration of $4\tan^{-1}(1) = \int_0^1 \frac{4}{1+x^2} = \pi$



Example (pseudocode)

```
void main()
{
  int rank, size, chunk, i_start, i_end;
  const int numsteps = 100000;
  Initialize();
  Get_Rank(&rank);
  Get_Size(&numprocs);
  chunk = numsteps/numprocs;
  i_start = rank*chunk;
  i_end = (rank+1)*chunk;
  if(rank == numprocs – 1) { i_end = numsteps; }
  double sum = 0.0;
  for(int i = i_start; i < i_end; ++i)
  {
    double x = (0.5+i)/numsteps;
    sum += 4.0/(1.0+x*x);
  }
  sum /= numsteps;
  double pi;
  Collect(&sum, &pi, 0);
  if( rank == 0 )
  {
     printf(“PI is %f\n”, pi);
  }
}
```

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. **Parallelization and optimization strategies**
   - Types of parallelism
     - In hardware: processor, node & system level parallelism
     - In software: data & task parallelism
     - In algorithms: dwarfs, parallel patterns, design spaces
   - **Parallel patterns & design spaces**
     - Algorithm structure
       - Divide and conquer
       - Geometric decomposition
       - Adaptive mesh refinement
     - **Supporting structures**
       - Load balancing
       - SPMD
       - **Master/ worker**
       - Loop parallelism
       - Fork/ join
     - Case study: Molecular dynamics
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Motivation

- **How should a program be designed with need of dynamic load balancing among the available UEs?**

- **Parallel efficiency results out of an algorithm's parallel overhead, the serial fraction of execution time and the load balancing**

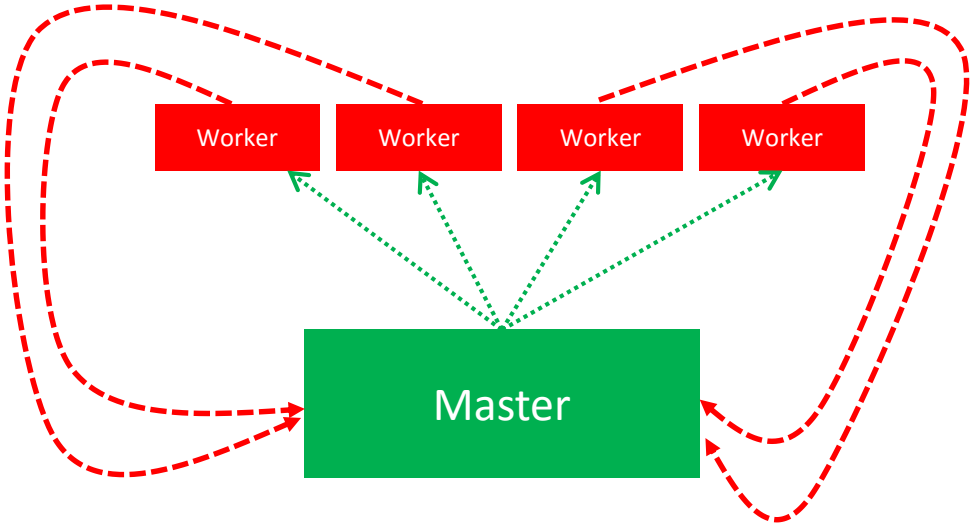
  → Sometimes load balancing is so difficult that it dominates the program's design

  1. Workload associated with tasks is highly variable/ unpredictable
  2. Computationally intensive parts don't map to simple loops
  3. The system is heterogeneous in it's compute capabilities

Master/Worker

- **Forces that imply usage of the Master/Worker pattern**
  - → Work for each task varies unpredictably
  - → Operations to balance the load impose expensive communication overhead
  - → Logic to produce an optimal load balance can be obfuscating & error-prone

- **Master/Worker-pattern: Who does the actual computations? Who is responsible for worksharing?**
  - → Usually 1 *master* initiates computation and sets up the problem
  - → 1 or more instances of a *worker* process the computational tasks



Master/Worker

- **Classic approach: the master…**
  - → Waits for the workers to complete computation
  - → Collects the results
  - → Shuts down computation
- **Easiest way to implement the pattern: Shared Queue**
  - → Tasks are inserted into the Queue by the master process
  - → Workers acquire tasks out of the queue when finished with previous work
  - → If the queue is empty the computation is finished

Example (pseudocode)

- **Assume: shared address space for master & workers**

  → Tasks/ results in queue are visible to all UEs

master

```
int Nworkers = 10;
// initialize queues
SharedQueue work_queue; // task queue
SharedQueue global_results; // result queue
void worker();

void main()
{
  // put N tasks into queue
   for(int i=0; i<N, ++i)
        enqueue(work_queue,i);
   // launch workers & wait for them to finish
   ForkJoin(Nworkers, worker);
   consume_results(); // gather results
}
```

worker

```
void worker()
{
  int i;
  Result res;
  // loop until task queue is empty
  while(!empty(SharedQueue))
  {
     // get new task from queue
     i = dequeue(SharedQueue);
     // do actual work
     res = do_lots_of_work(i);
     // store results in global result queue
     enqueue(global_results, res);
  }
}
```

Dynamic work distribution

- **Dynamic: If number of workers << number of tasks**
  - → Automatically good load balancing
    - →Even if computational expenses for individual tasks is not known beforehand
    - →Even with inhomogeneous parallel architectures
- **Challenges**
  - → Runtime is dominated by computation and the amount of workers is of moderate size
    - →Master process may also participate as worker
  - → Amount of workers is of large size and the communication to the master process may become a bottleneck
    - →Several master processes might be instantiated

Utilization

- **Well known-examples for utilization of the Master/Worker pattern**
  - → The OpenMP tasks construct
  - → Projects like SETI@Home, Folding@Home, …
  - → Manual MPI implementation
    - →Although the implementation of worksharing over a global queue is more complicated
- **Master/Worker is closely related to the Loop parallelism pattern if the loop scheduling is of dynamic nature**
  - → E.g. schedule(dynamic) in OpenMP

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. **Parallelization and optimization strategies**
   - Types of parallelism
     - In hardware: processor, node & system level parallelism
     - In software: data & task parallelism
     - In algorithms: dwarfs, parallel patterns, design spaces
   - **Parallel patterns & design spaces**
     - Algorithm structure
       - Divide and conquer
       - Geometric decomposition
       - Adaptive mesh refinement
     - **Supporting structures**
       - Load balancing
       - SPMD
       - Master/ worker
       - **Loop parallelism**
       - Fork/ join
   - Case study: Molecular dynamics
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Loop parallelism

- **A majority of scientific codes are expressed in terms of iterative constructs – they are loop-based**

- **Traditional approach: strict focus on the loops in such a code**
  - → Loops usually account for most of the computational costs
  - → Vector computers do so as well

- **Loop-parallelism pattern can often be used for existing programs**
  - → Code can be parallelized even if convoluted (or not fully understood)
  - → Dominant pattern in HPC

Loop parallelism

- **Iterations of loops are identified as tasks**
  - → If loop iterations are independent: parallel execution possible
- **Amdahl's Law applies**
  - → Parallelization of loops will only scale with many processors if most of the program's runtime can be efficiently parallelized
  - → But, loop-parallelism is often only effective with a relative small number of threads
    - →In orders of magnitude there exist more parallel machines with 2, 4, … cores than with hundreds of cores
- **Early OpenMP was designed primarily to support loop-based parallelism**
  - → Particularly relevant in context to OpenMP and Shared Memory parallelism

Basic approach for loop parallelism

1. **Find the most computationally expensive loops**
   - → By inspection of the code
   - → By understanding the performance needs of each subproblem
   - → By using performance analysis tools as Intel Vtune, Vampir, Scalasca, gprof, etc…

2. **Eliminate loop dependencies**
   - → Loop iterations need to be nearly independent for parallelization
   - → Find dependencies between iterations and remove or mitigate them

Basic approach for loop parallelism

3. Parallelize the loops

- → Split up the operations among the threads one loop at a time
- → It might be beneficial to jam multiple loops to increase the computational costs per (parallelized) loop iteration
- → Dependencies and race conditions might be discovered only after parallelization

4. Optimize the loop schedule

- → Iterations must be scheduled for execution in a way that the load is balanced most evenly among threads
- → There is room for experimentation

Example (Pi)

- **Example code uses OpenMP parallel for directive for loop-parallelism**

```
void main()
{
  int numsteps = 100000;
  double pi = 0.0;
 // distribute work of loop across threads
  #pragma omp parallel for reduction(+,pi)
  for( int i = 0; i < numsteps; ++i )
  {
    double x = (0.5+i)/numsteps;
    pi += 4.0/(1.0+x*x);
  }
  pi /= numsteps;
  printf("%f\n", pi);
}
```

process/ thread 1
process/ thread 2
process/ thread 3
process/ thread 4

Work distribution with static scheduling

But there are other scheduling clauses in OpenMP (dynamic, guided, …).

Example work distribution with dynamic scheduling

Considerations

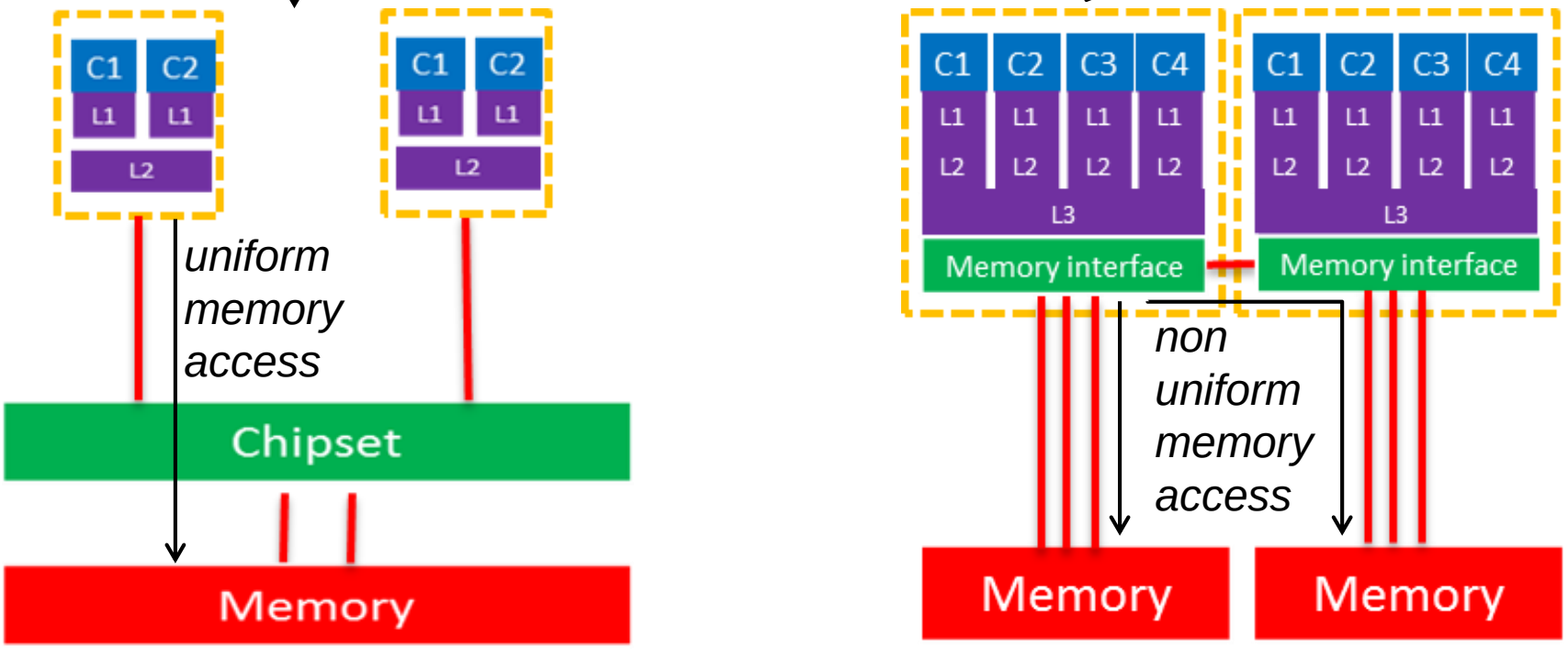
- **Sequential equivalence**

- **Incremental parallelism**
  - → Introducing parallelism one loop at a time is less likely to “break” the program
  - → Detecting errors if one transformation “breaks” the correctness of the program is also easier as one has only to reconsider the last recent modifications

- **Memory utilization**
  - → To achieve good performance, the data access patterns of the loops have to adapt well to the memory hierarchy of the underlying parallel system
  - → Restructuring might be needed if this is not the case

Loop parallelism

- **Common assumption: all UEs share a virtual address space**
  - → True, but what about memory access time?
  - → Recap: UMA & NUMA architecture → impact on performance!

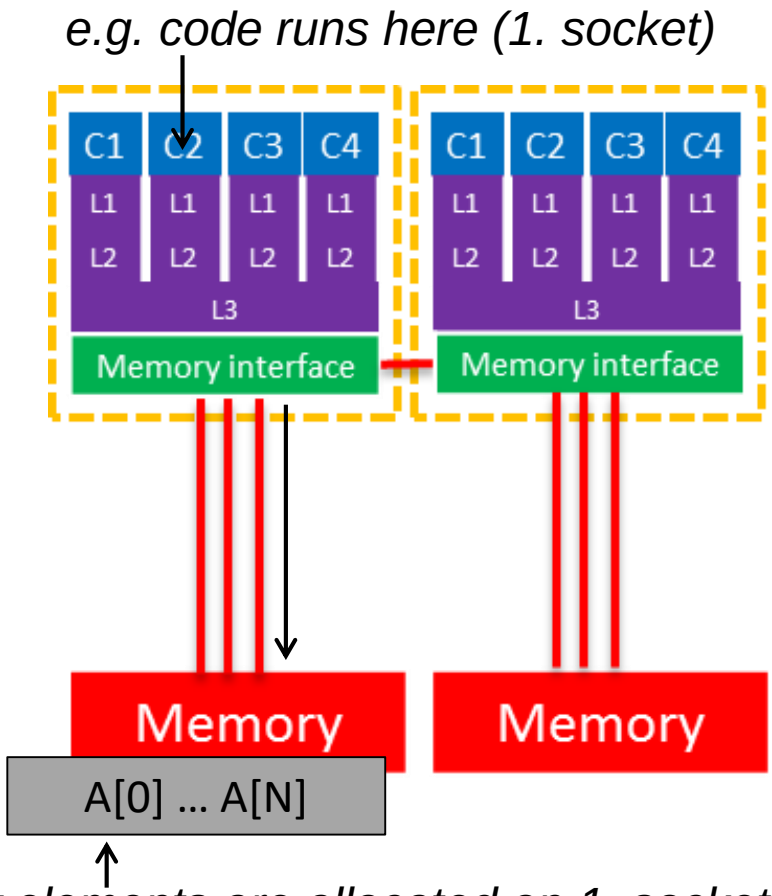


Excursus on NUMA affinity: Memory placement

- **Serial code**

  → All array elements are allocated in the memory of the NUMA node containing the core executing this thread

```
double* A;
A = (double*)
    malloc(N * sizeof(double));



for (int i = 0; i < N; i++){//init
   A[i] = 0.0;
}

for (int i = 0; i < N; i++){//compute
   A[i] += i;
}
```



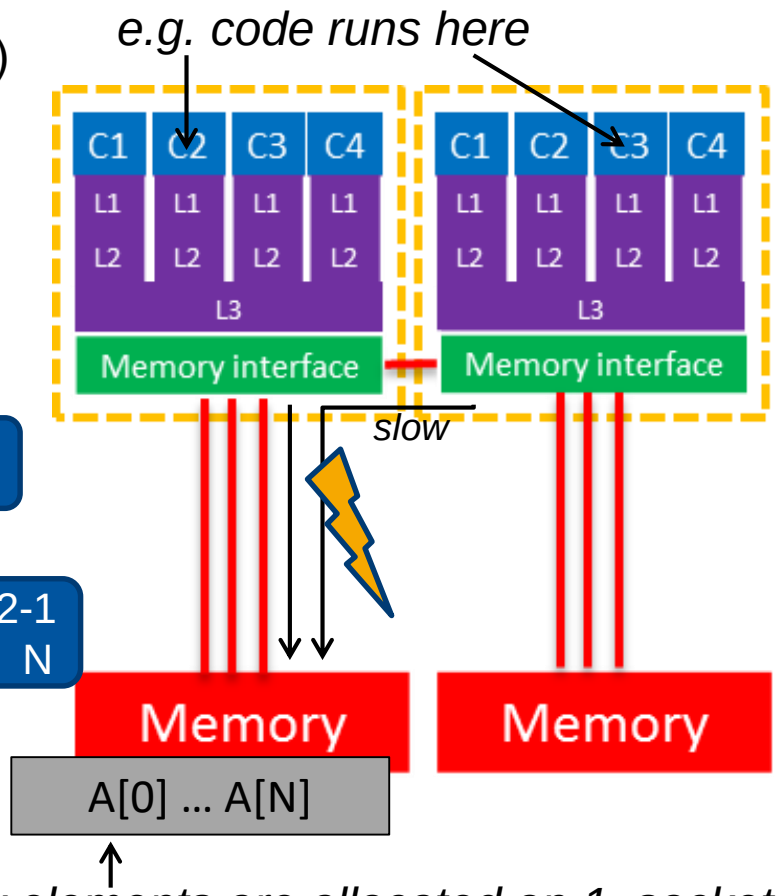
Excursus on NUMA affinity: Memory placement

- **Parallel code (e.g. loop parallelism)**
  - → All array elements are allocated in the memory of the NUMA node containing the core executing the thread initializing the respective partition (“lazy allocation”)
  - → Called “first touch” policy

```
double* A;
A = (double*)
    malloc(N * sizeof(double));

set_num_threads(2);

for (int i = 0; i < N; i++){//init
   A[i] = 0.0;
}
//Parallel: share work among threads
for (int i = 0; i < N; i++){//compute
   A[i] += i;
}
```

2 threads

Thread 1: 0 - N/2-1
Thread 2: N/2 - N

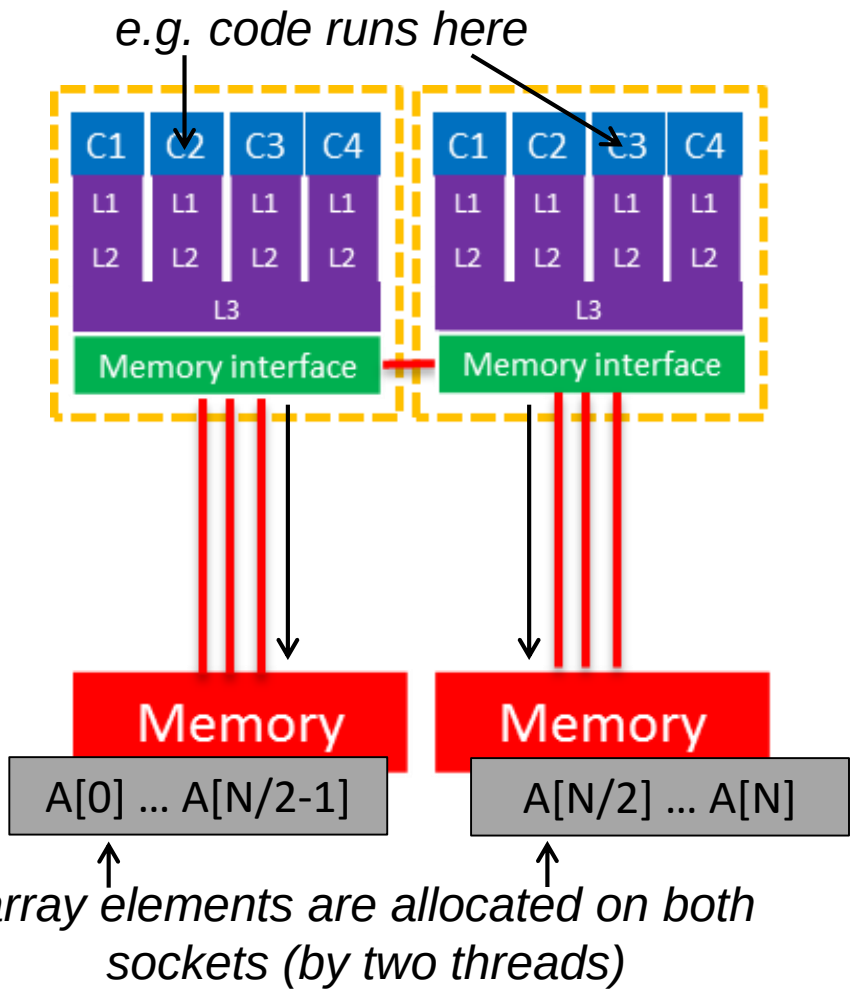


array elements are allocated on 1. socket (by master thread)

Excursus on NUMA affinity: Memory placement

- **Parallel code (e.g. loop parallelism)**
  - → Wanted: distribution of data to different NUMA domains (here: sockets)
  - → At best: same data distribution as work distribution!
  - → Often achieved by parallel initialization (see lecture on OpenMP)

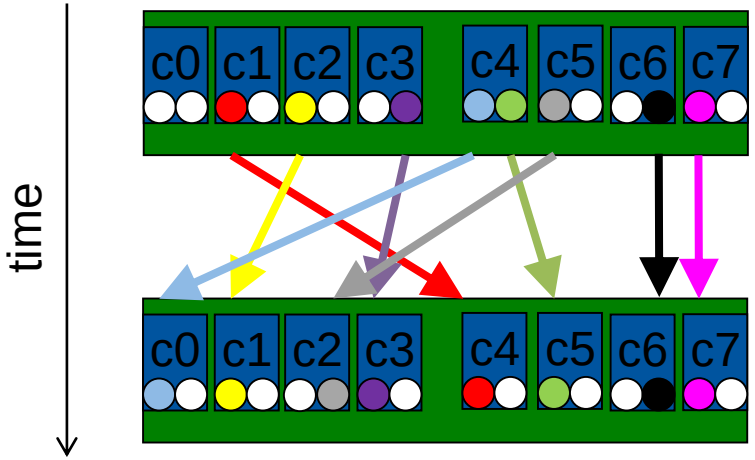
```
double* A;
A = (double*)
    malloc(N * sizeof(double));

set_num_threads(2);
//Parallel: share init among threads
for (int i = 0; i < N; i++){//init
   A[i] = 0.0;
}
//Parallel: share work among threads
for (int i = 0; i < N; i++){//compute
   A[i] += i;
}
```



Excursus on NUMA affinity: Thread/ Process Migration

- **So far: assumption that thread/ process runs on a fixed core**
- **But, default: OS can migrate threads/ processes across/within (physical) cores**

→ Linux, Windows



Example

2-socket system with 8 cores

each core with 2 hyper threads

#threads to run = 8

- **May have impact on performance (for single & multi-threaded/process)**
  - → No data locality (non-uniform memory accesses likely)
  - → Overhead by context switch
- **Solution: Bind / pin threads/ processes to hardware**

Excursus on NUMA affinity: Get Info on the System Topology

- **Understanding of system topology needed to design a thread/ process binding strategy**

  → Intel MPI's **`cpuinfo`** tool

  →#sockets (= packages),

  mapping of processor ids used by the operating system to CPU cores

  →**`hwloc-ls`** or **`lstopo`** tool (comes with Open-MPI)

  →Graphical representation of system topology (separated into NUMA nodes),

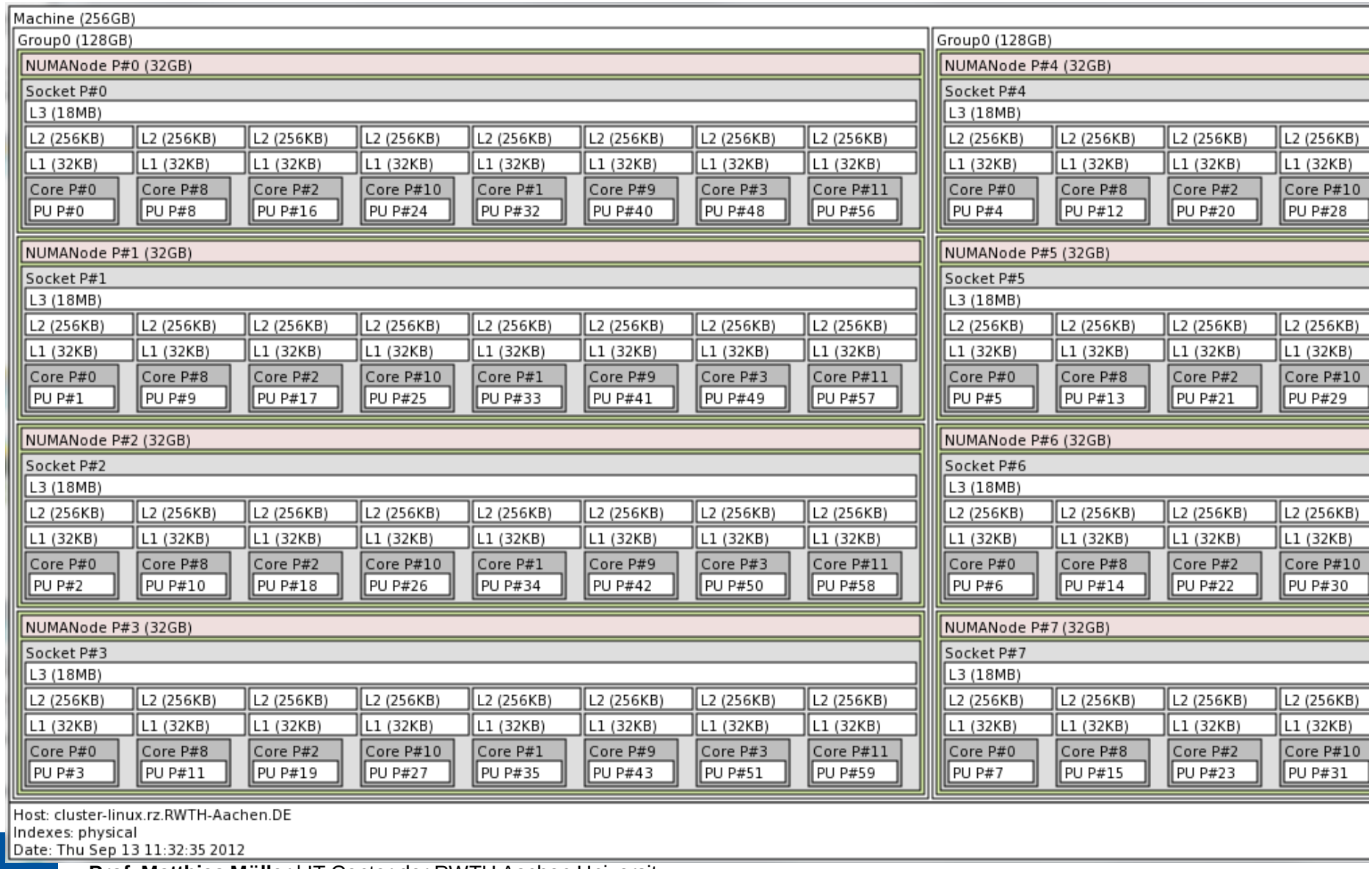
  mapping of processor ids used by the operating system to CPU cores,

  additional info on caches

  → RRZE's **`likwid-topology [-g]`**

  →Shows the thread and cache topology

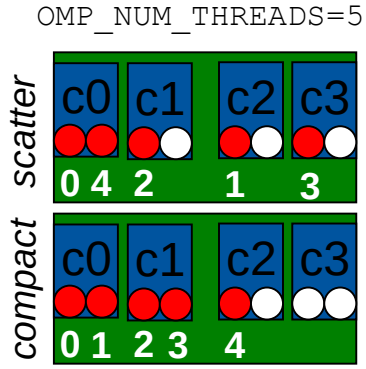
Excursus on NUMA affinity: Example
hwloc-ls



Excursus on NUMA affinity: Decide for a binding strategy

- **“Right” binding strategy depends not only on the topology, but also on the characteristics of your application**
  - → Putting threads/ processes far apart, i.e. on different sockets
    - →May improve the aggregated memory bandwidth available to your app
    - →May improve the combined cache size available to your app
    - →May decrease performance of synchronization constructs
  - → Putting threads/ processes close together, i.e. on two adjacent cores which possibly share some caches
    - →May improve performance of synchronization constructs
    - →May decrease the available memory bandwidth and cache size
- **If you are unsure, just try a few options and then select the best one**

Excursus on NUMA affinity: Implement the binding strategy (threading)



- **Intel C/C++/Fortran Compiler**
  - → Use environment variable `KMP_AFFINTIY`
    - → `KMP_AFFINITY=`**`scatter`**: Put threads far apart
    - → `KMP_AFFINITY=`**`compact`**: Put threads close together
    - → `KMP_AFFINITY=<core_list>`: Bind threads in the order in which they are started to the cores given in the list, one thread per core.
    - → Add "`,verbose`" to print out binding information to stdout
- **GNU C/C++/Fortran Compiler**
  - → Use environment variable GOMP_CPU_AFFINITY
    - → `GOMP_CPU_AFFINTIY=<core_list>`: Bind threads in the order in which they are started to the cores given in the list, one thread per core.
- **Linux Systems**
  - → Restrict a process to a subset of cores (no binding!): `taskset -c corelist`

In the OpenMP lecture, you will learn about a portable way (*places*).

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. **Parallelization and optimization strategies**
   - Types of parallelism
     - In hardware: processor, node & system level parallelism
     - In software: data & task parallelism
     - In algorithms: dwarfs, parallel patterns, design spaces
   - **Parallel patterns & design spaces**
     - Algorithm structure
       - Divide and conquer
       - Geometric decomposition
       - Adaptive mesh refinement
     - **Supporting structures**
       - Load balancing
       - SPMD
       - Master/ worker
       - Loop parallelism
       - **Fork/ join**
   - Case study: Molecular dynamics
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Fork/Join

- **Problem: number of concurrent tasks varies within program**
  - → Program may execute in a way that prevents the use of simple control structures such as loop-parallelism
- **Often: relationships between tasks are simple**
  - → Efficient parallelization by using either loop-parallelism or the Master/Worker pattern
- **Sometimes: relationships between tasks must be captured in the way tasks are created**
  - → Examples: Recursive creation of tasks as in Divide-and-Conquer algorithms (Quicksort, Mergesort, …)
  - → **Fork/ Join pattern**

Example (pseudocode)

```
void sort(int *A, int lo, int hi)  {
  // cancellation condition omitted
  int pivot = (lo+hi)/2;
  #pragma omp parallel
  {
    #pragma omp task  // fork new task
    {
      sort(A, lo, pivot);
    }
    #pragma omp task  // fork new task
    {
      sort(A, pivot, hi);
    }
  } // implicit barrier → join tasks
  // merge sorted arrays
  int n = hi-lo;
  int ws[] = new int[n];
  Copy(A,lo,ws,0,n);
  int wlo = 0, wpivot = pivot-lo, whi = wpivot;
  for(int i = 0; i < n; ++i)  {
    if( (wlo <= wpivot) && (whi >=n || ws[wlo] <= ws[whi])
      A[i] = ws[wlo++];
    else
      A[i] = ws[whi++];
}  }
```

- **Parallel merge sort with OpenMP**
  - → Recursive code