Workloads

Programmierung Paralleler und Verteilter Systeme (PPV)

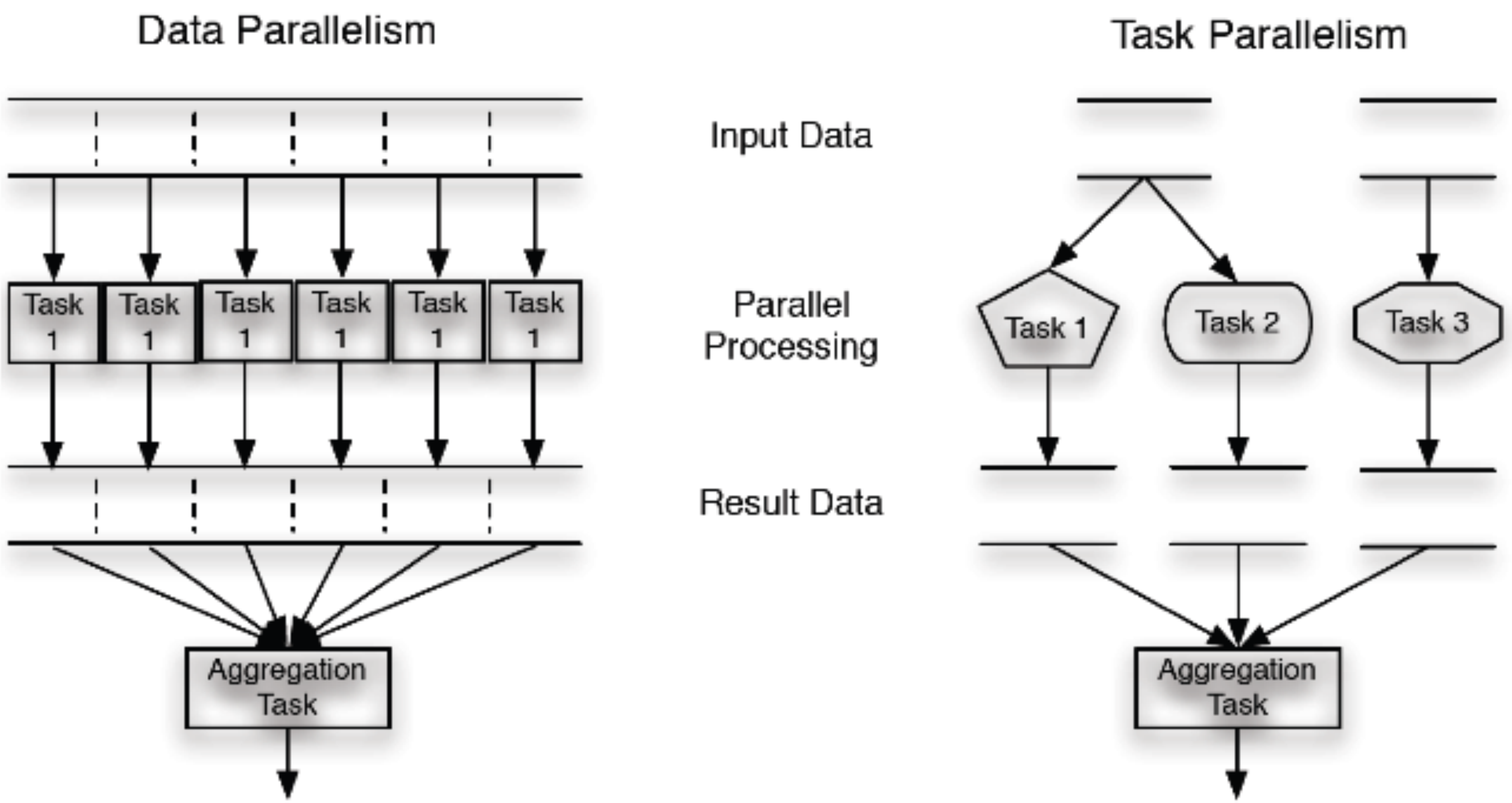
Sommer 2015

Frank Feinbube, M.Sc., Felix Eberhardt, M.Sc.,
Prof. Dr. Andreas Polze

Workloads

- Hardware / software execution environment are typically designed and optimized for specific workload
- **„task parallel workload“**
  - Different tasks being performed at the same time
  - Might originate from the same or different programs
- **„data parallel workload“**
  - Parallel execution of the same task on disjoint data sets
- Sometimes also **„flow parallelism“** added
  - Overlapping work on data stream
  - Examples: Pipelines, assembly line model
- Task / data size can be **coarse-grained** or **fine-grained**
  - Decision of algorithm design and / or configuration
  - No common semantics for these terms

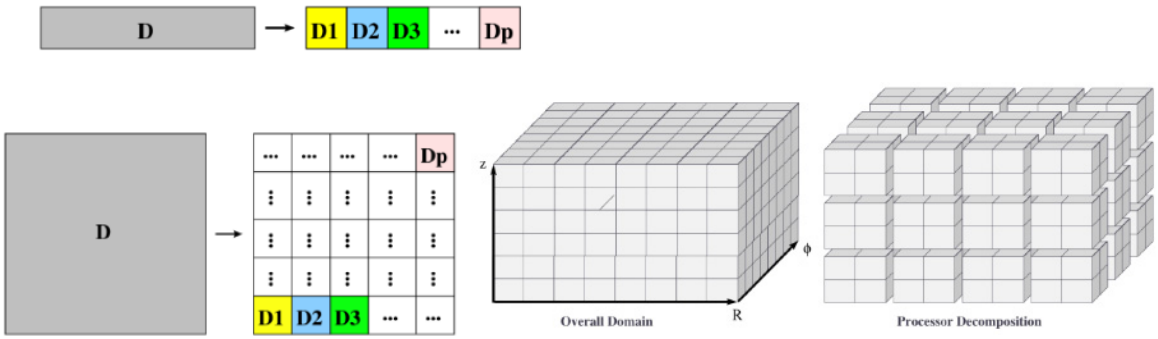
Workloads



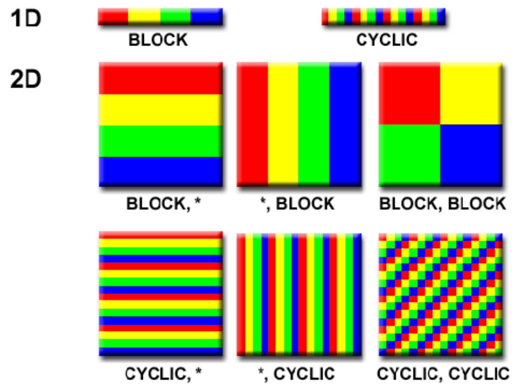
Data Parallelism
Task Parallelism
Input Data
Task 1
Task 1
Task 1
Task 1
Task 1
Task 1
Parallel Processing
Task 1
Task 2
Task 3
Result Data
Aggregation Task
Aggregation Task

Workloads

- Data Parallelism

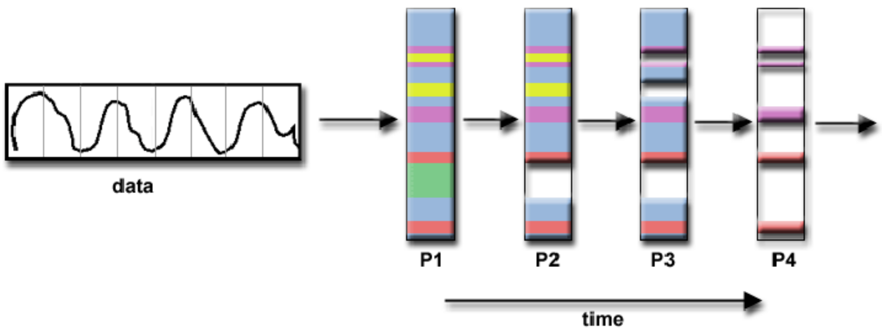


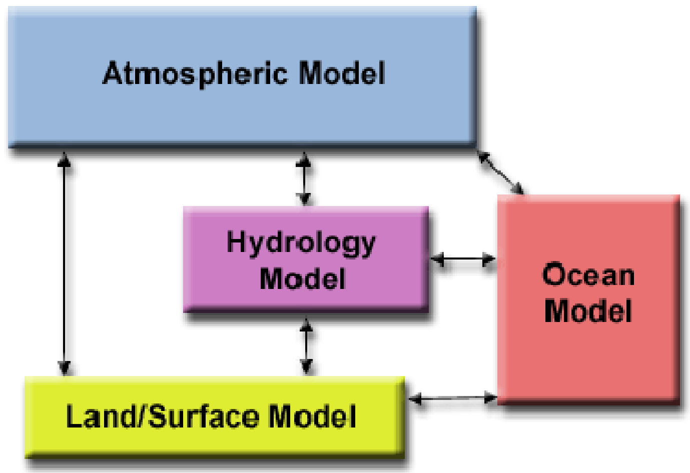
[4] 2013 SMU HPC Summer Workshop



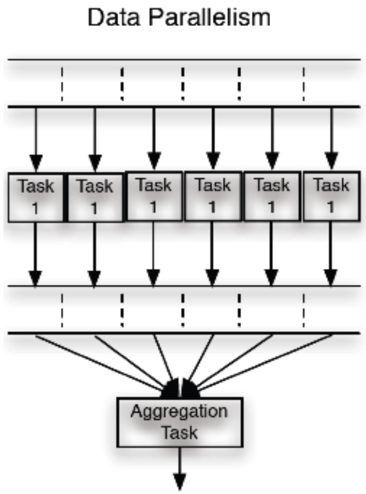
[5] Parallel Computing Tutorial

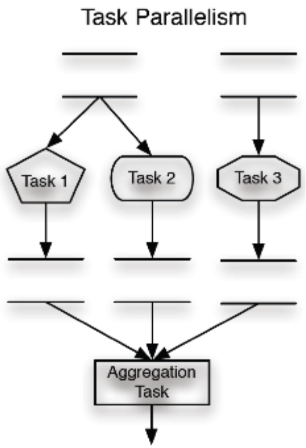
- Functional Parallelism

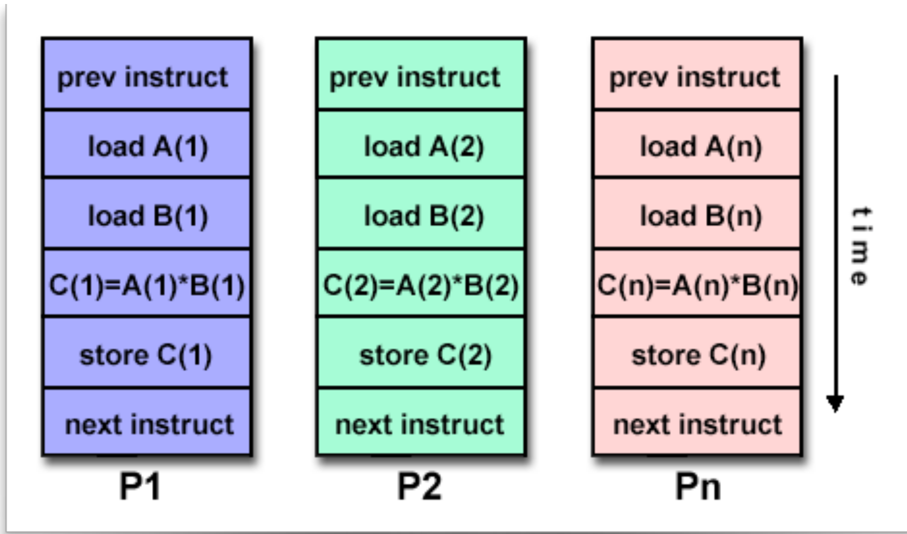




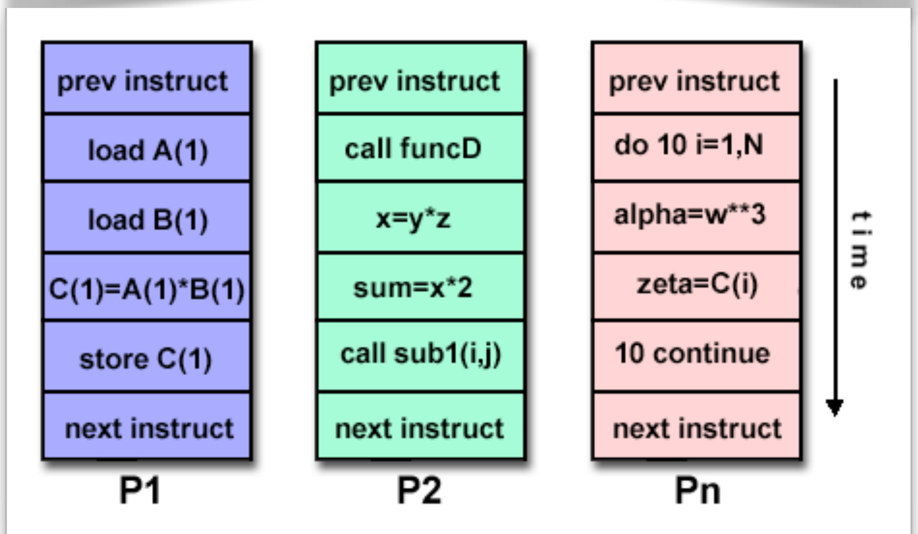
Execution Environment Mapping







Single Instruction, Multiple Data (**SIMD**)



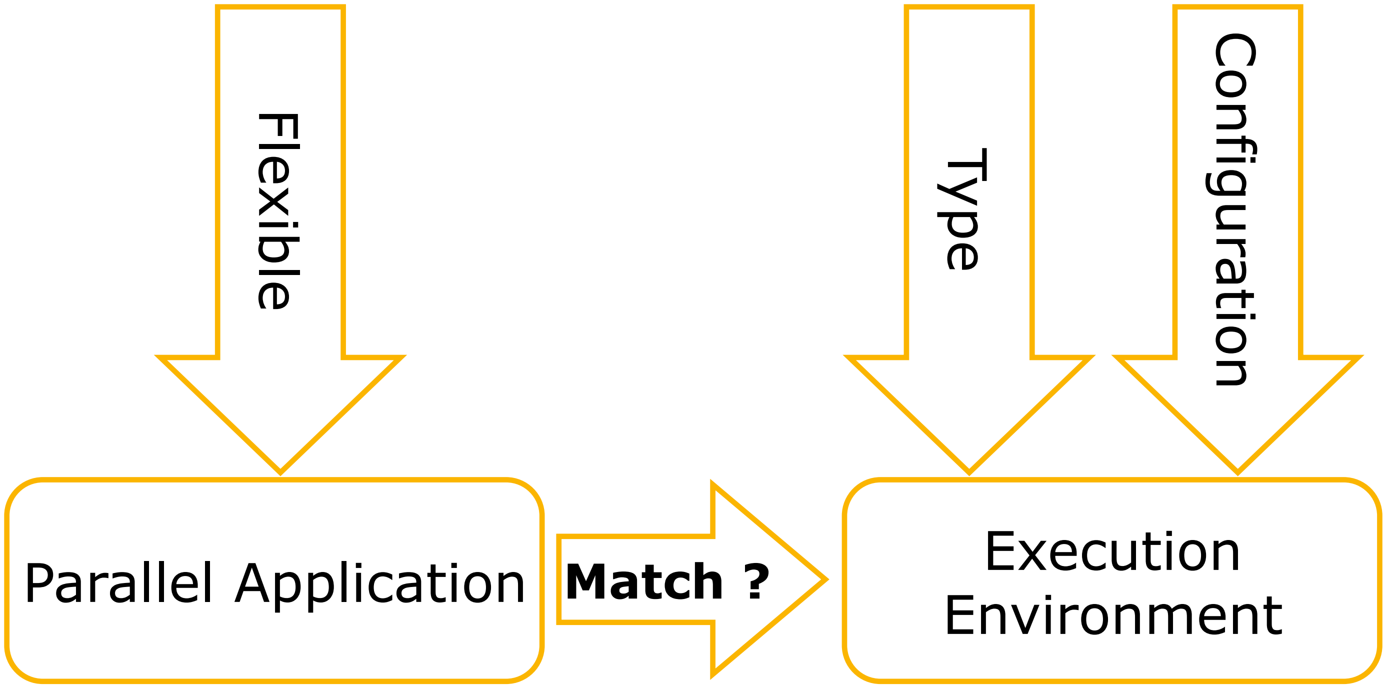
Multiple Instruction, Multiple Data (**MIMD**)

Execution Environment Mapping

	Data Parallel → SIMD	Task Parallel → MIMD
Shared Memory (SM)	SM-SIMD systems: GPU, Cell, SSE, AltiVec Vector processor ...	SM-MIMD systems: ManyCore/SMP system ...
Shared Nothing / Distributed Memory (DM)	DM-SIMD systems: processor-array systems systolic arrays Hadoop ...	DM-MIMD systems: cluster systems MPP systems ...

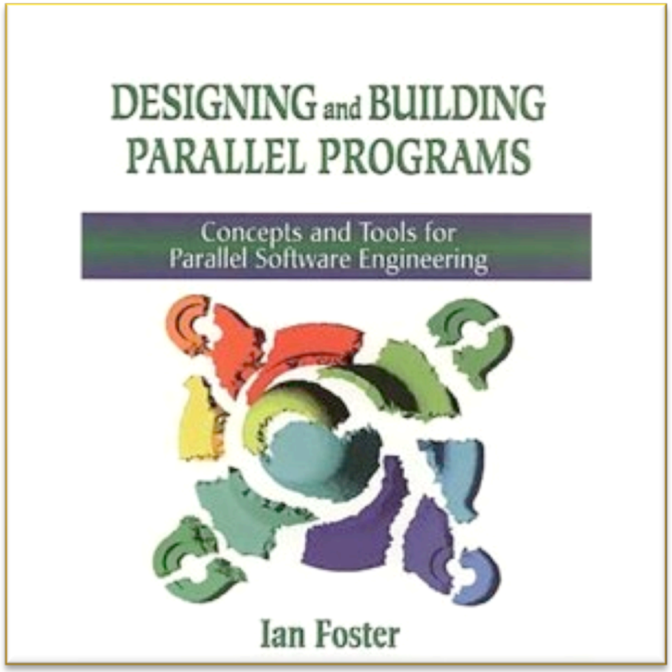
- Task parallel workload is a MIMD problem
- Data parallel workload is a SIMD problem
- Execution environments are optimized for one kind of workload, event though they can also the other one

The Parallel Programming Problem



Designing Parallel Algorithms [Foster]

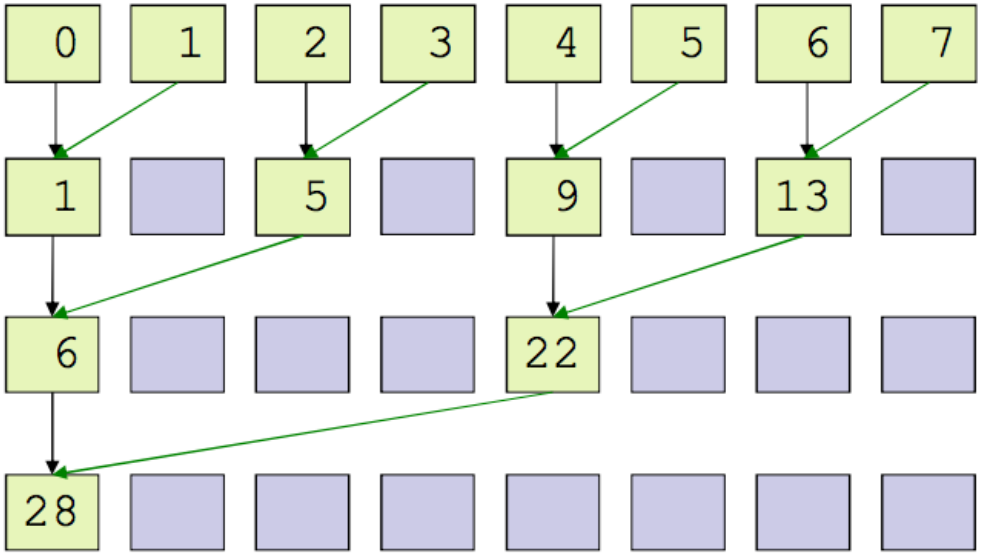
- Map workload problem on an execution environment
  - Concurrency for speedup
  - Data locality for speedup
  - Scalability
- Best parallel solution typically differs massively from the sequential version of an algorithm
- Foster defines four distinct stages of a methodological approach
- We will use this as a guide in the upcoming discussions
- Note: Foster talks about communication, we use the term synchronization instead
- Example: Parallel Sum



Example: Parallel Reduction

- Reduce a set of elements into one, given an operation

- Example: Sum



Example: All Prefix Sums

- Input: Ordered set $[a_0, a_1, \ldots, a_n]$
- Output: Ordered set $[a_0, (a_0 + a_1), \ldots, (a_0 + a_1 + \ldots + a_n)]$
- "+" is an arbitrary operation
- Multiple use cases
  - Lexically comparison of strings
  - Add multi-precision numbers that do not fit into one word
  - Implementation of quick sort
  - Delete marked elements from an array
  - Search for regular expressions
- Serial version is trivial, and demands O(n) steps

Designing Parallel Algorithms [Foster]

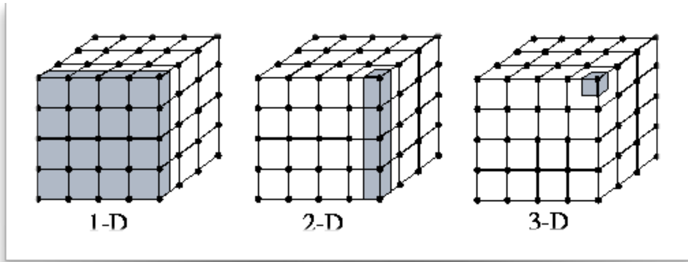
- A) Search for concurrency and scalability
  - **Partitioning** –
    Decompose computation and data into small tasks
  - **Communication** –
    Define necessary coordination of task execution
- B) Search for locality and other performance-related issues
  - **Agglomeration** –
    Consider performance and implementation costs
  - **Mapping** –
    Maximize processor utilization, minimize communication
- Might require backtracking or parallel investigation of steps

Partitioning

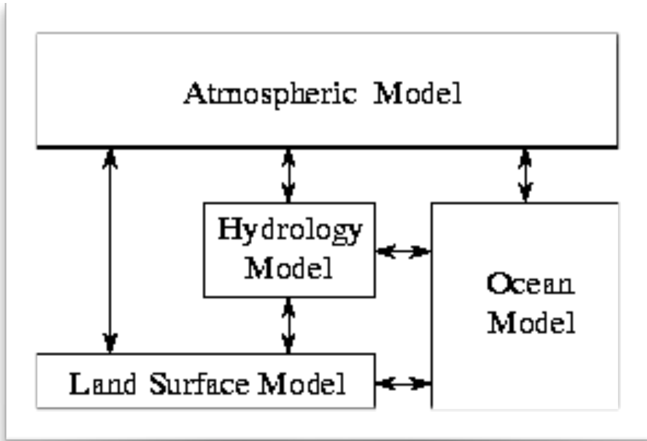
- Expose opportunities for parallel execution – fine-grained decomposition
- Good partition keeps computation and data together
  - **Data partitioning** leads to data parallelism
  - **Computation partitioning** leads task parallelism
  - Complementary approaches, can lead to different algorithms
  - Reveal hidden structures of the algorithm that have potential
  - Investigate complementary views on the problem
- Avoid replication of either computation or data, can be revised later to reduce communication overhead
- Step results in multiple candidate solutions

Partitioning - Decomposition Types

- **Domain Decomposition**
  - Define small data fragments
  - Specify computation for them
  - Different phases of computation on the same data are handled separately
  - Rule of thumb: First focus on large or frequently used data structures



- **Functional Decomposition**
  - Split up computation into disjoint tasks, ignore the data accessed for the moment
  - With significant data overlap, domain decomposition is more appropriate



Partitioning - Checklist

- Checklist for resulting partitioning scheme
  - Order of magnitude more tasks than processors ?
    -> Keeps flexibility for next steps
  - Avoidance of redundant computation and storage requirements ?
    -> Scalability for large problem sizes
  - Tasks of comparable size ?
    -> Goal to allocate equal work to processors
  - Does number of tasks scale with the problem size ?
    -> Algorithm should be able to solve larger tasks with more processors
- Resolve bad partitioning by estimating performance behavior, and eventually reformulating the problem

Communication Step

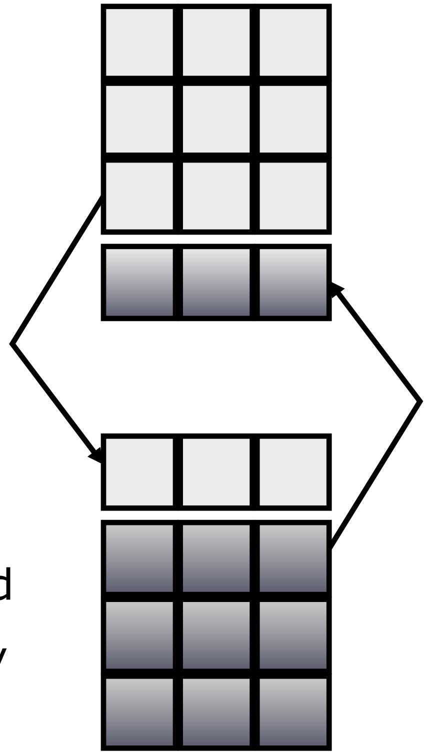
- Specify links between data consumers and data producers
- Specify kind and number of messages on these links
- Domain decomposition problems might have tricky communication infrastructures, due to data dependencies
- Communication in functional decomposition problems can easily be modeled from the data flow between the tasks
- Categorization of communication patterns
  - Local communication (few neighbors) vs. global communication
  - Structured communication (e.g. tree) vs. unstructured communication
  - Static vs. dynamic communication structure
  - Synchronous vs. asynchronous communication

Communication - Hints

- Distribute computation and communication, don‘t centralize algorithm
  - Bad example: Central manager for parallel summation
  - Divide-and-conquer helps as mental model to identify concurrency
- Unstructured communication is hard to agglomerate, better avoid it
- Checklist for communication design
  - Do all tasks perform the same amount of communication ? -> Distribute or replicate communication hot spots
  - Does each task performs only local communication ?
  - Can communication happen concurrently ?
  - Can computation happen concurrently ?

Ghost Cells

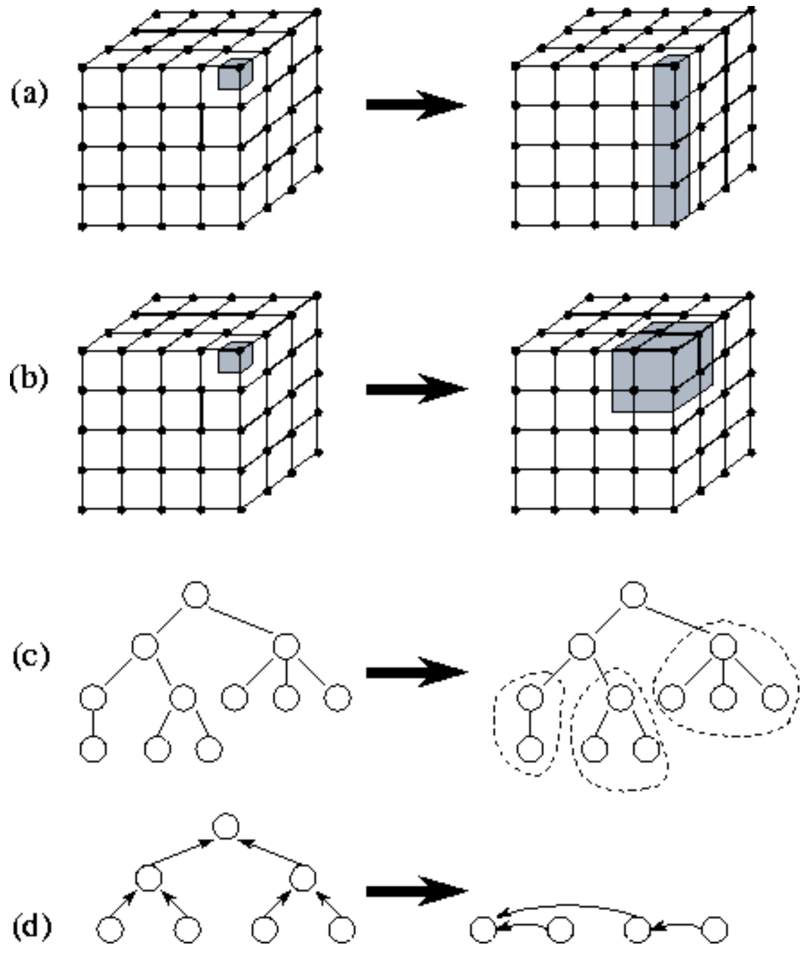
- Domain decomposition might lead to chunks that demand data from each other for their computation
  - Solution 1: Copy necessary portion of data (‚ghost cells‘)
    - Feasible if no synchronization is needed after update
    - Data amount and frequency of update influences resulting overhead and efficiency
    - Additional memory consumption
  - Solution 2: Access relevant data ‚remotely‘ as needed
    - Delays thread coordination until the data is really needed
    - Correctness („old“ data vs. „new“ data) must be considered on parallel progress



Agglomeration Step

- Algorithm so far is correct,
  but not specialized for some execution environment
- Check again partitioning and communication decisions
  - Agglomerate tasks for efficient execution on some machine
  - Replicate data and / or computation for efficiency reasons
- Resulting number of tasks can still be greater than the number of processors
- Three conflicting guiding decisions
  - Reduce communication costs by coarser granularity of computation and communication
  - Preserve flexibility with respect to later mapping decisions
  - Reduce software engineering costs (serial -> parallel version)

Agglomeration [Foster]



Agglomeration – Granularity vs. Flexibility

- Reduce communication costs by coarser granularity
  - Sending less data
  - Sending fewer messages (per-message initialization costs)
  - Agglomerate, especially if tasks cannot run concurrently
    - Reduces also task creation costs
  - Replicate computation to avoid communication (helps also with reliability)
- Preserve flexibility
  - Flexible large number of tasks still prerequisite for scalability
- Define granularity as compile-time or run-time parameter

Agglomeration - Checklist

- Communication costs reduced by increasing locality ?
- Does replicated computation outweighs its costs in all cases ?
- Does data replication restrict the range of problem sizes / processor counts ?
- Does the larger tasks still have similar computation / communication costs ?
- Does the larger tasks still act with sufficient concurrency ?
- Does the number of tasks still scale with the problem size ?
- How much can the task count decrease, without disturbing load balancing, scalability, or engineering costs ?
- Is the transition to parallel code worth the engineering costs ?

Mapping Step

- Only relevant for shared-nothing systems, since shared memory systems typically perform automatic task scheduling
- Minimize execution time by
  - Place concurrent tasks on different nodes
  - Place tasks with heavy communication on the same node
- Conflicting strategies, additionally restricted by resource limits
  - In general, NP-complete bin packing problem
- Set of sophisticated (dynamic) heuristics for load balancing
  - Preference for local algorithms that do not need global scheduling state

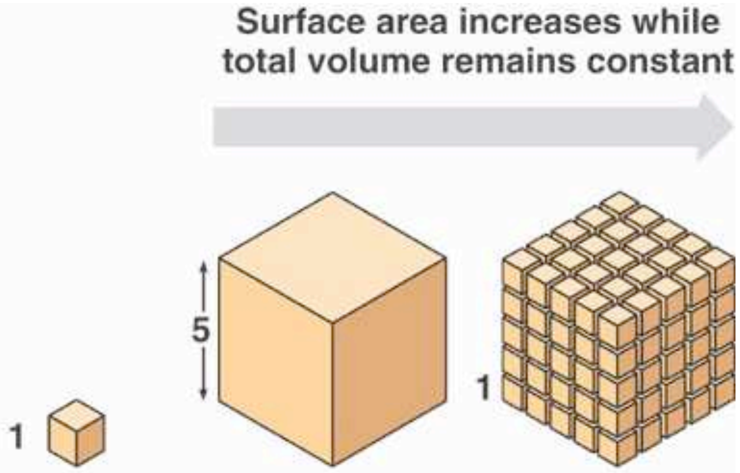
Partitioning Strategies [Breshears]

- Produce at least as many tasks as there will be threads / cores
  - But: Might be more effective to use only fraction of the cores (granularity)
  - Computation must pay-off with respect to overhead
- Avoid synchronization, since it adds up as overhead to serial execution time
- Patterns for data decomposition
  - By element (one-dimensional)
  - By row, by column group, by block (multi-dimensional)
  - Influenced by ratio of computation and synchronization

Surface-To-Volume Effect [Foster, Breshears]

- Visualize the data to be processed (in parallel) as sliced 3D cube
- **Synchronization** requirements of a task
  - Proportional to the **surface** of the data slice it operates upon
  - Visualized by the amount of ‚borders' of the slice
- **Computation** work of a task
  - Proportional to the **volume** of the data slice it operates upon
  - Represents the granularity of decomposition
- **Ratio of synchronization and computation**
  - High synchronization, low computation, high ratio → bad
  - Low synchronization, high computation, low ratio → good
  - Ratio decreases for increasing data size per task
- Coarse granularity by agglomerating tasks in all dimensions
  - For given volume, the surface then goes down → good

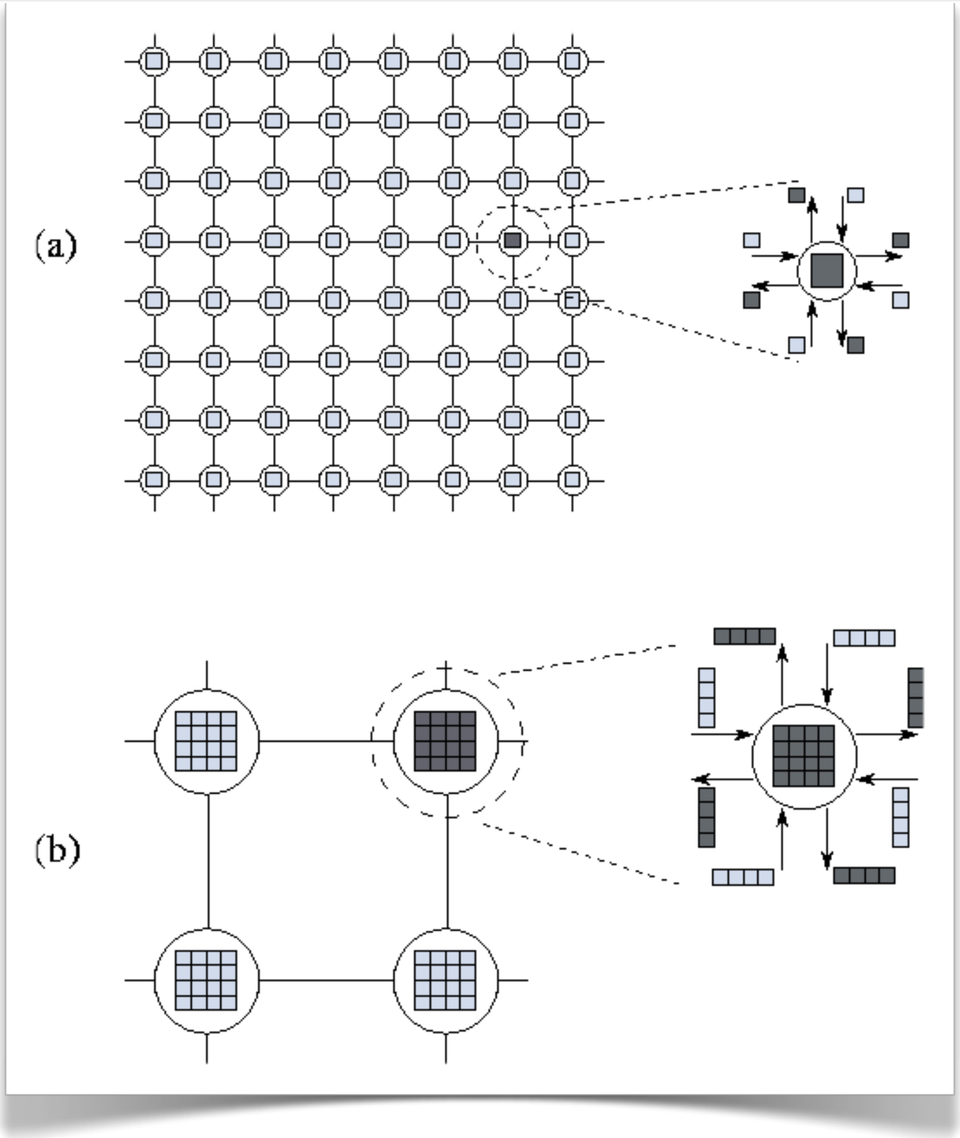
Surface-To-Volume Effect [Foster, Breshears]



Total surface area (height × width × number of sides × number of boxes)	6	150	750
Total volume (height × width × length × number of boxes)	1	125	125
Surface-to-volume ratio (surface area / volume)	6	1.2	6

Surface-to-Volume Effect [Foster]

- Computation on 8x8 grid
- (a): 64 tasks, one point each
  - 64x4=256 synchronizations
  - 256 data values are transferred
- (b): 4 tasks, 16 points each
  - 4x4=16 synchronizations
  - 16x4=64 data values are transferred



Designing Parallel Algorithms [Breshears]

- Parallel solution must keep *sequential consistency* property
- „Mentally simulate“ the execution of parallel streams
  - Check critical parts of the parallelized sequential application
- Amount of computation per parallel task
  - Always introduced by moving from serial to parallel code
  - Speedup must offset the parallelization overhead (Amdahl)
  - *Granularity*: Amount of parallel computation done before synchronization is needed
- **Fine-grained granularity** overhead vs. **coarse-grained granularity** concurrency
  - Iterative approach of finding the right granularity
  - Decision might be only correct only for the execution host under test