PMPP 2015/16

Design Patterns

TECHNISCHE UNIVERSITÄT DARMSTADT

(Preliminary) Course Schedule

TECHNISCHE UNIVERSITÄT DARMSTADT

12.10.2015	Introduction to PMPP
13.10.2015	Lecture CUDA Programming 1
19.10.2015	Lecture CUDA Programming 2
20.10.2015	Lecture CUDA Programming 3
26.10.2015	Lecture Parallel Basics, Exercise 1 assigned
27.10.2015	Questions and Answers (Q&A), S3|19, Room 2.8
2.11.2015	Intro Final Proj., Ex. 1 due, Ex. 2 assigned, Lecture PRAM
3.11.2015	Lecture PRAM (2)
9.11.2015	Final Projects assigned, L. Parallel Sort., Exercise 2 due
10.11.2015	Questions and Answers (Q&A)
16.11.2015	Questions and Answers (Q&A)
17.11.2015	Questions and Answers (Q&A)
23.11.2015	1st Status Presentation Final Projects
24.11.2015	1st Status Presentation Final Projects (continued)
30.11.2015	Lecture Design Patterns
1.12.2015	Questions and Answers (Q&A)

you are here →

GCC Graphics, Capture and Massively Parallel Computing

(Preliminary) Course Schedule

TECHNISCHE UNIVERSITÄT DARMSTADT

7.12.2015	Lecture
8.12.2015	Questions and Answers (Q&A)
14.12.2015	Lecture
15.12.2015	Questions and Answers (Q&A)
11.1.2016	2nd Status Presentation Final Projects
12.1.2016	2nd Status Presentation Final Projects (continued)
18.1.2016	
19.1.2016	
25.1.2016	
26.1.2016	
1.2.2016	
2.2.2016	
8.2.2016	Final Presentation Final Projects
9.2.2016	Final Presentation Final Projects (continued)

Final Projects - Feedback

TECHNISCHE UNIVERSITÄT DARMSTADT

- first presentation on 23.11.2014 and 24.11.2014
  - present the topic and the approach you plan to use to the class
  - overview over the problem
  - proposed solution using CUDA
  - everybody in class should understand the problem and the proposed solution!
  - submit slides ahead of time for faster change between presenters

- meet with your advisor early enough
- total time 20 minutes per topic including questions (for all teams together)
- everybody should give a portion of the presentation
- mandatory (talk to us if this is a problem)

Written Exam

TECHNISCHE UNIVERSITÄT DARMSTADT

- Date for the exam will be 02.03.2016, starting at 2pm
- Rooms: S101/A1 and S101/A01
- The exam will be in English only
- Important: Let us know today via email to
  pmpp2015@gris.informatik.tu-darmstadt.de
  if you also plan to take the exam “Communication Networks 2” which overlaps with this exam

“If you build it, they will come.”

TECHNISCHE UNIVERSITÄT DARMSTADT

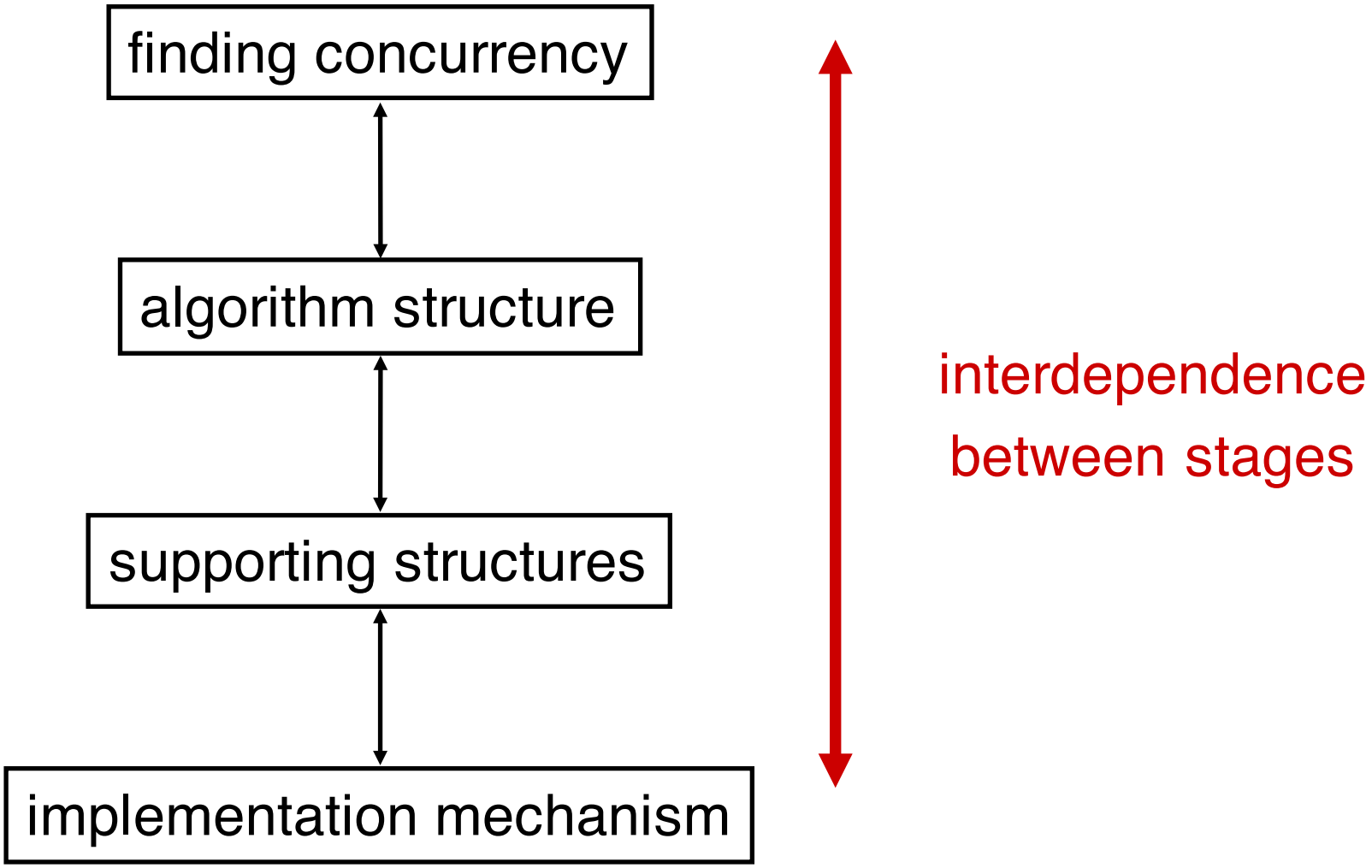
“And so we built them. Multiprocessor workstations, massively parallel supercomputers, a cluster in every department … and they haven’t come. Programmers haven’t come to program these wonderful machines.

[…]

The computer industry is ready to flood the market with hardware that will only run at full speed with parallel programs. But who will write these programs?“

Mattson, Sanders, Massingill.
Patterns for Parallel Programming (2005)

Design Spaces



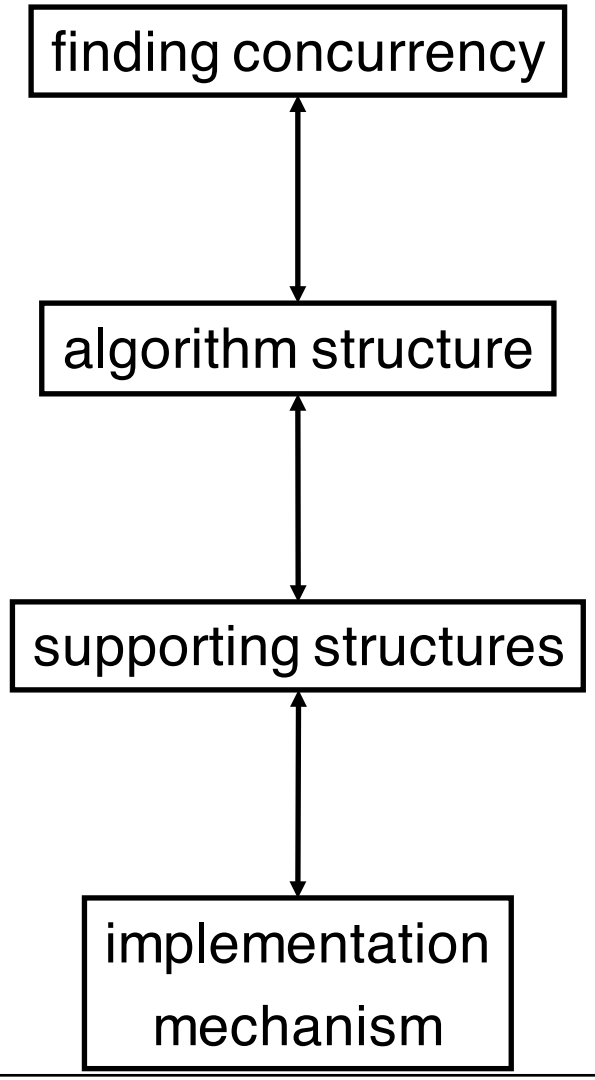
from Patterns for Parallel Programming by Mattson, Sanders, and Massingill

Design Patterns

TECHNISCHE UNIVERSITÄT DARMSTADT

- design pattern (from Wikipedia):

  In software engineering, a design pattern is a general reusable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.



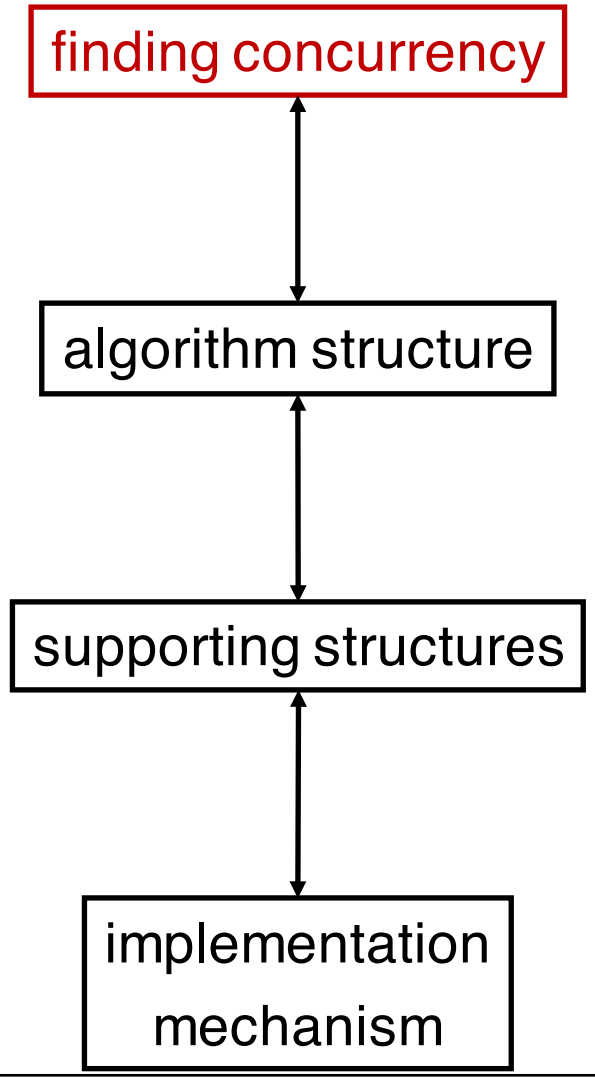
GCC Graphics, Capture and Massively Parallel Computing

Design Spaces and Design Patterns

TECHNISCHE UNIVERSITÄT DARMSTADT

- finding concurrency
  - programmer working in the problem domain to identify available concurrency and expose it in the algorithm design

➔task decomposition pattern
➔data decomposition pattern
➔group tasks pattern
➔order tasks pattern
➔data sharing pattern
➔design evaluation pattern



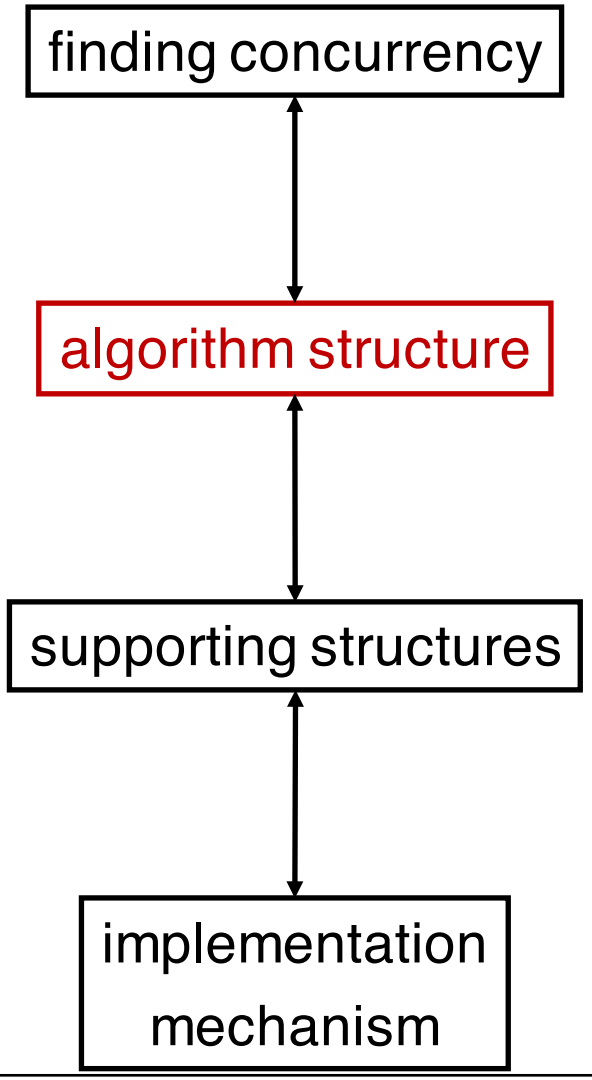
GCC
Graphics, Capture and Massively Parallel Computing

Design Spaces and Design Patterns

TECHNISCHE UNIVERSITÄT DARMSTADT

- algorithm structure
  - programmer working with high-level structures for organizing a parallel algorithm

➔task parallelism pattern
➔divide and conquer pattern
➔geometric decomposition pattern
➔recursive data pattern
➔pipeline pattern
➔event-based coordination pattern



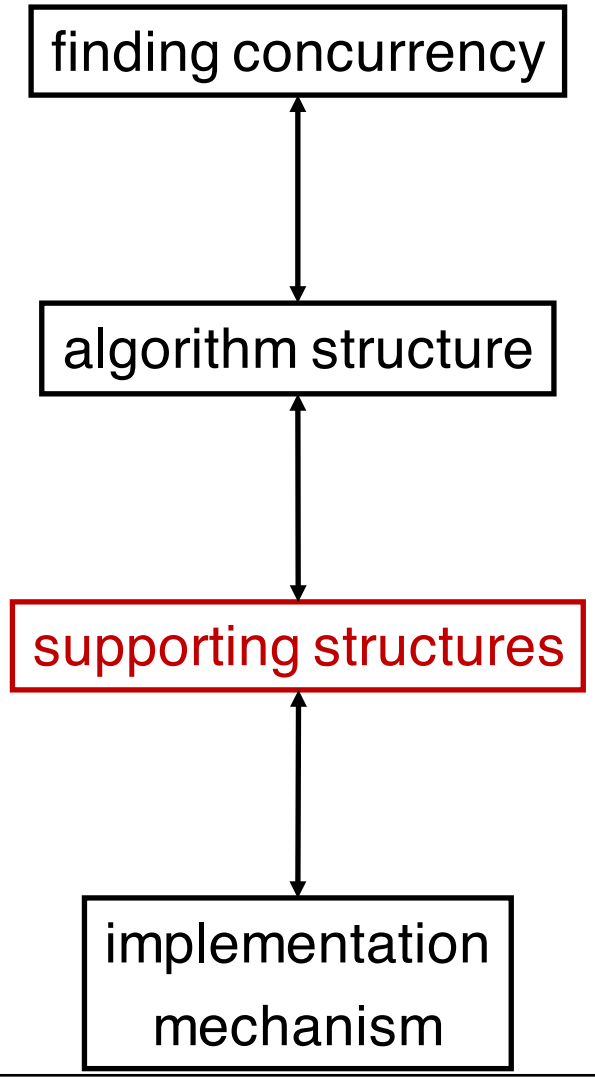
GCC Graphics, Capture and Massively Parallel Computing

Design Spaces and Design Patterns

TECHNISCHE UNIVERSITÄT DARMSTADT

- supporting structures
  - shift from algorithms to source code
  - organization of parallel program
  - techniques to manage shared data

➔SPMD pattern
➔master/worker pattern
➔loop parallelism pattern
➔fork/join pattern
➔shared data pattern
➔shared queue pattern
➔distributed array pattern
➔…



GCC Graphics, Capture and Massively Parallel Computing

Design Spaces (and Design Patterns)

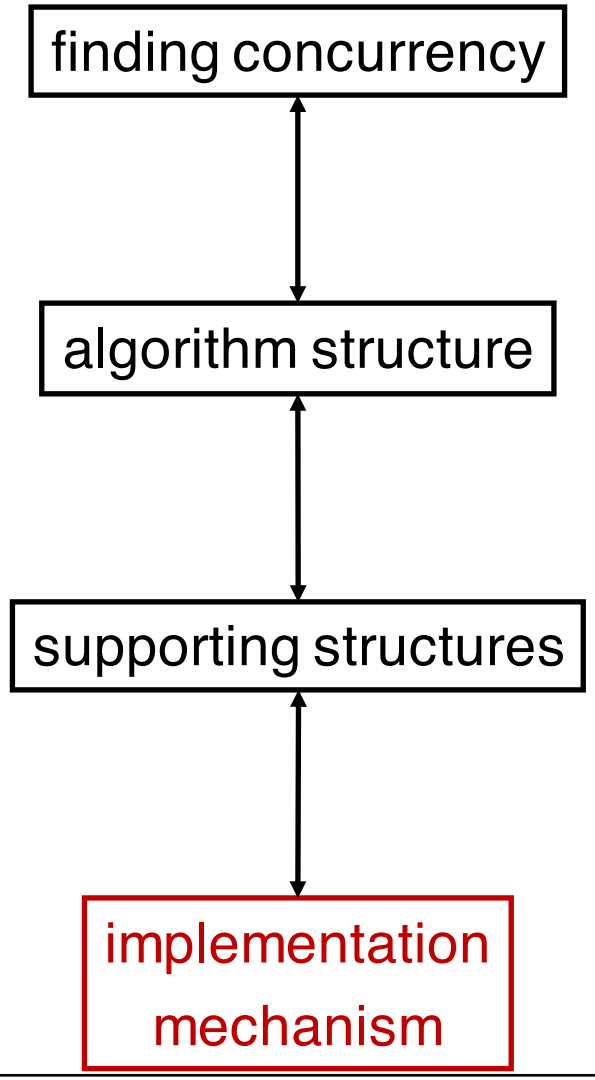
TECHNISCHE UNIVERSITÄT DARMSTADT

- implementation mechanisms
  - specific software constructs for implementing a parallel program

➔UE management

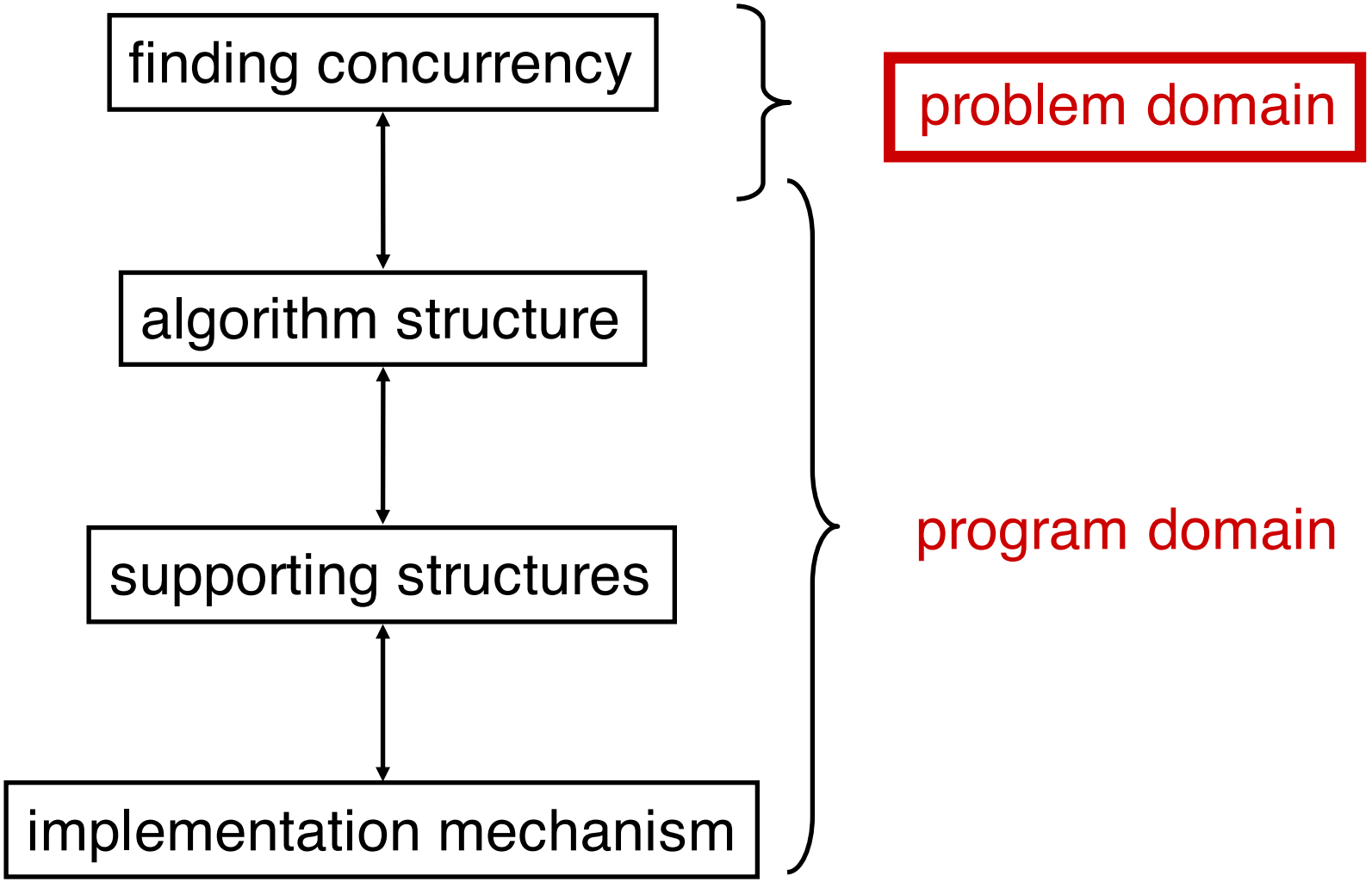
➔synchronization

➔communication



GCC Graphics, Capture and Massively Parallel Computing

Design Spaces



Three Questions

TECHNISCHE UNIVERSITÄT DARMSTADT

- Is the problem large enough and the results significant enough to justify expending effort to solve it faster?
- Are the key features and data elements well understood?
- What is the most computationally intensive part of the problem on which efforts should be focused?

Quantifying Parallel Computation

TECHNISCHE UNIVERSITÄT DARMSTADT

- serial fraction γ
  - relative running time of serial terms that cannot be parallelized

$$\gamma = \frac{T_{setup} + T_{finalization}}{T_{total}(1)}$$

  - fraction of time spent in parallelizable part is (1 - γ)
  - rewriting total computation time using P PEs

$$T_{total}(P) = \gamma \cdot T_{total}(1) + \frac{(1-\gamma) \cdot T_{total}(1)}{P}$$

Quantifying Parallel Computation

TECHNISCHE UNIVERSITÄT DARMSTADT

- Amdahl's law
  - relative speedup of an ideal parallel algorithm, no overhead in parallel part

$$S(P) = \frac{T_{total}(1)}{\left(\gamma + \frac{1-\gamma}{P}\right) \cdot T_{total}(1)} = \frac{1}{\gamma + \frac{1-\gamma}{P}}$$

  - limit for $P \rightarrow$ infinity

$$S = \frac{1}{\gamma}$$

  - upper bound for speedup obtainable if serial part represents γ of the total computation

Quantifying Parallel Computation

TECHNISCHE UNIVERSITÄT DARMSTADT

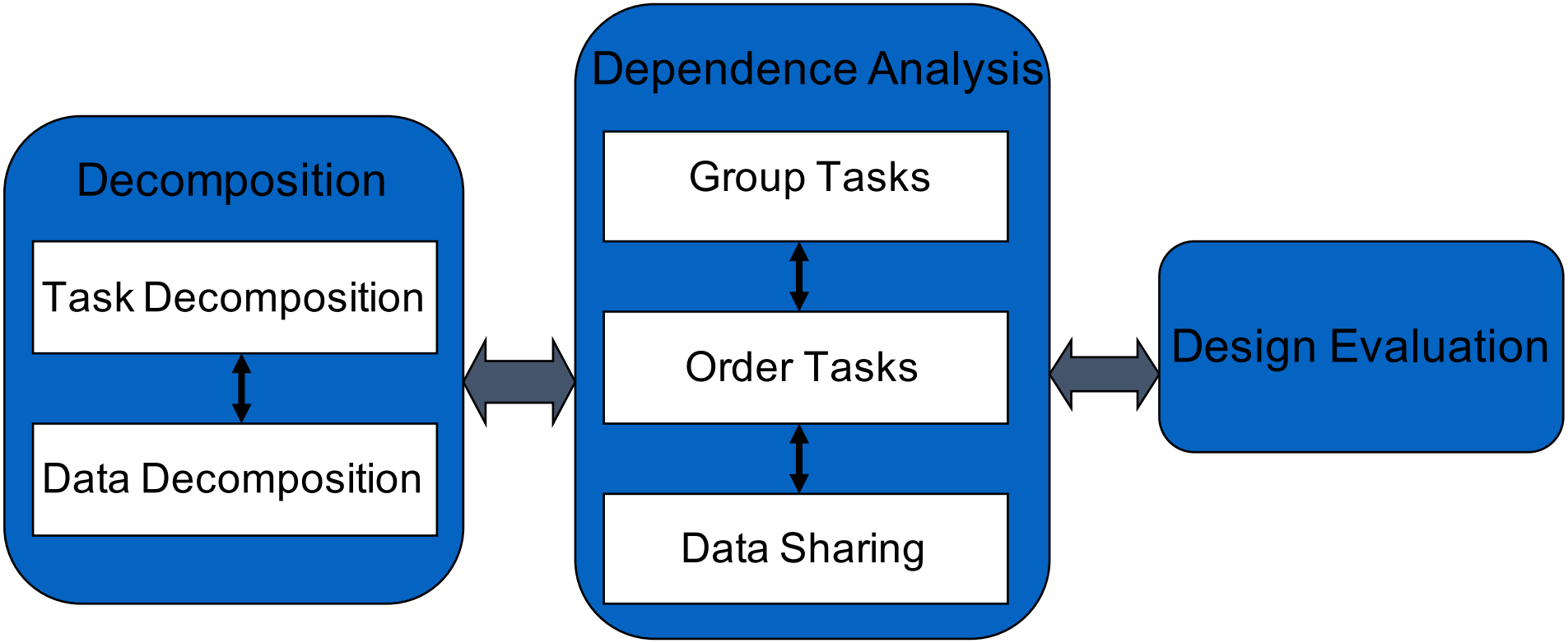
- Amdahl's law in practice
  - how much effort should be spent on parallel implementation if even 10% of the algorithm is serial?
- CUDA: initial setup time for a kernel
- CUDA: transfer time host ↔ device can be a serious bottleneck
- Amdahl's law may be too pessimistic
  - what resources limit computation (e.g., memory instead of CPU cycles)
  - assumes that parallel and serial computation perform identically
  - looks at fixed-size speedup, but also possible to solve larger problems

Finding Concurrency

TECHNISCHE UNIVERSITÄT DARMSTADT

- identify a decomposition of the problem into sub-problems that can be solved simultaneously
  - a task decomposition that identifies tasks that can executed concurrently
  - a data decomposition that identifies data local to each task
  - a way of grouping tasks and ordering the groups to satisfy temporal constraints
  - an analysis on the data sharing patterns among the concurrent tasks
  - a design evaluation that assesses of the quality the choices made in all the steps

Finding Concurrency



- typically an iterative process
- opportunities exist for dependence analysis to play earlier role in decomposition

Task Decomposition

TECHNISCHE UNIVERSITÄT DARMSTADT

- “In an ideal world, the compiler would find tasks for the programmer. Unfortunately, this almost never happens.”

  Mattson, Sanders, Massingill

- many large problems have natural independent tasks
  - distinct calls to a function (functional decomposition)
  - distinct iterations of a loop (loop-splitting algorithms)
  - tasks generated by data-driven decompositions, e.g., splitting a large data structure in chunks that are processed independently

Task Decomposition

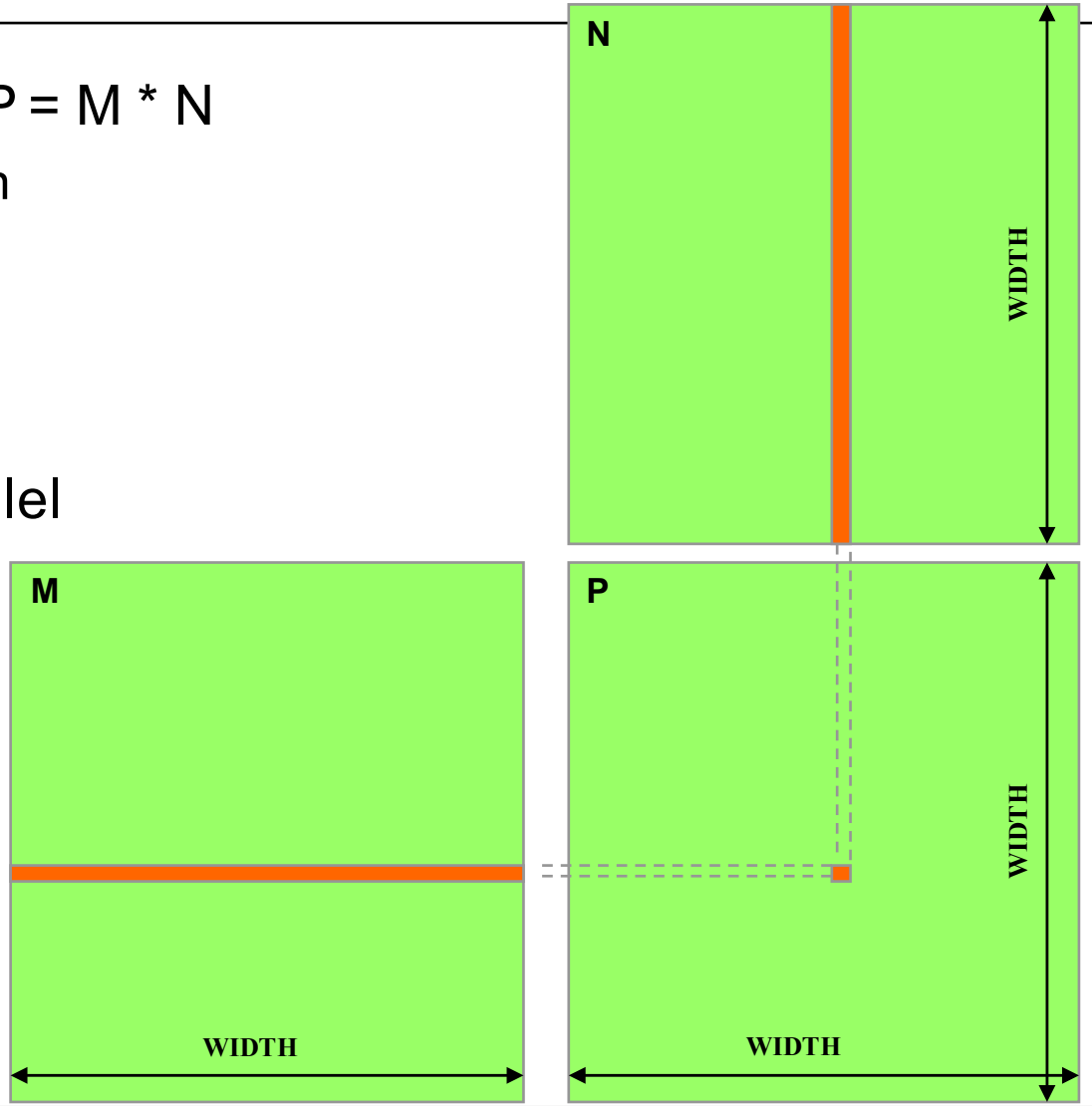
TECHNISCHE UNIVERSITÄT DARMSTADT

- main forces influencing the design
- flexibility
  - keep options to adapt design to different implementation requirements, e.g., hardware architecture, programming style
- efficiency
  - enough tasks to keep all PEs busy
  - enough work per task to compensate for overhead incurred to manage dependencies
  - efficiency can yield complex and inflexible decompositions
- simplicity
  - complex enough to solve the problem
  - simple enough to enable debugging, maintenance

Task Decomposition Examples

TECHNISCHE UNIVERSITÄT DARMSTADT

- square matrix multiplication P = M * N
  - all matrices have dimension WIDTH x WIDTH
- natural task (sub-problem) computes one element of P
- all tasks can execute in parallel

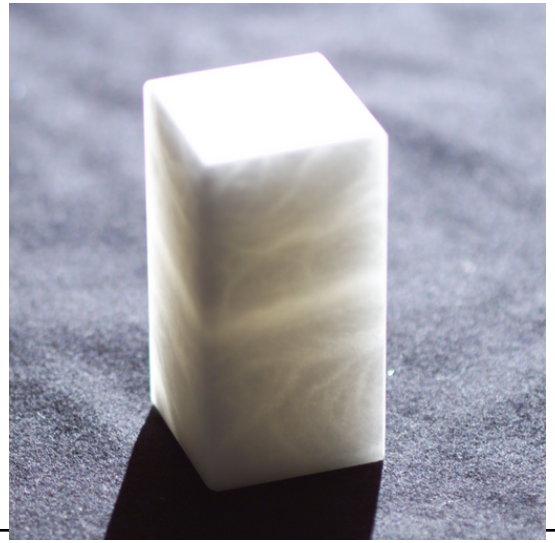


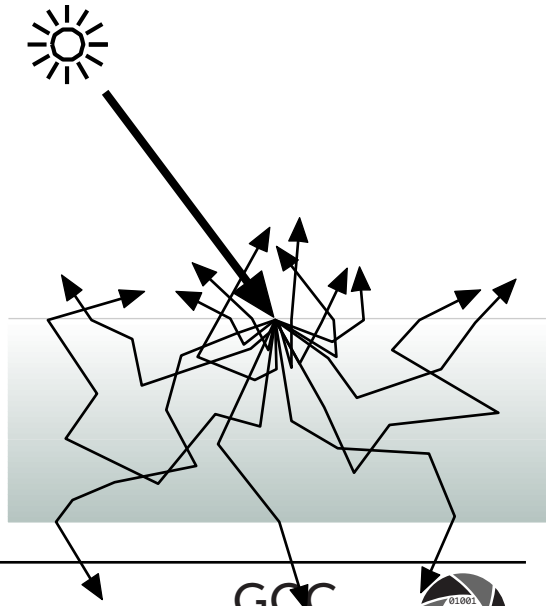
GCC Graphics, Capture and Massively Parallel Computing

Task Decomposition Examples

TECHNISCHE UNIVERSITÄT DARMSTADT

- ray tracing
  - follow a large number of independent rays traveling through a scene
  - interactions with surfaces, material modeling reflectance
  - e.g., in realistic rendering, medical imaging
- natural task follows a single ray through the scene
- all tasks can execute in parallel
  - no interdependence between scene points





GCC Graphics, Capture and Massively Parallel Computing

Task Decomposition Examples

TECHNISCHE UNIVERSITÄT DARMSTADT

- molecular dynamics
- simulation of motions of a large molecular system
  - vibrational forces
  - rotational forces
  - neighbors that must be considered in non-bonded forces
  - non-bonded forces
  - update position and velocity
  - misc physical properties based on motions
- a natural task is to perform calculations of individual atoms in parallel
  - some of these can go in parallel for an atom

➔ often multiple ways exist to decompose a given problem

Data Decomposition

TECHNISCHE UNIVERSITÄT DARMSTADT

- the most compute intensive parts of many large problems manipulate a large data structure

- similar operations are being applied to different parts of the data structure, in a mostly independent manner
  - this is what CUDA is optimized for

- typical data decomposition patterns
  - array-based computations
    updates to different sections of a large array
  - recursive data structures
    decomposing the data structure into smaller problems that are treated independently

Data Decomposition

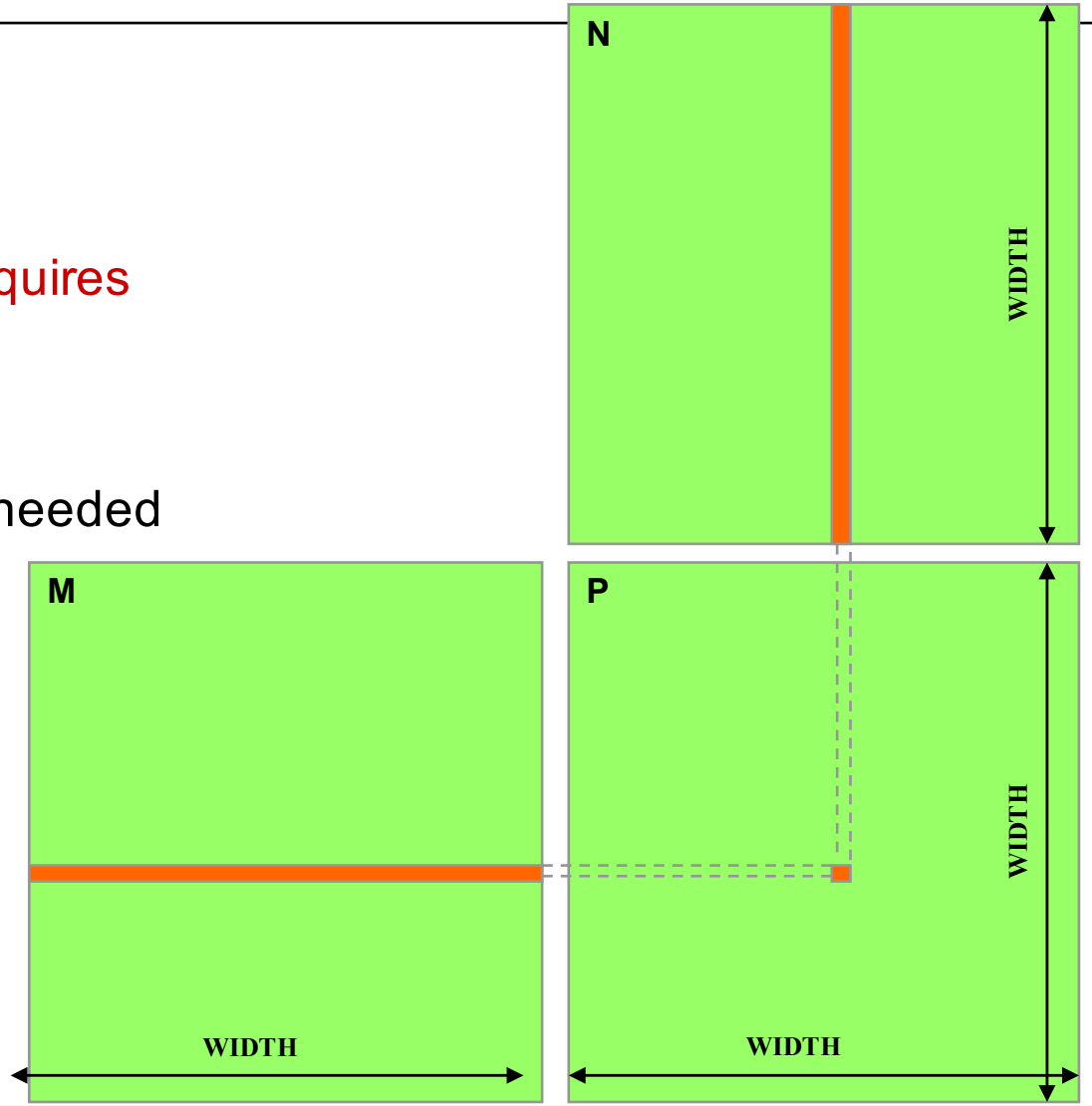
TECHNISCHE UNIVERSITÄT DARMSTADT

- flexibility
  - size and number of data chunks controlled by a small number of granularity knobs
  - trade-off between larger number of chunks and effort needed to manage dependencies
- efficiency
  - enough tasks to keep all PEs busy and offset the overhead of managing dependencies
  - enable good load balancing (especially of tasks take different amount of time)
- simplicity
  - complex enough to solve the problem
  - simple enough to enable debugging, maintenance

Data Decomposition Examples

TECHNISCHE UNIVERSITÄT DARMSTADT

- square matrix multiplication
- P row blocks (adjacent rows)
  - computing each partition requires access to entire N array
- square sub-blocks of P
  - only bands of M and N are needed

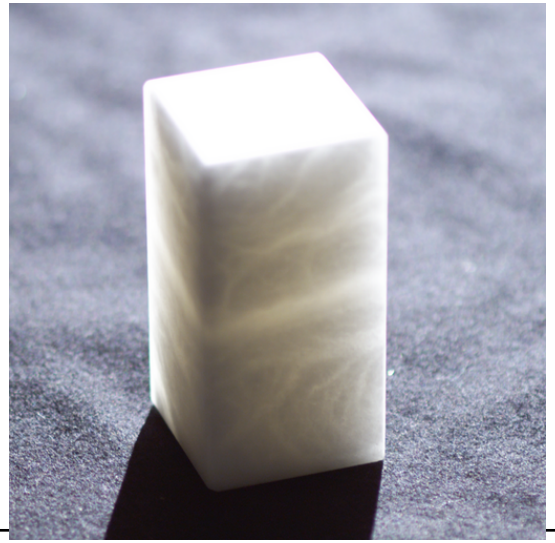


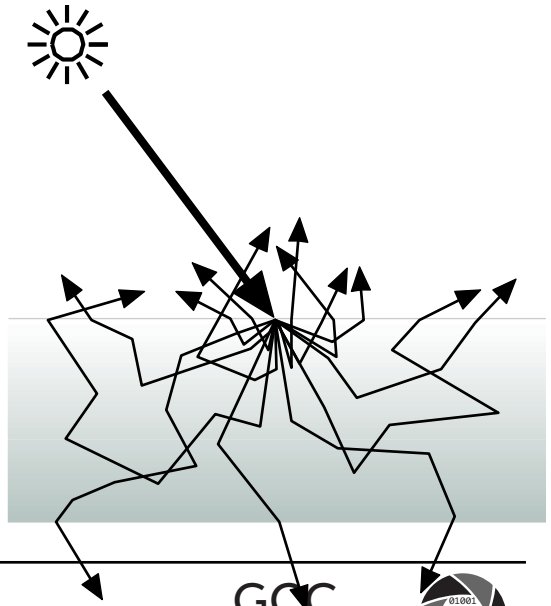
GCC
Graphics, Capture and Massively Parallel Computing

Data Decomposition Examples

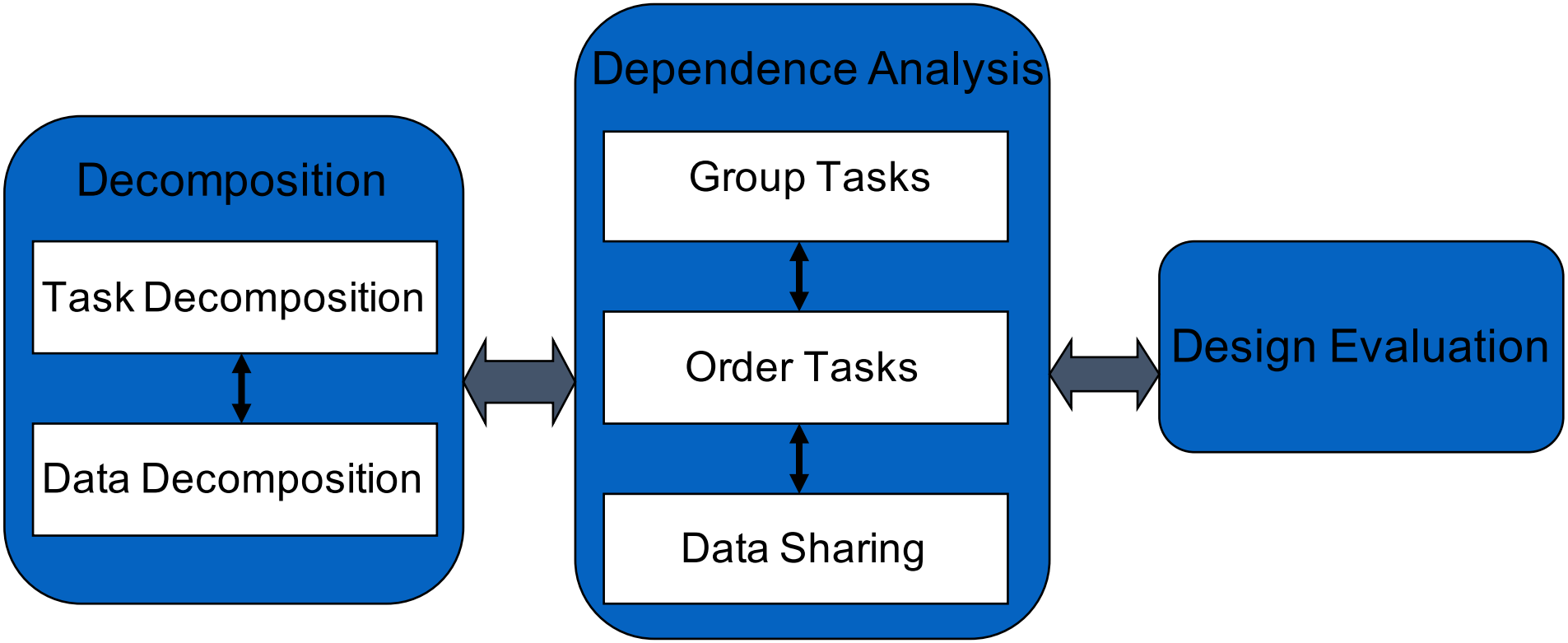
TECHNISCHE UNIVERSITÄT DARMSTADT

- ray tracing
- partition scene model into independent volumes
  - task corresponds to follow all rays inside this volume
  - need to transfer rays between PEs when crossing volume boundaries
- all tasks can execute in parallel





Finding Concurrency



- typically an iterative process
- opportunities exist for dependence analysis to play earlier role in decomposition

Task Grouping

TECHNISCHE UNIVERSITÄT DARMSTADT

- group tasks determined in the decomposition phase together to improve efficiency and simplify dependencies
  - determine structure of the tasks
- possible constraints between tasks
  - temporal dependency
    - task B must be completed before task A starts
    - data flow, possibly in a pipeline pattern
  - concurrent execution
    - e.g., iterative solution of a data-parallel problem
  - independent execution
    - order between tasks not relevant
    - can yield more concurrency

Task Grouping

TECHNISCHE UNIVERSITÄT DARMSTADT

- benefits
- reduced synchronization overhead
  - all tasks in the group can use a barrier to wait for a common dependence
- all tasks in the group efficiently share data loaded into a common on-chip, shared storage (Shared Memory)
- grouping and merging dependent tasks into one task reduces need for synchronization

Task Grouping

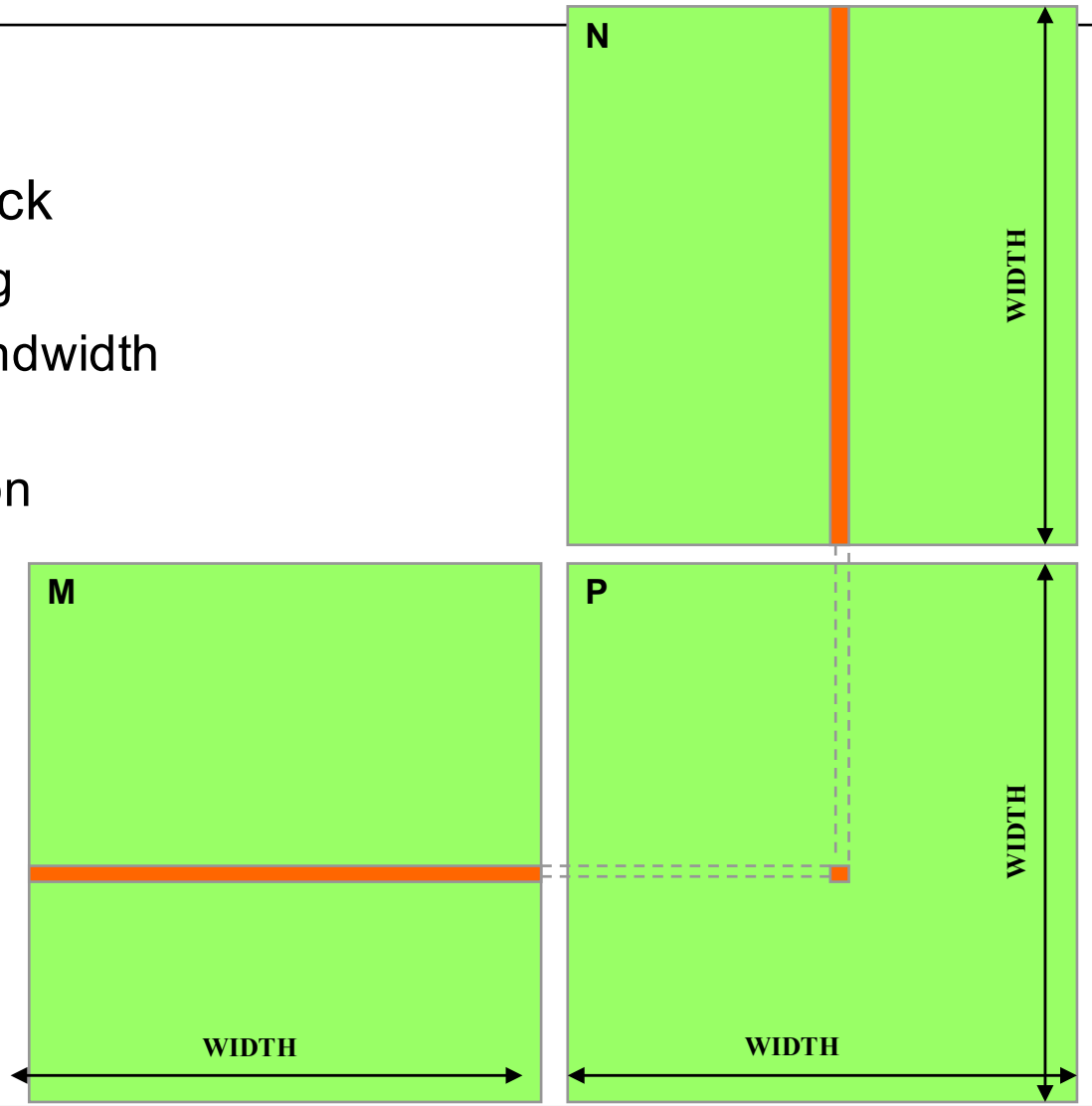
TECHNISCHE UNIVERSITÄT DARMSTADT

- possible approach
- group tasks according to high-level operation
  - e.g., iterations of a loop
  - keep tasks that share a constraint in distinct groups
  - ➔ constraint needs to be fulfilled by algorithm design
- merge tasks that share the same constraint
  - yields additional concurrency, flexibility in sheduling
- consider constraints between groups
  - e.g., temporal ordering between groups
  - merge independent groups sharing constraints

Task Grouping Example

TECHNISCHE UNIVERSITÄT DARMSTADT

- square matrix multiplication
- tasks calculating a P sub-block
  - extensive input data sharing
  - reduced global memory bandwidth using shared memory
  - all synchronized in execution



GCC
Graphics, Capture and Massively Parallel Computing

Task Ordering

TECHNISCHE UNIVERSITÄT DARMSTADT

- identify an ordering between tasks that satisfies all constraints among them
  - as restrictive as required
  - keep as much flexibility as possible
- temporal dependencies
  - directly implies an order of the tasks
- concurrent execution of tasks
  - mutual dependency on information created by the tasks
- total independence of tasks
  - not a constraint but important property
  - can be executed in any order

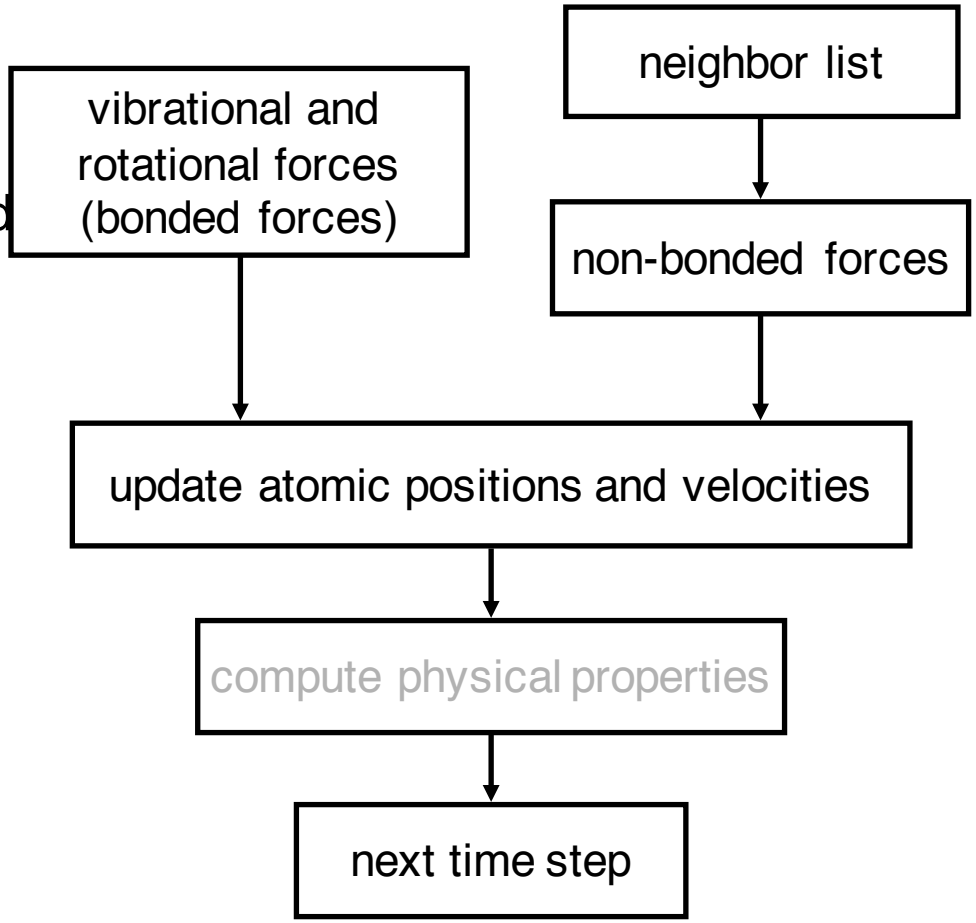
Task Ordering

TECHNISCHE UNIVERSITÄT DARMSTADT

- possible approach
- look at the data and the tasks producing/requiring it
  - defines a sequence of execution
- consider external services
  - e.g., data needs to be written to a file in a particular order
  - imposes also ordering constraint
- determine independent tasks

Task Ordering Example

- molecular dynamics
  - vibrational forces
  - rotational forces
  - neighbors that must be considered in non-bonded forces
  - non-bonded forces
  - update position and velocity
  - misc physical properties based on motions



Data Sharing

TECHNISCHE UNIVERSITÄT DARMSTADT

- how is data shared among the tasks?
  - collection of tasks that can run concurrently from task decomposition
  - corresponding data decomposition
  - manage dependency between tasks for safe concurrent execution
- shift focus from tasks to data
- task-local data can be updated independently
- shared data must be updated in a consistent way
- sometimes access to local data of a neighboring task is required

Data Sharing

TECHNISCHE UNIVERSITÄT DARMSTADT

- forces in data sharing
  - influence correctness and efficiency of the algorithm
- incorrect data sharing due to invalid data must be avoided
  - e.g., race conditions
- guaranteeing consistency in shared data might yield huge synchronization overhead
  - barrier sync via syncthreads
  - try to use local data if possible
- overhead due to communication caused by shared data
  - see e.g. atomic actions in CUDA

Data Sharing

TECHNISCHE UNIVERSITÄT DARMSTADT

- data sharing can be a double-edged sword
  - excessive data sharing can drastically reduce advantage of parallel execution
  - localized sharing can improve memory bandwidth efficiency
- efficient memory bandwidth usage can be achieved by synchronizing the execution of task groups and coordinating their usage of memory data
  - efficient use of on-chip, shared storage
- read-only sharing can usually be done at much higher efficiency than read-write sharing, which often requires synchronization

Data Sharing

TECHNISCHE UNIVERSITÄT DARMSTADT

- shared data categories
- read only
  - never modified, no protection required
  - sometimes replication of shared data advantageous
  - CUDA: constant or texture memory
- effectively local
  - partitioning into non-overlapping subsets (e.g. parts of an array)
  - access (read or write) by a single task
- read-write
  - general access by multiple tasks
  - requires protection with exclusive access mechanism

Data Sharing

TECHNISCHE UNIVERSITÄT DARMSTADT

- sub-categories of read-write
- accumulate
  - values from local copies of individual tasks are combined (e.g., using reduction with associative operation such as sum, minimum)
  - CUDA: could use atomic operations
- multiple read/single write
  - initial value read by many tasks
  - only one task writes back the result

Data Sharing Example

TECHNISCHE UNIVERSITÄT DARMSTADT

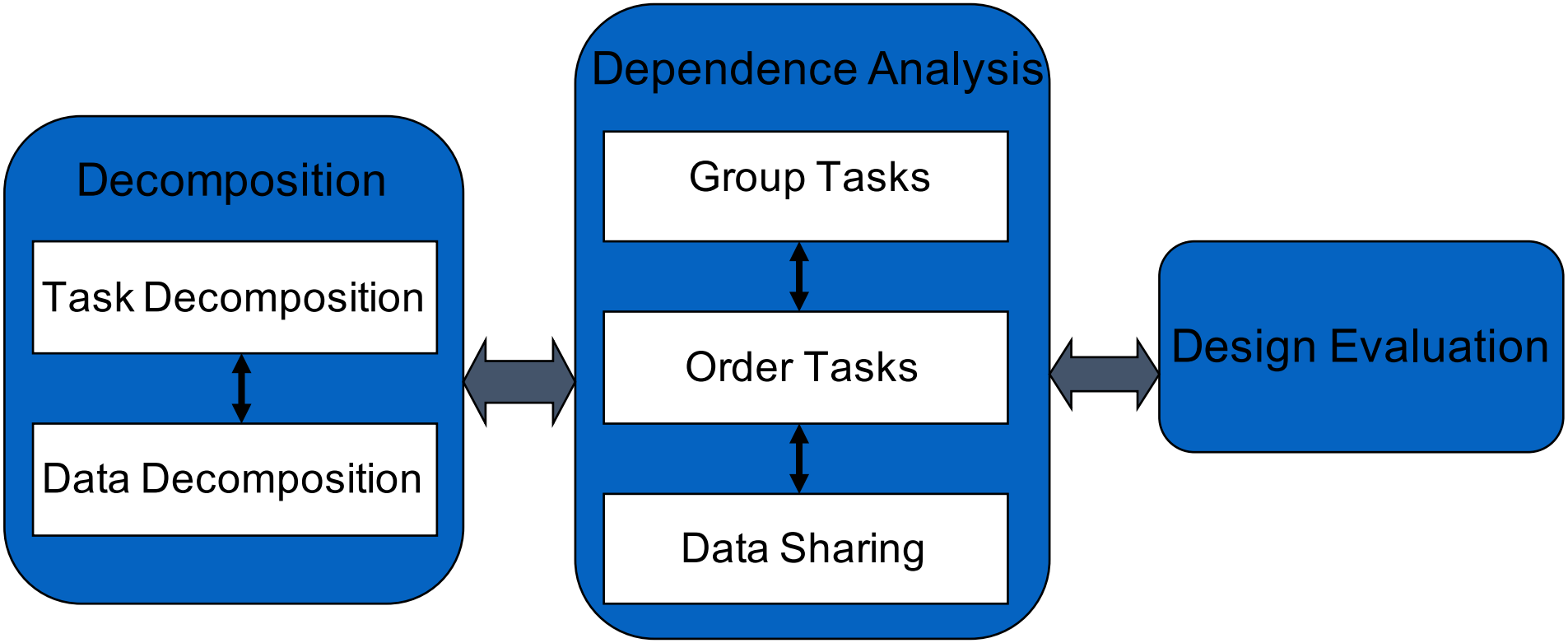
- matrix multiplication
- each task group will finish usage of each sub-block of N and M before moving on
  - N and M sub-blocks loaded into shared memory for use by all threads of a P sub-block
  - amount of on-chip shared memory strictly limits the number of threads working on a P sub-block
  - side note: registers can be pushed out to local memory
- read-only shared data can be more efficiently accessed as constant or texture data

Data Sharing Example

TECHNISCHE UNIVERSITÄT DARMSTADT

- molecular dynamics
- atomic coordinates
  - read-only access by the neighbor list, bonded force, and non-bonded force task groups
  - read-write access for the position update task group
- force array
  - read-only access by position update group
  - accumulate access by bonded and non-bonded task groups
- neighbor list
  - read-only access by non-bonded force task groups
  - generated by the neighbor list task group

Finding Concurrency



- typically an iterative process
- opportunities exist for dependence analysis to play earlier role in decomposition

Design Evaluation

- evaluate the result before moving on to the next design space
  - task decomposition
  - data decomposition
  - grouping and ordering of tasks
  - dependency analysis

- avoid continuing with unsuitable results
  - may need more iterations through design space

➔ typically multiple ways to decompose the problem

Design Evaluation

TECHNISCHE UNIVERSITÄT DARMSTADT

- forces
- suitability for the target platform
  - number of processors, shared data access, …
- design quality
  - simplicity
  - flexibility
  - efficiency
- preparation for the next phase of the design
  - regularity
  - dependencies

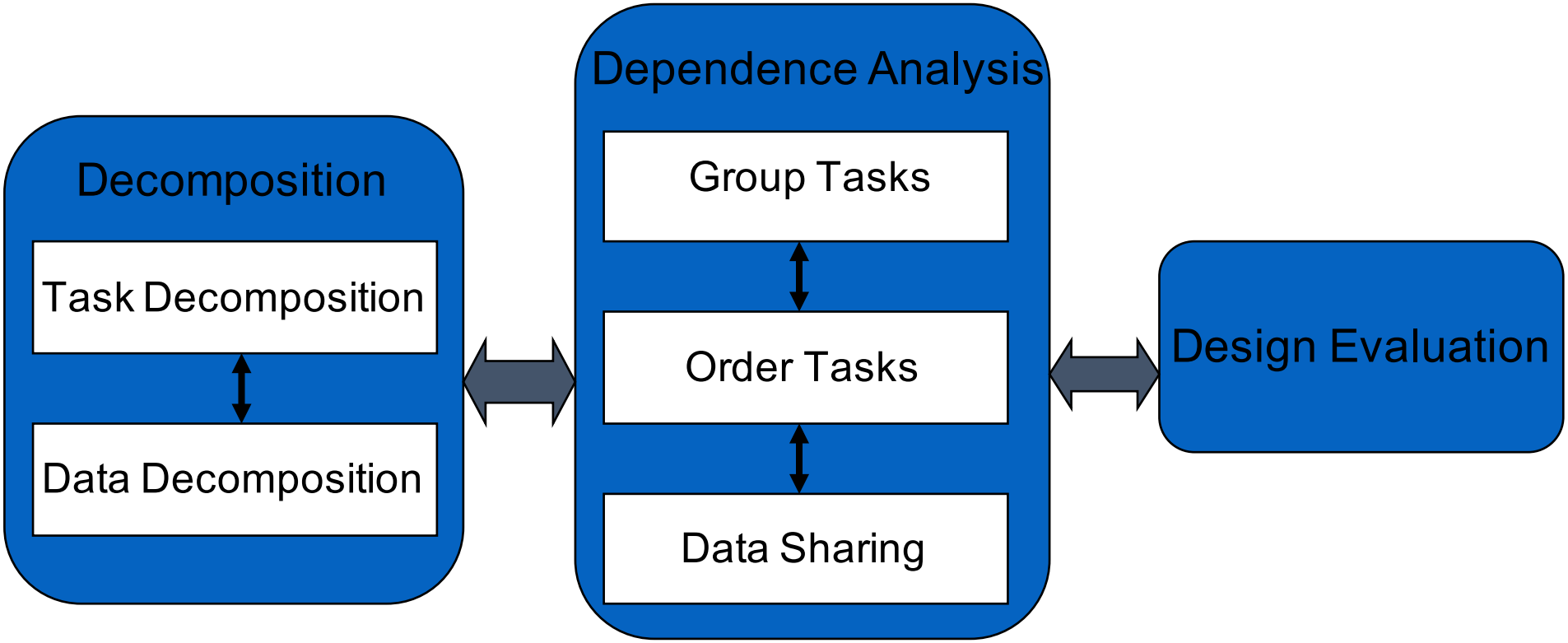
Design Evaluation

TECHNISCHE UNIVERSITÄT DARMSTADT

- key questions to ask about target platform
  - how many PEs are available?
  - how are data structures shared among PEs?
  - what does the target architecture imply about the number of UEs and how structures are shared among them?
  - will the time spent doing useful work in a task be significantly greater than the time taken to deal with dependencies?

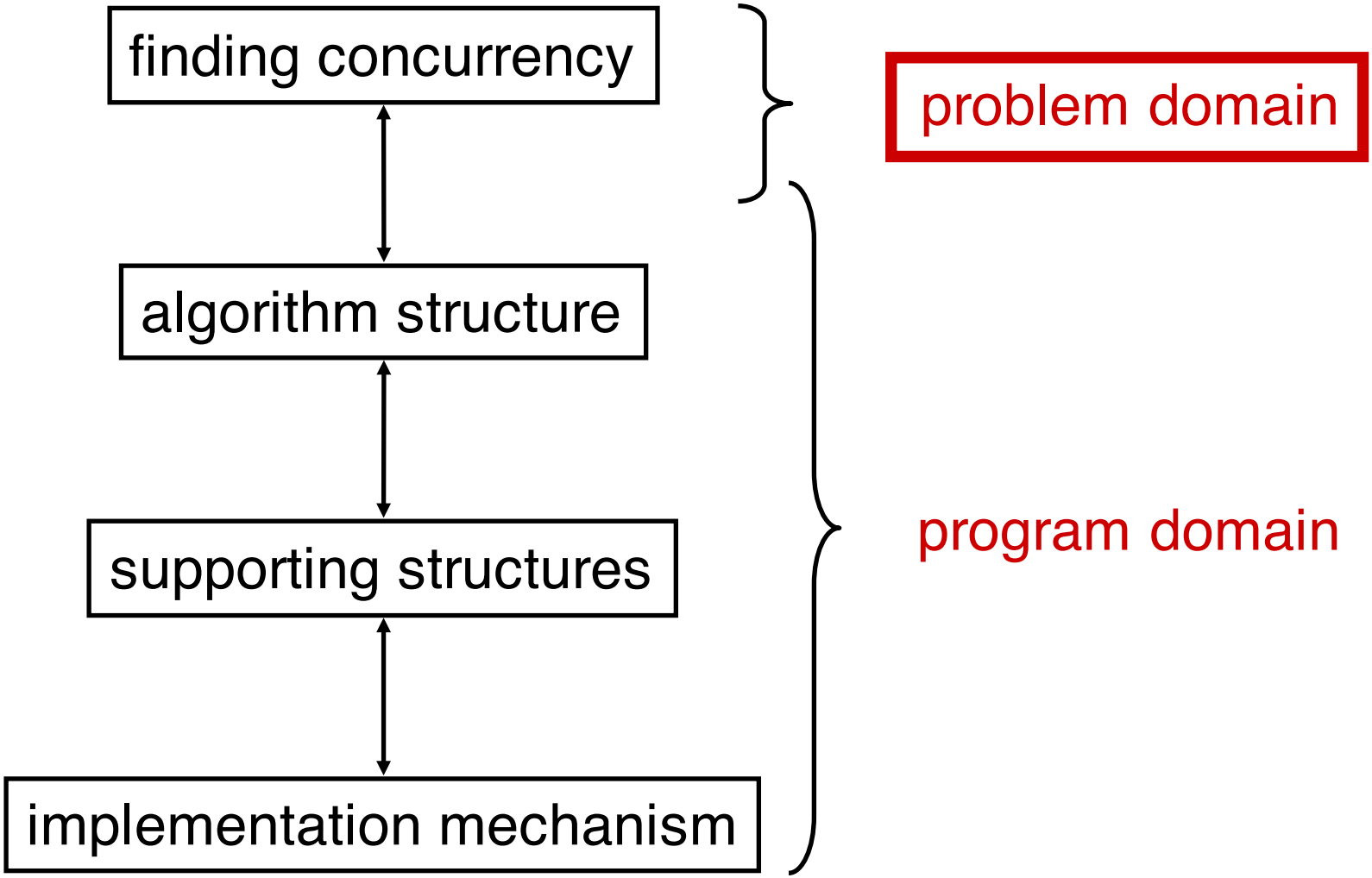
- for a CUDA example
  - how many threads can be supported?
  - how many threads are needed?
  - how are the data structures shared?
  - is there enough work in each thread between synchronizations to make parallel execution worthwhile?

Finding Concurrency



- typically an iterative process
- opportunities exist for dependence analysis to play earlier role in decomposition

Design Spaces



Recommended Reading

TECHNISCHE UNIVERSITÄT DARMSTADT

- Chapters 3, 4, 5, and 6 of
  Patterns for Parallel Programming
  by Mattson, Sanders, and Massingill
  - finding concurrency
  - algorithm structure
  - supporting structures
  - implementation mechanisms