Finding concurrency

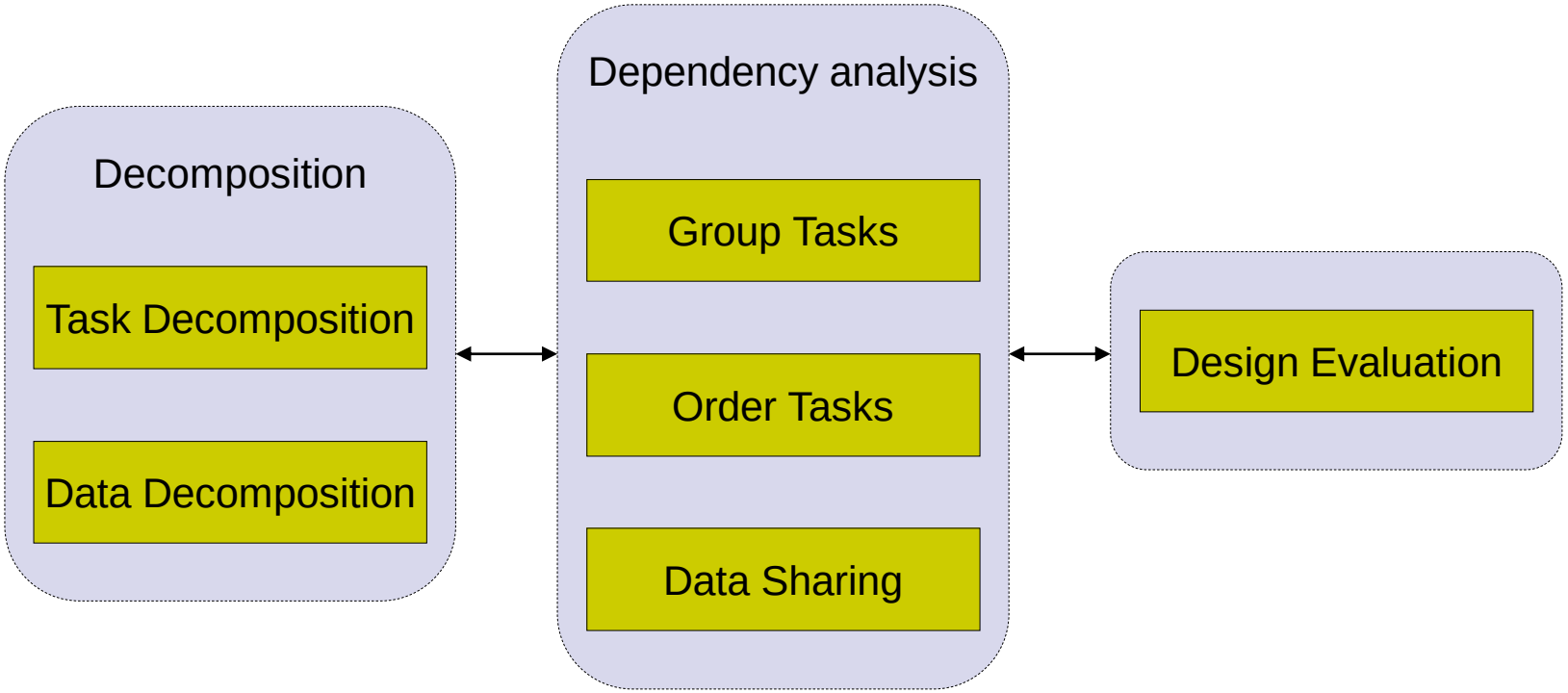
Jakub Yaghob

Finding concurrency design space

- Starting point for design of a parallel solution
- Analysis
- The patterns will help identify and analyze the exploitable concurrency in a problem
- After this is done, patterns from Algorithm structure space can be chosen to help design the appropriate algorithm structure to exploit identified concurrency

Overview

Finding concurrency



Overview

- Before starting to work:
  - Is the problem large enough?
  - Are the results significant enough?
- Understanding key features and data elements
- Understanding, which parts of the problem are most computationally intensive
  - Focus the effort here

Overview

- Decomposition patterns
  - Decompose the problem into pieces that can execute concurrently
- Dependency Analysis patterns
  - Group tasks and analyze the dependencies among them
  - Theoretically, patterns should be used in order
  - Practically, work back and forth between them, possible even revisit decomposition
- Design Evaluation pattern
  - Final pattern, checkpoint before moving to the Algorithm Structure design space
  - Best design not found on the first attempt

Using decomposition patterns

- Decomposition occurring in two dimensions
  - Task-decomposition
    - Problem is a stream of instructions that can be broken into sequences called tasks that can execute simultaneously
    - To be efficient, the operations from one task should be largely independent from other tasks
  - Data-decomposition
    - Focuses on the data required by the tasks
    - How can it be decomposed into distinct chunks?
    - The computation associated with the data chunk must be relatively independent

The Task Decomposition pattern

- Problem
  - How can a problem be decomposed into tasks that can execute concurrently?

The Task Decomposition pattern

- Context
  - Starting point
  - Sometimes the problem naturally breaks down into a collection of independent (nearly) tasks
  - In other cases, the tasks are difficult to isolate, then the data-decomposition is a better starting point
  - Not always clear, which approach is best, so consider both
  - Regardless of used starting point, tasks must be identified

The Task Decomposition pattern

- Forces
  - Flexibility
    - Allows adaptation to different implementation requirements
    - Do not stick with a single computer system or style of programming
  - Efficiency
    - A parallel program is only useful if it scales efficiently with the size of the parallel computer or problem
    - We need enough tasks to keep PEs busy with enough work per task to compensate for overhead incurred by managing dependencies
  - Simplicity
    - Complex enough to get the job done
    - Simple enough to be debugged and maintained

The Task Decomposition pattern

- Solution
  - Tasks sufficiently independent
    - Managing dependencies takes only a small fraction
  - Execution of the tasks can be evenly distributed among PEs
  - Usually done by hand
  - Sometimes complete recasting the problem
  - Look at the problem as a collection of distinct tasks
    - Actions to solve the problem
    - Are these actions distinct and independent?

The Task Decomposition pattern

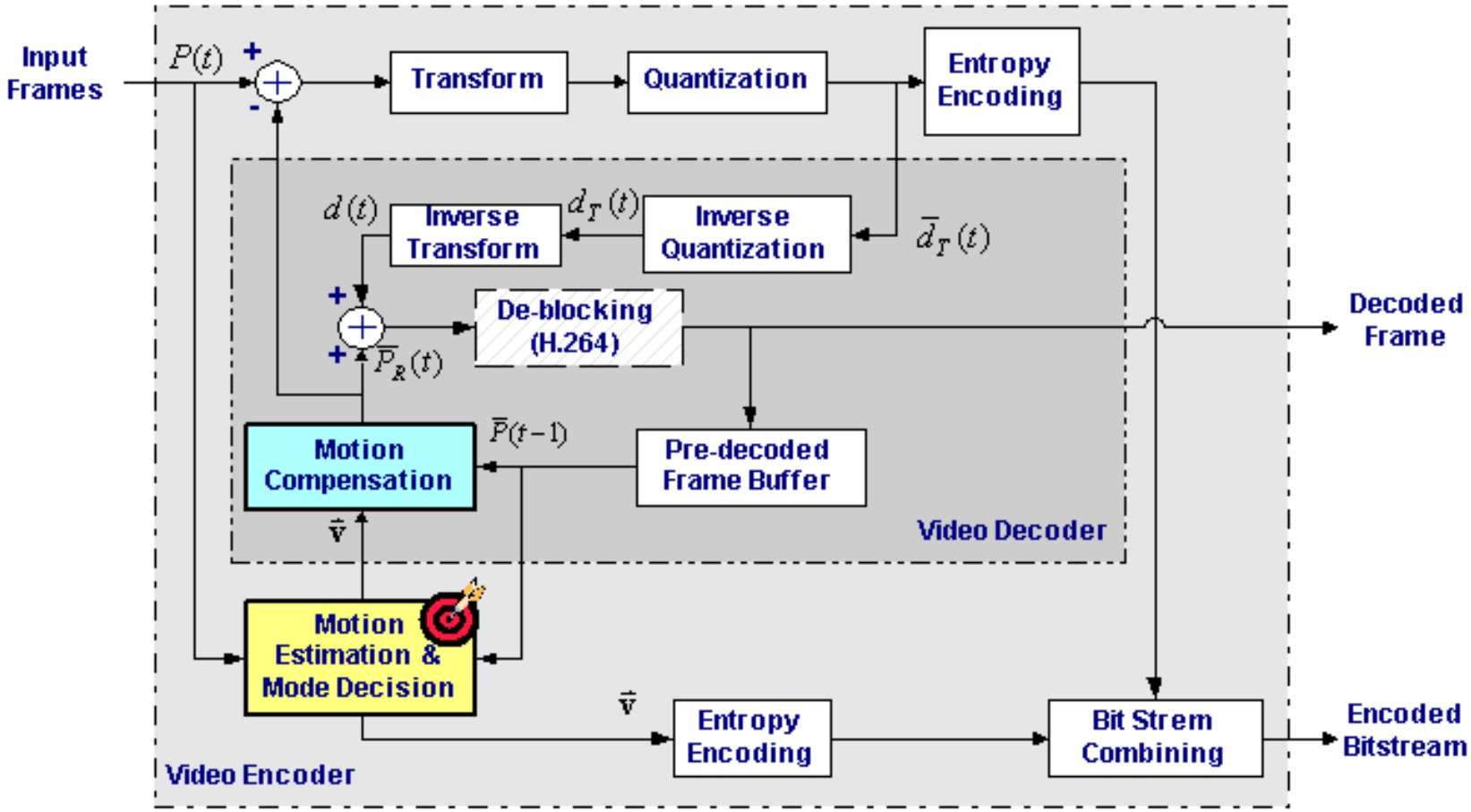
- Solution – cont.
  - First pass
    - Identify as many tasks as possible
  - Where to find tasks?
    - Distinct call to a function
      - Task for each function – functional decomposition
    - Distinct iterations
      - Independent iterations, enough of them
      - Loop-splitting algorithms
    - Data-driven decomposition
      - Large data structure decomposed into chunks
      - Task updates on individual chunk

The Task Decomposition pattern

- Solution – cont.
  - Flexibility
    - Parameterizing the number and size of tasks
  - Efficiency
    - Each tasks must include enough work to compensate for overhead incurred by creating the tasks and managing their dependencies
    - The number of tasks should be large enough – all PEs are busy
  - Simplicity
    - Simple debugging and maintenance
    - Reuse code from existing sequential programs
  - After the tasks have been identified, continue to data-decomposition

The Task Decomposition pattern

- H.264



The Data Decomposition pattern

- Problem
  - How can a problem's data be decomposed into units that can be operated on relatively independently?

The Data Decomposition pattern

- Context
  - Starting point when
    - The most computationally intensive part is organized around the manipulation of a large data structure
    - Similar operations are being applied to different parts of the data structure in relatively independent manner

The Data Decomposition pattern

- Forces
  - Flexibility
    - Allows adaptation to different implementation requirements
    - Do not stick with a single computer system or style of programming
  - Efficiency
    - A parallel program is only useful if it scales efficiently with the size of the parallel computer or problem
    - We need enough tasks to keep PEs busy with enough work per task to compensate for overhead incurred by managing dependencies
  - Simplicity
    - Complex enough to get the job done
    - Simple enough to be debugged and maintained

The Data Decomposition pattern

- Solution
  - In shared-memory environments (OpenMP), the data decomposition will frequently be implied by the task decomposition
  - In most cases, the decomposition is done by hand
    - Distributed memory, NUMA
    - Data dependencies too complex
  - If task decomposition finished
    - Data decomposition driven by the needs of each task
    - Simple for well-defined and distinct data

The Data Decomposition pattern

- Solution – cont.
  - Data decomposition as a start point
    - Look for control data structures defining the problem
    - Consider breaking them down into chunks that can be operated on concurrently
      - Array based computations – updates of different segments of the array
      - Recursive data structures – decomposing parallel update of a large tree

The Data Decomposition pattern

- Solution – cont.
  - Flexibility
    - The size and number of data chunks should be flexible - parameters
      - Granularity knobs
    - Check the impact of granularity in the overhead required to manage dependencies among chunks
      - The dependencies scale at a lower dimensions than the computational effort associated with each chunk
      - Example: boundaries

The Data Decomposition pattern

- Solution – cont.
  - Efficiency
    - Data chunks large enough that the amount of work to update the chunk offsets the overhead of managing dependencies
    - How the chunks are mapped onto UEs?
      - Balance the load between UEs
  - Simplicity
    - Overly complex data decomposition can be very difficult to debug
    - Mapping of a global index space onto a task-local index space
      - Make it abstract

The Data Decomposition pattern

- Examples
  - Matrix multiplication
    - Decomposition into set of rows
    - Decomposition into blocks (submatrices)

The Group Tasks pattern

- Problem
  - How can the tasks that made up a problem grouped to simplify the job of managing dependencies?

The Group Tasks pattern

- Context
  - Applied after the task and data decomposition
  - First step in analyzing dependencies
  - Considerable structure to the set of tasks
    - Grouping of tasks
    - Simplify a problem’s dependency analysis
  - Grouping
    - Sharing temporal constraint
    - Working together on a shared data structure
    - Independent tasks

The Group Tasks pattern

- Solution
  - Temporal dependency
    - Easiest
    - Constraint on the order in which a collection of tasks executes
      - Task A must wait until task B completes
  - Running at the same time
    - Often in data-parallel problems
      - Processing boundaries of its neighboring regions
  - Tasks truly independent
    - No ordering constraint among them

The Group Tasks pattern

- Solution – cont.
  - Why?
    - Simplify ordering between tasks
      - Ordering constraints applied to groups rather than to individual tasks
    - Easier identification which tasks can execute concurrently
  - Many ways to group tasks
    - Simplify the dependency analysis

The Group Tasks pattern

- Solution – cont.
  - Look at how the original problem was decomposed
    - High-level operation
      - Solving a matrix
    - Large iterative program structure
      - Loop
    - Many small groups of tasks
    - Check for constraints shared between the tasks within a group
      - If they share a constraint, keep them as a distinct group. They will execute at the same time

The Group Tasks pattern

- Solution – cont.
  - Check, if any other task groups share the same constraint
    - If so, merge
    - Large task groups provide additional concurrency
      - Keep PEs busy
      - Flexibility, balancing
  - Constraints between groups of tasks
    - Easy for a clear temporal ordering or when a distinct chain of data moves between groups
    - Complex case, when otherwise independent task groups share constraints between groups
      - Merge

The Group Tasks pattern

- Examples
  - Matrix multiplication
    - Grouping rows or blocks

The Order Tasks pattern

- Problem
  - Given a way decomposing a problem into tasks and a way of collecting these tasks into logically related groups, how must these groups of tasks be ordered to satisfy constraints among tasks?

The Order Tasks pattern

- Context
  - The second step in analyzing dependencies
  - Constraints among tasks fall into a few major categories
    - Temporal dependency
    - Tasks must execute at the same time
    - Total independence
  - Help find and correctly account for dependencies resulting from constraints on the order of execution of collection of tasks

The Order Tasks pattern

- Solution
  - Goals
    - The ordering must be restrictive enough to satisfy all the constraints
    - The ordering should not be more restrictive than it needs to be
      - Overly constraining can impair program efficiency

The Order Tasks pattern

- Solution – cont.
  - Identifying ordering constraints
    - Data required by a group of tasks before they can execute
      - Find the task group that creates it
      - These two groups must execute in sequence
    - Ordering constraints by external services
      - Writing a file in a certain order
    - Ordering constraint does not exist
      - Opportunity to execute independently

The Data Sharing pattern

- Problem
  - Given a data and task decomposition for a problem, how is data shared among the tasks?

The Data Sharing pattern

- Context
  - Every parallel algorithm consists of
    - A collection of tasks that can execute concurrently
    - A data decomposition corresponding to the collection of concurrent tasks
    - Dependencies among the tasks that must be managed to permit safe concurrent execution
  - Dependency analysis (group tasks, order tasks)
    - Group tasks using ordering constraints
  - How data is shared among groups of tasks?
    - Access to shared data can be managed correctly

The Data Sharing pattern

- Context – cont.
  - Shift focus to the data decomposition
    - Division of the problem's data into chunks that can be updated independently
    - Task-local data
    - Each task operating only its own local data
      - Rare situation

The Data Sharing pattern

- Context – cont.
  - Two common situations
    - Some data must be shared among tasks, it cannot be identified with any given task, it is global to the problem, it is modified by multiple tasks, it is source of dependencies among them
    - One task needs access to some portion of another task's local data

The Data Sharing pattern

- Forces
  - Race condition
    - Often in shared-address-space environment
  - Excessive synchronization overhead
    - Barrier for all but only a small partition needs a synchronization
  - Communication overhead
    - Minimize amount of shared data to communicate

The Data Sharing pattern

- Solution
  - Identify shared data
    - Data decomposition
      - Borders between neighboring blocks
    - Task decomposition
      - How data is passed in or out of the task

The Data Sharing pattern

- Solution – cont.
  - How it is used
    - Read-only
      - No protection
      - Replicate on distributed memory systems
    - Effectively-local
      - Data partitioned into subsets accessed RW by only one of the tasks
      - Distribute on distributed memory, can be recombined
    - Read-write
      - Accessed RW by more than one task
      - Most difficult

The Data Sharing pattern

- Solution – cont.
  - Special cases for RW data
    - Accumulate
      - Associative accumulation operation
      - Each task has a separate copy
      - Accumulated into a single global copy as a final step
    - Multiple read/single write
      - Algorithms based on data decomposition
      - Two copies – one for preserving the initial value, one for modifying task

The Design Evaluation pattern

- Problem
  - Is the decomposition and dependency analysis so far good enough to move on to the next design space, or should the design be revisited?

The Design Evaluation pattern

- Context
  - The original problem has been decomposed and analyzed to produce:
    - A task decomposition
    - A data decomposition
    - A way of grouping tasks and ordering the groups to satisfy constraints
    - An analysis of dependencies among tasks
  - Multiple decompositions
    - Find the best one

The Design Evaluation pattern

- Forces
  - Suitability for the target platform
    - The more the design depends on the target architecture, the less flexible it will be.
  - Design quality
    - Simplicity, flexibility, efficiency
  - Preparation for the next phase of the design

The Design Evaluation pattern

- Solution
  - Suitability for target platform
    - How many PEs are available?
      - More tasks than PEs keep all the PEs busy
      - Poor load balance for only one or a few tasks per PE
      - Exception: the number of tasks can be adjusted to fit the number of PEs with good load balance
    - How are data structures shared among PEs?
      - Extensive data sharing is easier to implement on a shared memory
      - Fine-grained data-sharing is more efficient on a shared-memory machine
      - Exception: group and schedule tasks that the only large-scale or fine-grained data sharing among tasks assigned to the same UE

The Design Evaluation pattern

- Solution – cont.
  - Suitability for target platform – cont.
    - What does the target architecture imply about the number of UEs and how structures are shared among them?
      - Revisit the preceding two questions in terms of UEs rather than PEs
      - Use more UEs then PEs?
        - Efficient support for multiple UEs per PE
        - The design can make good use of multiple UEs per PE – communication with high latency

The Design Evaluation pattern

- Solution – cont.
  - Suitability for target platform – cont.
    - On the target platform, will the time spent doing useful work in a task be significantly greater than the time taken to deal with dependencies?
      - Compute the ratio of time spent doing computation to time spent in communication or synchronization
        - Higher is better
        - Affected by problem size relative to the number of available PEs

The Design Evaluation pattern

- Solution – cont.
  - Design quality
    - Flexibility
      - Is the decomposition flexible in the number of tasks generated?
        - Adaptability to a wide range of parallel computers
      - Is the definition of tasks implied by the task decomposition independent of how they are scheduled for execution?
        - Easier load balancing
      - Can the size and number of chunks in the data decomposition be parameterized?
        - Easier scaling
      - Does the algorithm handle the problem's boundary cases?
        - Handle all relevant cases

The Design Evaluation pattern

- Solution – cont.
  - Design quality – cont.
    - Efficiency
      - Can the computational load be evenly balanced among the PEs?
        - Easier for independent tasks or tasks of roughly the same size
      - Is the overhead minimized?
        - Creation and scheduling of UEs
        - Communication – latency, small number of messages
        - Synchronization – shared memory, keep data well-localized to a task

The Design Evaluation pattern

- Solution – cont.
  - Design quality – cont.
    - Simplicity
      - Make it as simple as possible, but not simpler

The Design Evaluation pattern

- Solution – cont.
  - Preparation for the next phase
    - How regular are the tasks and their data dependencies?
      - The scheduling of the tasks and their sharing of data will be more complicated for irregular tasks
    - Are interactions between tasks (or task groups) synchronous or asynchronous?
    - Are the tasks grouped in the best way?