Algorithm structure

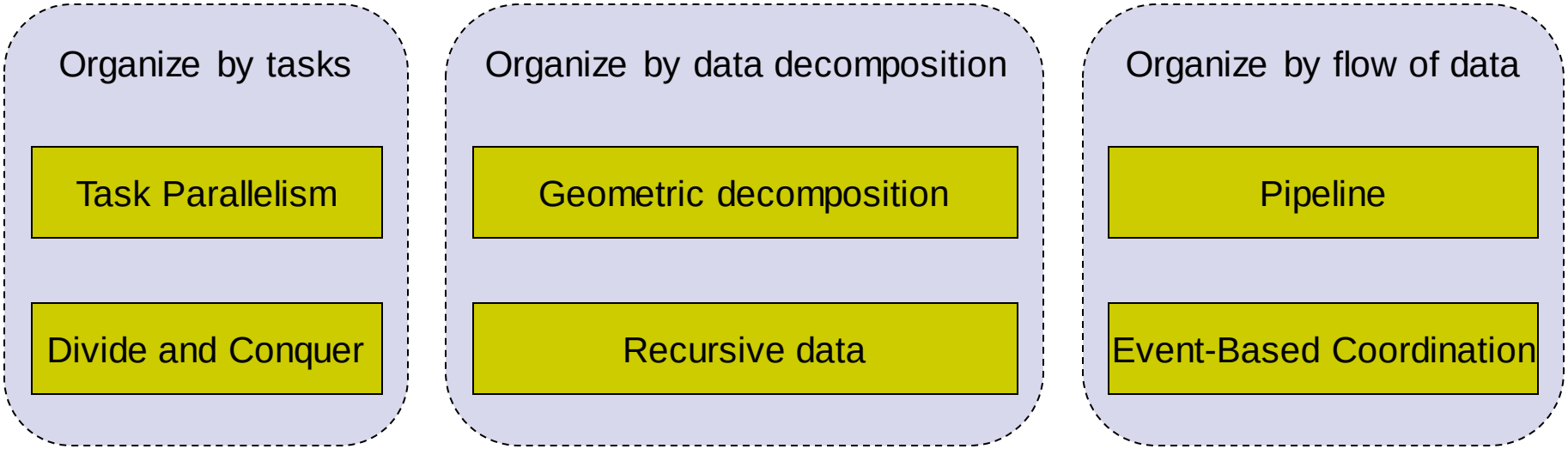
Jakub Yaghob

Algorithm structure design space

- Refine the design and move it closer to a program
- Six basic design patterns
- Decide, which pattern or patterns are most appropriate

Overview

Algorithm structure



Choosing an algorithm structure pattern

- Target platform
  - What constraints are placed on the parallel algorithm by the target machine or programming environment?
    - Not necessary at this stage of the design in an ideal world
  - How many UEs the system will effectively support?
    - An order of magnitude

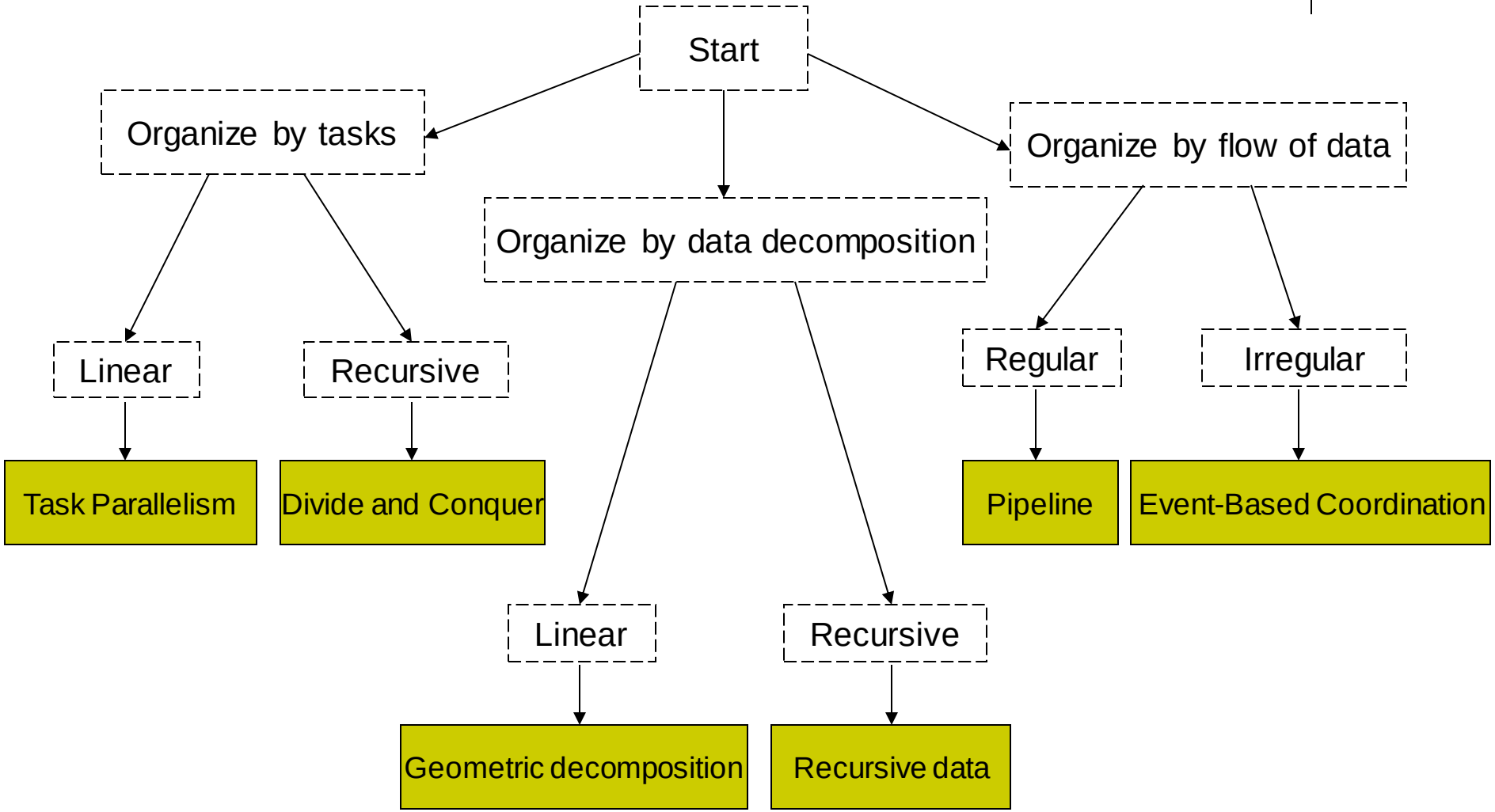
Choosing an algorithm structure pattern

- Target platform – cont.
  - How expensive it is to share information among UEs?
    - HW support for shared memory – frequent data sharing
    - Collection of nodes connected by a slow network – avoid information sharing
  - Do not over-constrain the design
    - Obtain a design that works well on the original target platform, but at the same time is flexible enough to adapt to different classes of HW

Choosing an algorithm structure pattern

- Major organizing principle
  - Implied by the concurrency
  - Organization by tasks
    - Only one group of tasks active at one time
    - Embarrassingly parallel problems
  - Organization by data decomposition
    - Data decomposition and sharing among tasks is the major way to organize concurrency
  - Organization by flow of data
    - Well-defined interacting groups of tasks
    - Data flow among the tasks
  - Effective parallel algorithm design uses multiple algorithm structures

The Algorithm structure decision tree



The Algorithm structure decision tree

- Organize by tasks
  - Execution of the tasks themselves
  - How tasks are enumerated?
    - Linear set in any number of dimensions
      - Including embarrassingly parallel problems
      - Dependencies among the tasks in the form of access to shared data or a need to exchange messages
    - Tasks enumerated by a recursive procedure

The Algorithm structure decision tree

- Organize by data decomposition
  - Decomposition of the data is the major organizing principle
  - How the decomposition is structured
    - Linearly in each dimension
      - Problem space is decomposed into discrete subspaces
      - Typically requiring data from a small number of other subspaces
    - Recursively
      - Following links through a recursive data structure

The Algorithm structure decision tree

- Organize by flow of data
  - How the flow of data imposes an ordering on the groups of tasks
    - Ordering regular and static
    - Ordering irregular and/or dynamic
- Re-evaluation
  - Is the Algorithm structure pattern suitable for the target platform?

The task parallelism pattern

- Problem
  - When the problem is best decomposed into a collection of tasks that can execute concurrently, how can this concurrency be exploited efficiently?

The task parallelism pattern

- Context
  - Design is based directly on the tasks
  - The problem can be decomposed into a collection of tasks that can execute concurrently
  - In many cases, all of the tasks are known at the beginning of the computation
  - Usually all tasks must be completed before the problem is done
    - Sometimes the solution is reached without completing all tasks

The task parallelism pattern

- Forces
  - Assign tasks to UEs
    - Ideally simple, portable, scalable, efficient
    - Balancing the load
  - Dependencies must be managed correctly
    - Simplicity, portability, scalability, efficiency

The task parallelism pattern

- Solution
  - Designs for task-parallel algorithms
    - The tasks and how they are defined
    - The dependencies among tasks
    - The schedule
      - How the tasks are assigned to UEs
    - Tightly coupled

The task parallelism pattern

- Solution – cont.
  - Tasks
    - At least as many tasks as UEs, preferably many more
      - Greater flexibility in scheduling
    - The computation associated with each task must be large enough to offset the overhead associated with managing the tasks and handling any dependencies

The task parallelism pattern

- Solution – cont.
  - Dependencies
    - Two categories
      - Ordering constraints
      - Dependencies related to shared data
    - Ordering constraints
      - Apply to task groups
      - Handled by forcing the groups to execute in the required order
    - Shared-data dependencies
      - No dependencies among the tasks

The task parallelism pattern

- Solution – cont.
  - Dependencies
    - Removable dependencies
      - Not true dependency between tasks
      - Can be removed by simple code transformation
      - Temporary variables local to each task
        - Copy of the variable local to each UE
      - Iterative expressions
        - Transformation into closed-form expression

The task parallelism pattern

- Removable dependencies example

```
int ii=0, jj=0;
for(int i=0;i<N;++i){
  ii = ii+1;
  d[ii] = big_time_consuming_work(ii);
  jj = jj+I;
  a[jj] = other_big_calc(jj);
}

for(int i=0;i<N;++i){
  d[i] = big_time_consuming_work(i);
  a[(i*i+i)/2] = other_big_calc((i*i+i)/2);
}
```

The task parallelism pattern

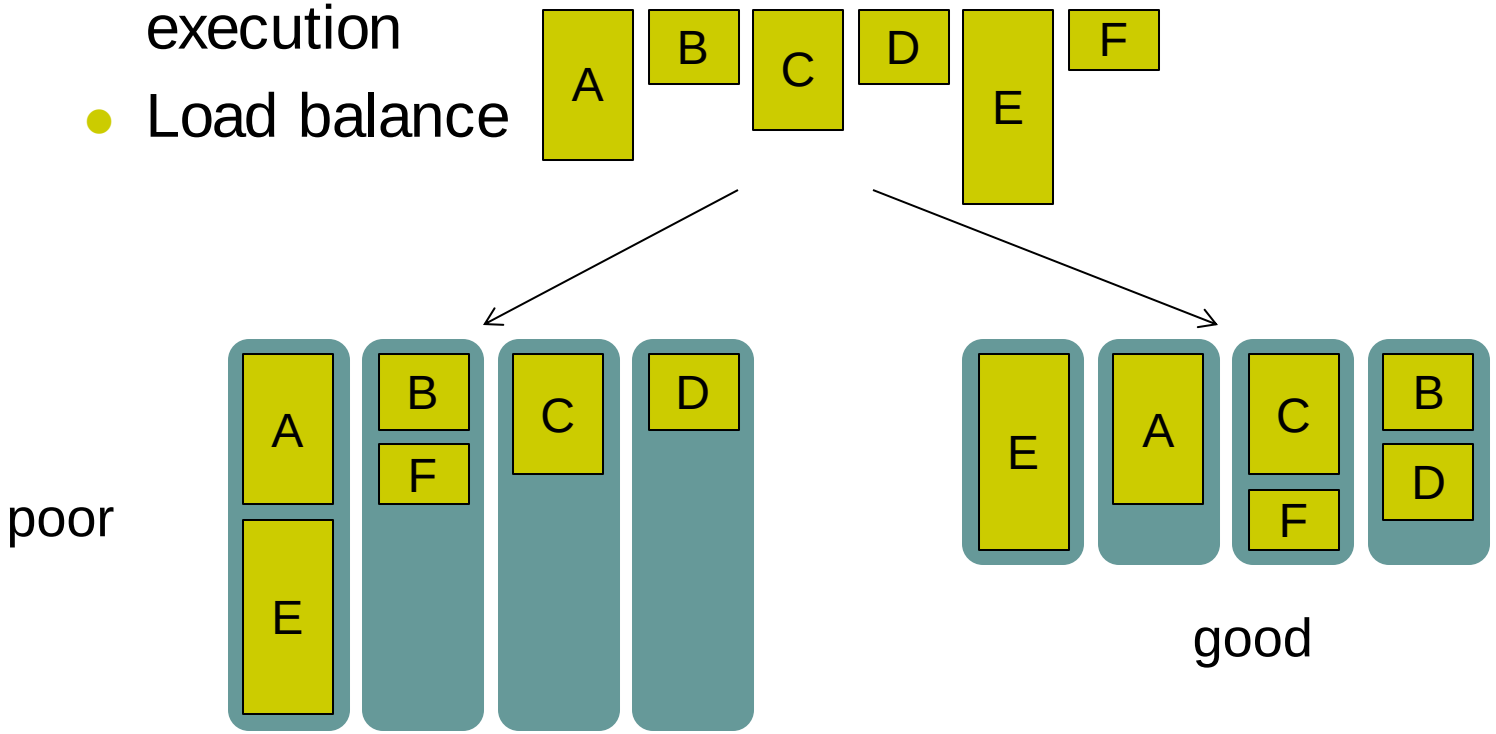
- Solution – cont.
  - Dependencies
    - Separable dependencies
      - Accumulation into a shared data structure
      - Replicating the data structure at the beginning of the computation
      - Executing the tasks
      - Combining copies into a single data structure after the task complete
      - Reduction operation – repeatedly applying a binary operation (addition, multiplication)

The task parallelism pattern

- Solution – cont.
  - Dependencies
    - Other dependencies
      - Shared data cannot be pulled out of the tasks
      - RW by the tasks
      - Data dependencies explicitly managed within the tasks

The task parallelism pattern

- Solution – cont.
  - Schedule
    - Assigning tasks to UEs and scheduling them for execution
    - Load balance



The task parallelism pattern

- Solution – cont.
  - Schedule
    - Static schedule
      - Distribution of tasks among UEs is determined at the start of the computation and does not change
      - Tasks are associated into blocks and then assigned to UEs
      - Each UE takes approximately the same amount of time to complete its tasks
      - Computational resources available from the UEs are predictable and stable
        - The most common case being identical UEs
      - When the effort associated with the tasks varies considerably, the number of blocks assigned to UEs must be much greater then the number of UEs

The task parallelism pattern

- Solution – cont.
  - Schedule
    - Dynamic schedule
      - The distribution of tasks among UEs varies as the computation proceeds
      - Used when
        - The effort associated with each task varies widely and is unpredictable
        - The capabilities of the UEs vary widely and unpredictably
      - Task queue used by all the UEs
      - Work stealing
        - Each UE has its own work queue
        - The tasks are distributed among the UEs at the start of the computation
        - When the queue is empty, the UE will try to steal work from the queue on some other UE (usually randomly selected)

The task parallelism pattern

- Solution – cont.
  - Program structure
    - Many task-parallel problems are loop-based
      - Use the Loop parallelism design pattern
    - Task queue
      - Loop parallelism is not a good fit
      - Not all tasks known initially
      - Use the Master/Worker or SPMD pattern
    - Terminating before all the tasks are complete
      - Termination condition
        - True means the computation is complete
      - The termination condition is eventually met
      - When the termination condition is met, the program ends

The task parallelism pattern

- Solution – cont.
  - Common idioms
    - Embarrassingly parallel
      - No dependencies among tasks
    - Replicated data
      - Dependencies can be managed by “separating them from the tasks”
      - Three phases
        - Replicate the data into local variables
        - Solve the now-independent tasks
        - Recombine the results into a single result

The divide and conquer pattern

- Problem
  - Suppose the problem is formulated using the sequential divide-and-conquer strategy. How can the potential concurrency be exploited?

The divide and conquer pattern

- Context
  - Splitting the problem into a number of smaller subproblems, solving them independently, and merging the subsolutions into a solution for the whole problem
  - The subproblems can be solved directly, or they can in turn be solved using the same divide-and-conquer strategy

The divide and conquer pattern

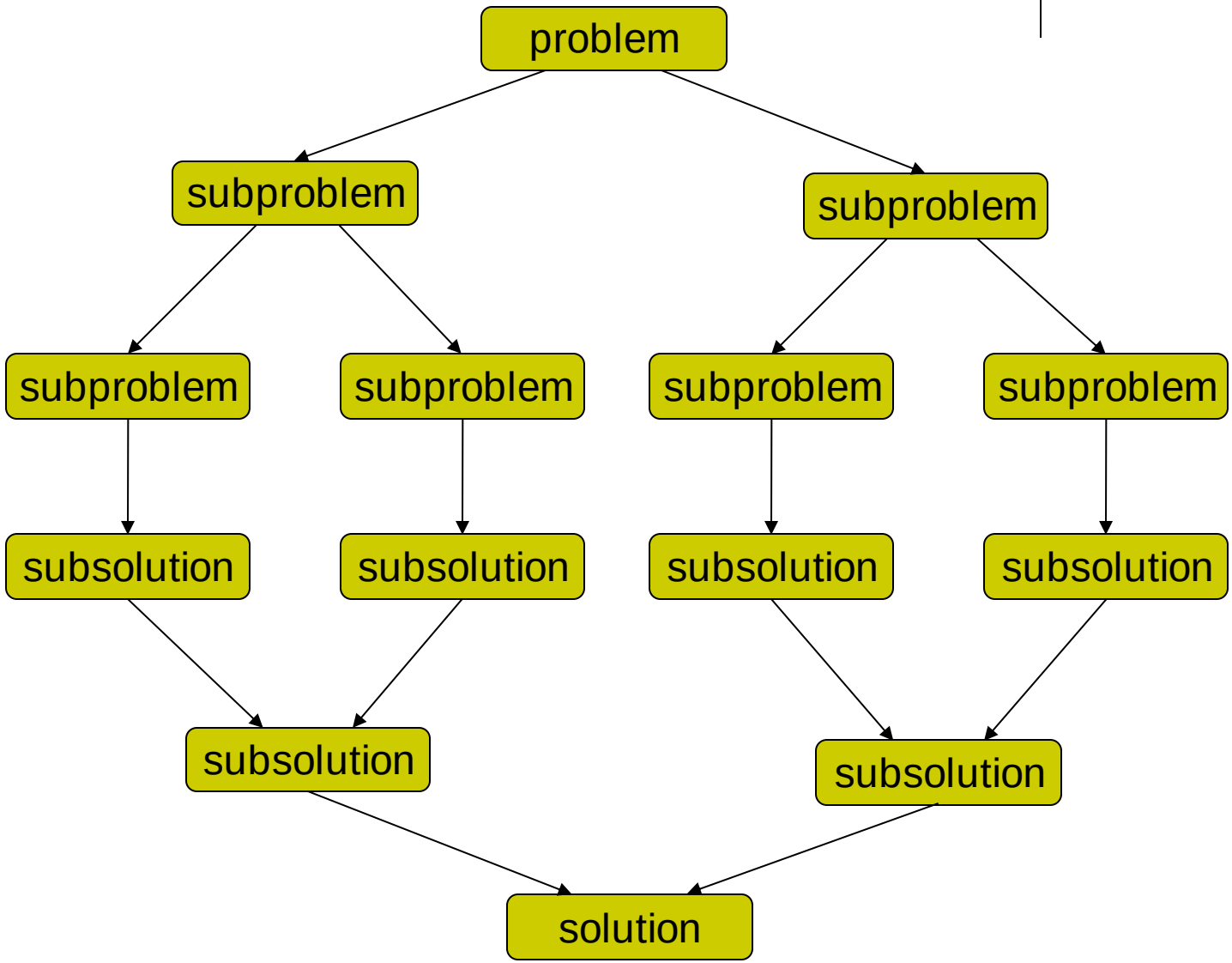
sequential

up to 2-way concurrency

up to 4-way concurrency

up to 2-way concurrency

sequential



The divide and conquer pattern

- Forces
  - Widely useful approach to algorithm design
  - The amount of exploitable concurrency varies over the life of the program
    - If the split and merge computations are nontrivial compared to the amount of computation for the base cases, then this pattern might not be able to take advantage of large number of PEs
    - If there are many levels of recursion, the number of tasks can grow quite large, that the overhead of managing the tasks is too large
  - In distributed-memory systems, subproblems can be generated on one PE and executed by another
    - Data and results moving between the PEs
    - Amount of data associated with a computation should be small
  - The tasks are created dynamically as the computation proceeds
    - In some cases, the resulting graph has irregular and data-dependent structure
      - Dynamic load balancing

The divide and conquer pattern

- Solution
  - Sequential algorithm
    - A recursively invoked function drives each stage
    - Inside the recursive function the problem is either split into smaller subproblems or it is directly solved
    - Recursion continues until the subproblems are simple enough to be solved directly
  - Modification for a parallel version
    - The direct solver should be called when
      - The overhead of performing further splits and merges significantly degrades performance
      - The size of the problem is optimal for the target system

The divide and conquer pattern

- Solution – cont.
  - Mapping tasks to UEs and PEs
    - Fork/Join pattern
    - Subproblems have the same size
      - Map each task to a UE
      - Stop the recursion when the number of active subtasks is the same as the number of PEs
    - Subproblems not regular
      - Create more finer-grained tasks
      - Use Master/Worker pattern
      - Queue of tasks, pool of UEs, typically one per PE

The divide and conquer pattern

- Solution – cont.
  - Communication costs
    - How to efficiently represent the parameters and results
    - Consider to replicate some data at the beginning of the computation
  - Dealing with dependencies
    - Sometimes the subproblems require access to a common data structure
    - Use techniques from Shared data pattern
  - Other optimizations
    - Serial split and merge sections limit the scalability
    - Reduce the number of levels of recursion
    - One-deep divide and conquer
      - Initial split into P subproblems, where P is the number of PEs

The geometric decomposition pattern

- Problem
  - How can an algorithm be organized around a data structure that has been decomposed into concurrently updatable “chunks”?

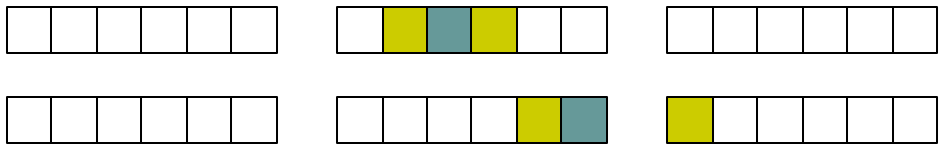
The geometric decomposition pattern

- Context
  - Sequence of operations on a core data structure
  - The concurrency exploited by a decomposition of the core data structure
  - Arrays and other linear data structures
  - Decomposing the data structure into contiguous substructures
    - Analogous to dividing a geometric region into subregions
    - Chunks
  - The decomposition of data into chunks then implies a decomposition of the update operation into tasks, where each task represents the update of one chunk
    - Strictly local computations (all required information is within the chunk) – task parallelism pattern should be used
    - Update requires information from points in other chunks (frequently from neighboring chunks) – information must be shared between chunks

The geometric decomposition pattern

- Example – 1D heat diffusion
  - Initially, the whole pipe is at a stable and fixed temperature
  - At time 0 both ends are set to different temperatures, which will remain fixed
  - How temperatures change in the rest of the pipe over time?
  - Discretize the problem space (U represented as 1D array)
  - At each time step
    - ukp1[i]=uk[i]+(dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1])

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2}$$

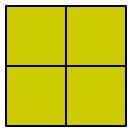


The geometric decomposition pattern

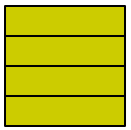
- Solution
  - Data decomposition
    - The granularity of the data decomposition
      - Coarse-grained – small number of large chunks, small number of large messages – reduce communication overhead
      - Fine-grained – large number of small chunks (many more than PEs) – large number of small messages (increases communication overhead), load balancing
      - Granularity controlled by parameters

The geometric decomposition pattern

- Solution – cont.
  - Data decomposition
    - The shape of the chunks
      - The amount of communication between tasks
      - Boundaries of the chunks
      - The amount of shared information scales with the surface area of the chunks
      - The computation scales with the number of points within a chunk – the volume of the region
      - Surface-to-volume effect
      - Higher-dimensional decompositions preferred



$$4\frac{N}{2} = 2N$$



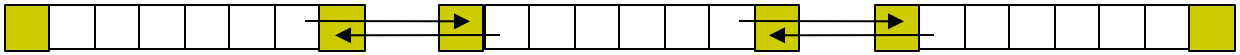
$$2N + 2\frac{N}{4} = 5\frac{N}{2}$$

The geometric decomposition pattern

- Solution – cont.
  - Data decomposition
    - The shape of the chunks
      - Sometimes the preferred shape of the decomposition can be dictated by other concerns
      - Existing sequential code can be more easily reused with a lower-dimensional decomposition
      - An instance of this pattern used as a sequential step in a larger computation, and the decomposition used in an adjacent step differs from the optimal one
      - Data decomposition decisions must take into account the capability to reuse sequential code and the need to interface with other steps in the computation
      - Suboptimal decompositions

The geometric decomposition pattern

- Solution – cont.
  - Data decomposition
    - Ghost boundary
      - Replicating the nonlocal data needed to update the data in a chunk
      - Each chunk has two parts
        - A primary copy owned by the UE (updated directly)
        - Zero or more ghost copies (shadow copies)
      - Ghost copies
        - Consolidate communication into fewer, larger messages
        - Communication of the ghost copies can be overlapped with the update of parts of array that don't depend on data within the ghost copy



The geometric decomposition pattern

- Solution – cont.
  - The exchange operation
    - Nonlocal data required for the update operation is obtained before it is needed
      - All the data needed is present before the beginning of the update
        - Perform the entire exchange before beginning the update
        - Store the required nonlocal data in a local data structure (ghost boundary)
      - Some data needed for the update is not initially available or an improvement of performance
        - Overlapping computation and communication
    - The low-level details of how the exchange operation is implemented can have a large impact on efficiency
      - Optimized implementations of communication patterns
        - Collective communication routines in MPI

The geometric decomposition pattern

- Solution – cont.
  - The update operation
    - Execute the corresponding tasks concurrently
    - All needed data is present at the beginning of the update and non of this data is modified during the update
      - Easier parallelization, efficient
    - The required exchange of information performed before beginning of the update
      - The update is straightforward to implement
    - The exchange and update operations overlap
      - Performing the update correctly
      - Nonblocking communication

The geometric decomposition pattern

- Solution – cont.
  - Data distribution and task scheduling
    - Two-tiered scheme for distributing data among UEs
      - Partitioning the data into chunks
      - Assigning the chunks to UEs
    - Each task is statically assigned to a separate UE
      - The simplest case
      - All tasks can execute concurrently
      - The intertask coordination (the exchange operation) simple
      - The computation times of the tasks are uniform
      - Sometimes poor load balance

The geometric decomposition pattern

- Solution – cont.
  - Data distribution and task scheduling
    - The solution for poor balanced problems
      - Decompose the problem into many more chunks than UEs and scatter them among the UEs with a cyclic or block-cyclic distribution
      - Rule of thumb – 10x as many tasks as UEs
    - Dynamic load balancing
      - Periodically redistribute the chunks among the UEs
      - Incurs an overhead
      - More complex program, consider static cyclic allocation first
  - Program structure
    - Use either Loop Parallelism or SPMD pattern

The recursive data pattern

- Problem
  - Suppose the problem involves an operation on a recursive data structure (list, tree, graph) that appears to require sequential processing. How can operations on these data structures be preformed in parallel?

The recursive data pattern

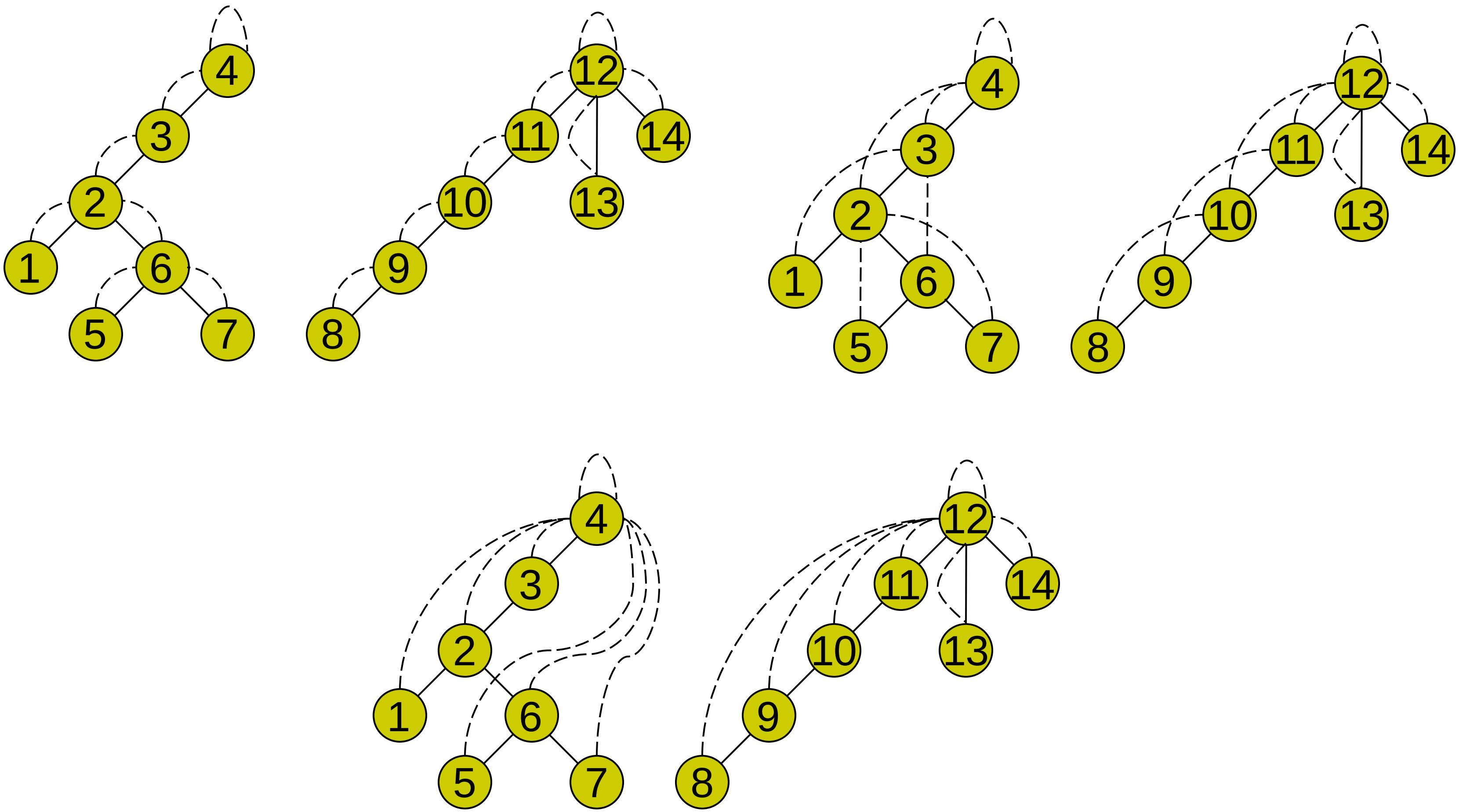
- Context
  - Some problems with recursive data structures seem to have little if any potential for concurrency
    - Only way to solve the problem is to sequentially move through the data structure
  - Sometimes it is possible to reshape the operation in a way that a program can operate concurrently on all elements of the data structure

The recursive data pattern

- Example
  - Forest of rooted directed trees
    - Each node knows its immediate ancestor
  - For each node compute its root of the tree
  - Sequentially trace depth-first through each tree from the root – O(N)
  - Some potential for concurrency operating on subtrees concurrently
  - Rethinking
    - Each node holds “successor” – initially its parent
    - Calculate for each node “successor’s successor”
    - Repeat the calculation until it converges
    - The algorithm converges in at most log N steps
    - More total work than the original sequential one O(N log N)
    - Reduces total running time O(log N)

The recursive data pattern

- Example – cont.



The recursive data pattern

- Forces
  - Recasting the problem from a sequential to parallel form
    - Increased total work of the computation
    - Must be balanced by the improved performance
  - The recasting may be difficult to achieve
    - Difficult to understand and maintain a design
  - The exposed concurrency effectively exploited
    - How computationally expensive the operation is
    - The cost of communication relative to computation

The recursive data pattern

- Solution
  - Restructuring the operations over a recursive data structure into a form that exposes additional concurrency
    - The most challenging part
    - No general guidelines
  - After the concurrency has been exposed, it is not always the case that the concurrency can be effectively exploited to speed up the solution of a problem
    - How much work is involved as each element of the recursive data structure is updated
    - Characteristics of the target parallel computer

The recursive data pattern

- Solution – cont.
  - Data decomposition
    - The recursive data structure completely decomposed into individual elements
    - Each element is assigned to a separate UE
    - Ideally each UE assigned to a different PE
    - It is possible to assign multiple UEs to each PE
      - The number of UEs per PE too large
        - Poor performance
        - Not enough concurrency to overcome the increase in the total amount of work

The recursive data pattern

- Solution – cont.
  - Data decomposition – example
    - Root-finding problem
    - Ignore overhead in computations
    - N = 1024, t is the time to perform one step for one data element
    - Sequential algorithm
      - Running time ≈1024t
    - Each UE assigned to its own PE
      - Running time ≈(log N)t = 10t
    - Only two PEs
      - Running time ≈(N log N)t/2 = 5120t

The recursive data pattern

- Solution – cont.
  - Structure
    - Top-level structure of an algorithm
      - Sequential composition in the form of a loop, each iteration described as “perform this operation simultaneously on all (or selected) elements of the recursive data structure”
    - Typical operations
      - “Replace each element’s successor with its successor’s successor”
      - “Replace a value held at this element with the sum of the current value and the value of the predecessor’s element”

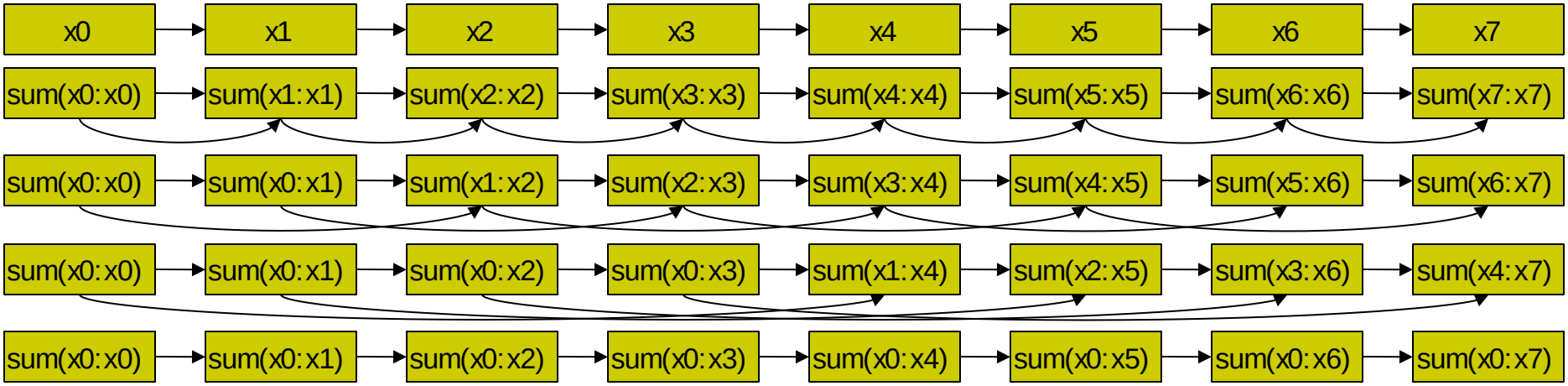
The recursive data pattern

- Solution – cont.
  - Synchronization
    - Simultaneously updating all elements of the data structure
    - Explicit synchronization for steps
      - Example: `next[k] = next[next[k]]`
      - Introduce a new variable `next2`, read from `next`, update `next2`, switch `next` and `next2` after each step
      - Use a barrier
        - Can substantially increase the overhead, which can overwhelm any speedup, when the calculation required for each element is trivial
    - Fewer PEs then data elements
      - Assign each data element to a UE and assign multiple UEs to PE
      - Assign multiple data elements to UE and process them serially
        - More efficient

The recursive data pattern

- Example – partial sums of a linked list

```
for all k in parallel {
   temp[k] = next[k];
   while temp[k] != null {
        x[temp[k] = x[k] + x[temp[k]];
        temp[k] = temp[temp[k]];
   }
}
```



The pipeline pattern

- Problem
  - Suppose that the overall computation involves performing a calculation on many sets of data, where the calculation can be viewed in terms of data flowing through a sequence of stages. How can the potential concurrency be exploited?

The pipeline pattern

- Context
  - An assembly line
  - Examples
    - Instruction pipeline in modern CPUs
    - Vector processing
    - Algorithm-level pipelining
    - Signal processing
    - Graphics
    - Shell programs in UNIX

The pipeline pattern

- Context – cont.
  - Applying a sequence of operations to each element in a sequence of data elements
  - There may be ordering constraints on the operations on a single data element
  - Perform different operations on different data elements simultaneously

The pipeline pattern

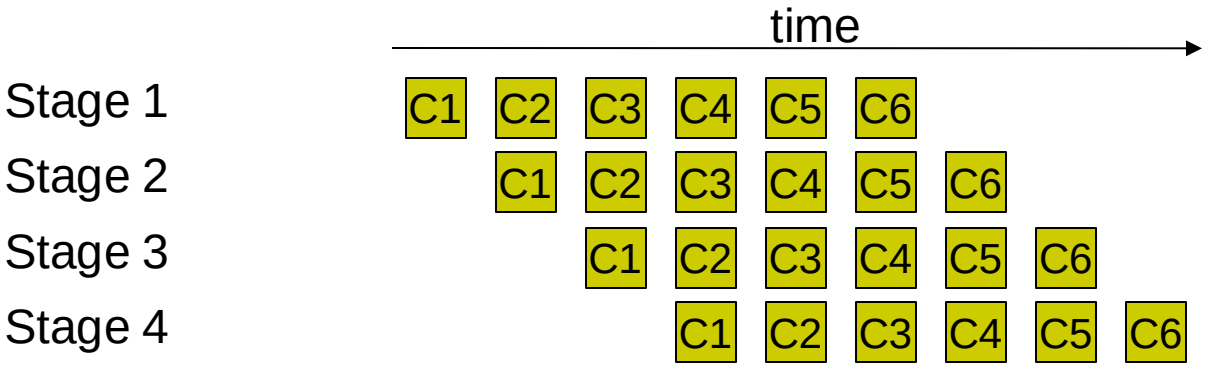
- Forces
  - The ordering constraints are simple and regular
    - Data flowing through a pipeline
  - The target platform can include special-purpose HW that can perform some of the desired operations
  - Sometimes future additions, modifications, or reordering of the stages in the pipeline are expected
  - Sometimes occasional items in the input sequence can contain errors that prevent their processing

The pipeline pattern

- Solution
  - Assign each task (stage of the pipeline) to a UE, provide a mechanism for sending data elements from a stage of the pipeline to the next stage
    - Deals with ordering constraints
    - Allows to take advantage of special-purpose HW by appropriate mapping of pipeline stages to PEs
    - A reasonable mechanism for handling errors
    - A modular design, it can be extended or modified

The pipeline pattern

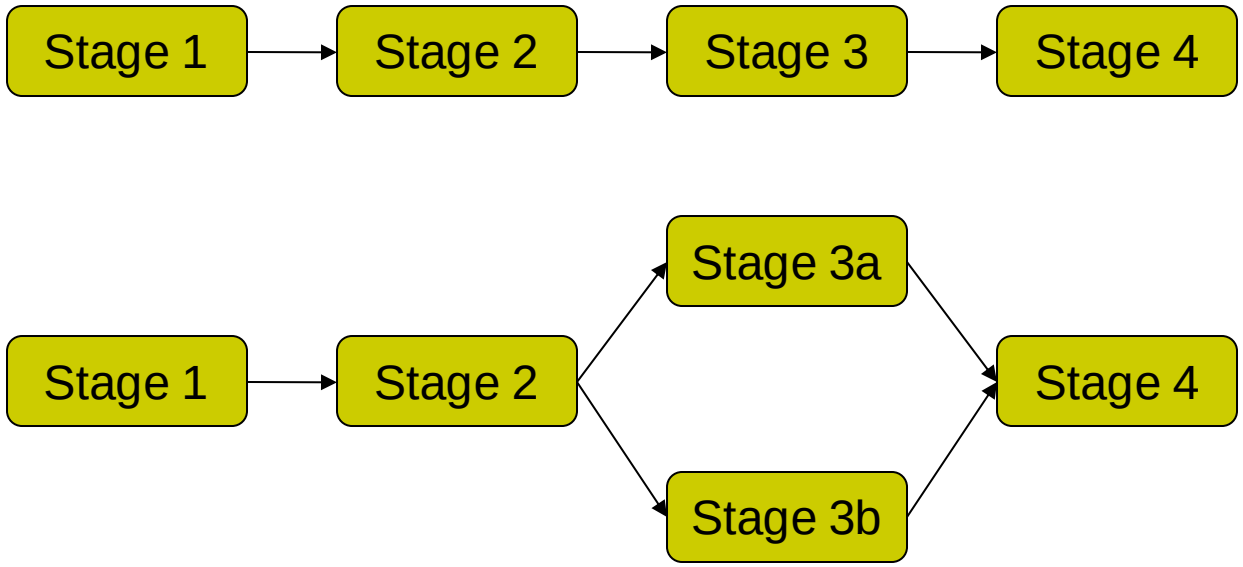
- Solution – cont.



  - Concurrency initially limited
    - Filling the pipeline
  - Limited concurrency at the end of the computation
    - Draining the pipeline
  - We want the time spent filling and draining the pipeline to be small compared to the total time of the computation
    - The number of stages is small compared to the number of items to be processed
  - Overall throughput/efficiency maximized if the time taken to process a data element is roughly the same for each stage

The pipeline pattern

- Solution – cont.
  - Extending the idea from a linear pipeline to more complex situations



The pipeline pattern

- Solution – cont.
  - Defining the stages of the pipeline
    - Normally each pipeline stage corresponds to one task
    - The number of data elements known in advance
      - Each stage can count the number of elements and terminate, when the number is reached
      - Send sentinel indicating termination through the pipeline

```
initialize
while(more data) {
   receive data element from previous stage
   perform operation on data element
   send data element to next stage
}
finalize
```

The pipeline pattern

- Solution – cont.
  - Defining the stages of the pipeline – cont.
    - Factors that affect performance
      - The amount of concurrency in a full pipeline is limited by the number of stages
        - A larger number of stages allows more concurrency
        - Overhead for data transfer
        - Computation work must be large compared to the communication overhead
      - The slowest stage creates a bottleneck
      - Number of stages
        - Fill and drain the pipeline

The pipeline pattern

- Solution – cont.
  - Defining the stages of the pipeline – cont.
    - Revisit the original decomposition
      - Combine lightly-loaded adjacent stages
      - Decompose a heavily-loaded stage into multiple stages
    - Parallelize heavily-loaded stage using one of the other Algorithm structure patterns

The pipeline pattern

- Solution – cont.
  - Structuring the computation
    - Structure the overall computation
      - SPMD pattern, use each UE's ID to select the stage
  - Handling errors
    - Separate task to handle errors

The pipeline pattern

- Solution – cont.
  - Representing the dataflow among pipeline elements
    - Depends on the target platform
    - Message-passing environment
      - Assign one UE to each stage
      - A connection implemented as a sequence of messages
        - Buffered and ordered – stages are not well synchronized
        - Multiple data elements in each message – high latency networks, it reduces total communication cost at the expense of increasing the time needed to fill the pipeline
    - Shared-memory environment
      - Buffered channels
        - Use shared queue pattern

The pipeline pattern

- Solution – cont.
  - Representing the dataflow among pipeline elements – cont.
    - Stages as parallel programs
      - Data redistribution between the stages
      - Example: one stage partitions each data element into three subsets, another stage partitions it into four subsets
      - Aggregate and disaggregate data element between stages
        - Only one task in each stage communicate with tasks in other stages, it redistributes and collects data to/from other tasks in the stage
        - Additional pipeline stages for aggregation/disaggregation
      - The earlier stage “knows” about the needs of its successor
        - No aggregation, improved performance, reduced simplicity and modularity
      - Networked file systems

The pipeline pattern

- Solution – cont.
  - Processor allocation and task scheduling
    - Allocate one PE to each stage of the pipeline
      - Good load balance for similar PEs, the amount of work for a data element roughly the same
      - Stages with different requirements assigned to proper PEs
    - Fewer PEs than pipeline stages
      - Multiple stages assigned to the same PE
        - It improves or at least does not much reduce overall performance
        - Stages that do not share many resources
        - Stages involving less work
        - Assigning adjacent stages reduces communication costs
        - Combine adjacent stages into a single stage

The pipeline pattern

- Solution – cont.
  - Processor allocation and task scheduling – cont.
    - More PEs than pipeline stages
      - Parallelize one or more stages using appropriate Algorithm structure pattern
        - Useful for the stage which was previously bottleneck
      - If there are no temporal constraints among the data items themselves
        - Run multiple independent pipelines in parallel – improves throughput of the overall calculation, but does not improve the latency

The pipeline pattern

- Solution – cont.
  - Throughput and latency
    - Throughput
      - The number of data items processed per time unit after the pipeline is already full
    - Latency
      - The amount of time between the generation of an input and the completion of processing of that input
      - Real-time applications require small latency

The event-based coordination pattern

- Problem
  - Suppose the application can be decomposed into groups of semi-independent tasks interacting in an irregular fashion. The interaction is determined by the flow of data between them which implies ordering constraints between the tasks. How can these tasks and their interaction be implemented so they can execute concurrently?

The event-based coordination pattern

- Context
  - The pipeline
    - The entities form a linear pipeline
    - Each entity interacts only with the entities to either side
    - The flow of data is one-way
    - The interaction occurs at fairly regular, predictable intervals
  - The event-based coordination
    - No restriction to a linear structure
    - No restrictions on one-way flow of data
    - The interaction takes place at irregular or unpredictable intervals

The event-based coordination pattern

- Context – cont.
  - Discrete-event simulation
    - Collection of objects
    - The interaction is represented by a sequence of discrete events
  - Composition of existing (possibly sequential) program components that interact in irregular way into a parallel program without changing the internals

The event-based coordination pattern

- Forces
  - It should be simple to express ordering constraints
    - Numerous, irregular, can arise dynamically
    - Perform as many activities as possible
  - Encoding ordering constraints into the program or using shared variables
    - Not simple, not capable expressing complex constraints, not easy to understand

The event-based coordination pattern

- Solution
  - The data flow expressed using abstraction called event
    - Each event has task that generates it and task that process it
    - An event must be generated before processed – ordering constraint between the tasks
    - Computation within each task consists of processing events

The event-based coordination pattern

- Solution – cont.
  - Defining the tasks
    - The basic structure of a task

```
initialize
while(not done) {
   receive event
   process event
   send events
}
finalize
```

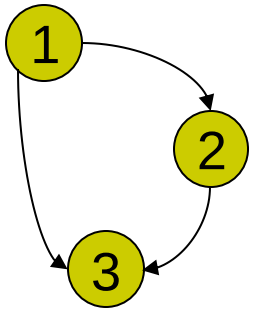
    - The program built from existing components
      - The task uses the Façade pattern
        - Consistent event-based interface to the component

The event-based coordination pattern

- Solution – cont.
  - Representing event flow
    - Communication and computation overlap
    - Asynchronous communication of events
      - Create the event and continue without waiting for the recipient
    - A message-passing environment
      - An asynchronous message
    - A shared-memory environment
      - A queue simulates message passing
        - Safe concurrent access

The event-based coordination pattern

- Solution – cont.
  - Enforcing event ordering
    - The enforcements of ordering constraints may make it necessary for a task to process events in a different order from the order in which are sent, or to wait to process an event until some other event from a given task has been received
      - Look ahead in the queue and remove elements out of order



The event-based coordination pattern

- Solution – cont.
  - Enforcing event ordering – cont.
    - Discrete-event simulation
      - An event has a simulation time, when it should be scheduled
      - An event can arrive to a task before other events with earlier simulation times
      - Determine whether out-of-order events are problem
      - Optimistic approach
        - Requires rollback of effects of events that are mistakenly executed (including effects of any new events created by the out-of-order execution)
        - Not feasible if an event interacts with the outside world
      - Pessimistic approach
        - The events are always executed in order
        - Increased latency and communication overhead
        - Null events – no data only event ordering

The event-based coordination pattern

- Solution – cont.
  - Avoiding deadlocks
    - Possible deadlock at the application level
      - A programming error
      - Problems in the model for a simulation
    - For a pessimistic approach
      - An event is available and actually could be processed, but it is not processed because the event is not yet known to be safe
      - The deadlock can be broken by exchanging enough information that the event can be safely processed
      - The overhead of dealing with deadlocks can cancel the benefits of parallelism
      - Sending frequent enough “null messages”
        - Many extra messages
      - Deadlock detection and resolving
        - Significant idle time before the deadlock is detected and resolved
      - Timeouts

The event-based coordination pattern

- Solution – cont.
  - Scheduling and processor allocation
    - One task per PE
    - Multiple tasks to each PE
      - Load balancing is a difficult problem – irregular structure and dynamic events
      - Task migration – dynamic load balancing
  - Efficient communication of events
    - Communication mechanism must be efficient