Design principles for parallel algorithms

Grama et al.

Introduction to Parallel Computing
Chapter 3

Mikael Rännar

(after material by Robert Granat)

Basic parallel algorithm design

- Serial algorithm design - sequence of steps to solve a given problem with the help of a computer, a kind of ”recipe”
- Nontrivial parallel algorithm design – extends the serial design:
  - Identify parts of the work that can be performed in concurrently
  - Map the concurrent parts onto multiple processes running in parallel
  - Distribute the input, results and intermediate data
  - Managing accesses to data shared by multiple processors
  - Synchronizing the processors during execution
- Especially:
  - Split the work in smaller pieces
  - Assign these pieces to different processors
  - Something also may also fall out during the process...

Conceptions in this lecture

- Decomposition
- Tasks
- Dependency Graphs
- Mapping
- Methods for hiding interaction
- Models for parallel algorithms

Detailed overview

Decomposition (splitting)
- Recursion
- Data
- Exploratory
- Speculative
- Hybrid

Tasks
- Characteristics
- Characteristics of inter-task interaction

Dependency graphs

Mapping
- Static
- Dynamic

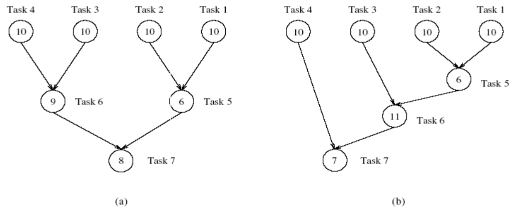
Methods for hiding interaction
- Maximize locality
- Minimize bottle necks
- Overlap computations and communication
- Collective communication operations
- Overlap interactions
- Replicate data
- Extra computations

Models for parallel algorithms
- Data parallel
- Task graph
- Work pool
- Master-slave
- Pipeline

Decomposition – different types of splitting

- Solve a problem in parallel ⇒ split the computations
- The decomposition defines the tasks
- The decomposition defines DAG-dependency graph (Directed, Acyclic Graph, nodes=tasks, edges=dependencies)
- The first (and most important) step in the design of a parallel algorithm
- The choice of the decomposition determines later choises

Exemple of a dependency graph



Granularity and parallelism

- The number of tasks from the decomposition and their size determines the granularity
  - Fine-grained vs. Coarse-grained
- Level of parallelism – ”level of concurrency”
  - Maximum: maximum number of tasks that can be executed simultaneously
  - Average: the average number of tasks that can be executed simultaneously in the whole execution
  - Maximum and average *degree of concurrency* depends on the granularity (normally increases as the granularity becomes finer)

Granularity and parallelism

- P = ”The critical path” in a dependency graph – the longest path between any pair of start and finish nodes.
- L = ”Critical path length”, sum of all work along the critical path
- W = Total amount of work in the parallel algorithm
- A = W / L, gives the average degree of concurrency
- => *Short critical path gives higher degree of concurrency*
- The granularity can/should not be increased too much
  - The problem can have inherent bounds of the granularity
  - Too fine granularity can give other performance problems
    - Bad cache-utilization
    - Too much interaction between tasks

Interaction between tasks

- Normally there exists some kind of interaction between sub-tasks in all parallel programs
- Maybe even between tasks that seems to be completely independent in dependency graph of the program
- The interaction pattern can be described in an task-interaction graph (nodes=tasks, edges=interaction)

Processes and mapping

- Decomposition results in sub-tasks that shall be executed on physical processors
- Mapping is the mechanism that assigns tasks to different processes
- Process = more abstract concept for unit that executes code and uses data belonging to a specific task within the frame of the parallel execution
- Allows for hierarchical mapping of tasks within multiple programming paradigms at the same time
  - E.g.: Message passing between nodes in a parallel computer, where each node is a shared-memory machine with >1 CPU:s (e.g., on Akka where each node is a dual quad core)

Processes and mapping

- The normal situation: #processes = #processors
- A good mapping
  - Maximize the degree of concurrency
  - Minimize the total time for computational work in the parallel program
  - Minimize the interaction between processes in the parallel execution
- Often a good mapping does not fulfill all of these demands due to conflicts, e.g., between parallelism and interaction

---

Recursive decomposition

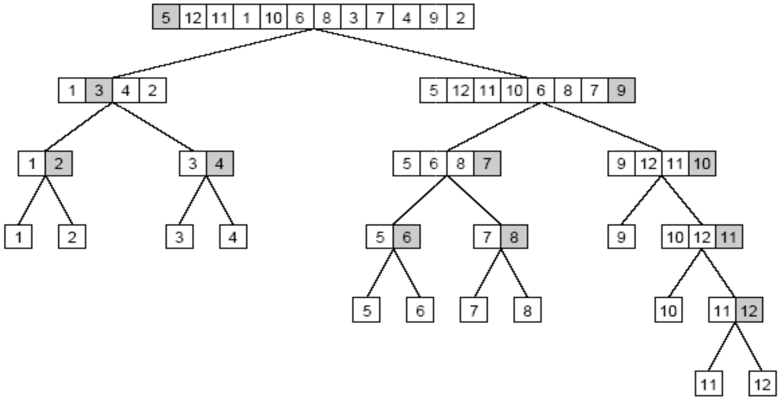
- To solve the problem means that the problem is split into several, smaller, problems of the same kind (*divide*)
- The solutions to the smaller problems must be combined in order to get the solution to the large problem (*conquer*)
- Makes many algorithms simple to express
- Warning! If the conquer step is too large, the overhead may be large
- Example of recursion: computing explicit matrix inverses of upper triangular matrices

$$A^{-1} = \begin{bmatrix} A_{11}^{-1} & -A_{11}^{-1}A_{12}A_{22}^{-1} \\ 0 & A_{22}^{-1} \end{bmatrix}$$

- Compute-intensive conquer step, but with the same complexity
- Lars Karlssons exjobb: http://www.cs.umu.se/~larsk/

Recursive decomposition

- Another example of recursive decomposition – with a small conquer step: quicksort



Data decomposition

In most parallel algorithms it is the large amount of data that is the most significant. Two main steps:

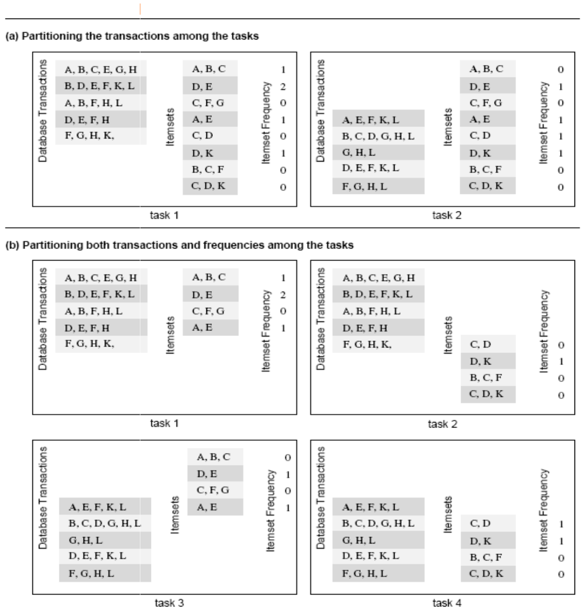
- The data that computations are made on are partitioned
- The data partition defines a partition of the computational work

Different kinds:

- Partition output data
  - Independent output data? E.g.: matrix multiplication C = A*B
- Partition input data
  - How to compute the output data with the help of sub results from partitioned input data?
- Partition both in- and output data
- Partition on sub results
  - Partition based on intermediate sub computation

Data decomposition

Example of partition of input data/input data and output data: computation of frequencies of groups of transactions in a transactional database


(a) Partitioning the transactions among the tasks

(b) Partitioning both transactions and frequencies among the tasks

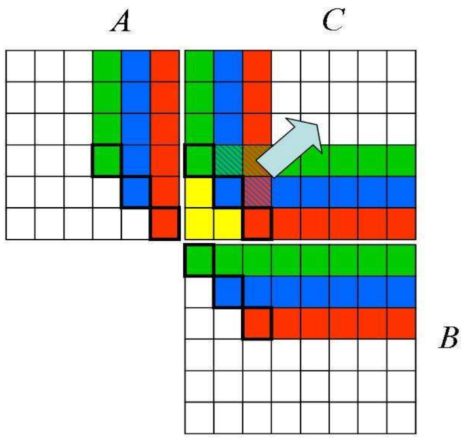
---

”The owner computes” rule

- Every data decomposition of input data and/or output data is also called the ”owner computes ” rule.
- The idea is that the holder of certain part if the data also is resonsible for the comptations that belongs to that part of the data
- Example: wave front algorithms for

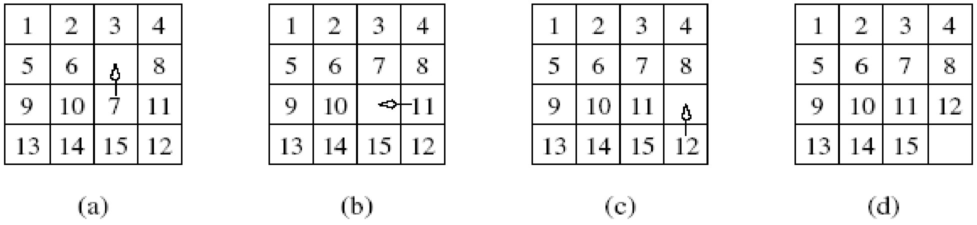
$$AX - XB = C$$

  - The owner of C_{ij} is responsible for the computation of X_{ij}
  - C_{ij} is over-written by X_{ij}

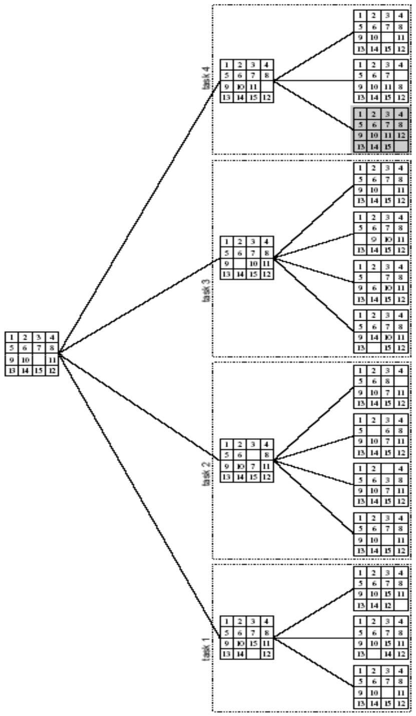


Exploratory decomposition

- Used in decomposition of problem whose solution is computed through a search in a solution space
- Graph problems, game search
- Split the problem area into parts where all results are not ”needed”. The search can be terminated when someone has found a (good enough) solution



Exploratory decomposition

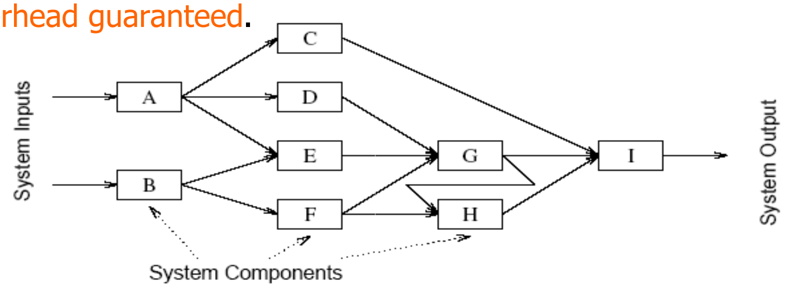


- Uncertain how much better a parallel algorithm will be: too much work may be performed unnecessary compared to serial algorithms
- Speed-up close to *p* only when you take the mean value of all test cases

---

Speculative decomposition

- Start performing calculations even though all inputs are not known. E.g.: start evaluating all alternatives in a branch (if, switch) before the condition for the choice is completely known (computed)
- Only suitable when the input can have a few different values
- Overhead guaranteed.



Speculative decomposition

- Other variants: evaluate only the alternatives in the branch that seems most likely
- Speedup can be significant if there are several levels of speculative decomposition
- Always <*p* sice there is always unnecessary work done
- The difference between exploratory and speculative decomposition:
  - In exploratory it is the output from multiple tasks from a branch that is unknown
  - In speculative it is the input to a branch that leads to multiple tasks that is unknown

---

Hybrid decompositions

- Different decomposition techniques can be combined
- Example: decomposition of matrix computations on a parallel machine with SMP-nodes
  - Data decomposition of input and/or output between the nodes
  - Recursive decomposition of the work between the processors within the respective SMP-node

Tasks

- The decomposition identifies different independent tasks
  - There can still be some interaction
- The tasks are now to be allocated to different processes
- How are tasks created? Statically or dynamically?
  - Static: all tasks known before the execution starts
  - Dynamic: all tasks are not known before the execution starts
  - Dynamically created tasks requires more care about the load balance and creates interaction between processes
  - Allocation is usually made statically for statically generated tasks and dynamically for dynamically generated tasks

---

Tasks

- (Computational) size of tasks?
  - The amount of work needed to finish the task
  - Uniform/non uniform – can influence the load balance
- Do we know the task size?
  - Can be used in the mapping onto processes
- Size of the data belonging to the task?

Interaction between tasks

E.g.: Communication or handling of shared memory

- Static – interaction graph and times known a priori
- Dynamic – interaction graph and times not known a priori
- Regular – interactions follow a given pattern
- Irregular – no given pattern

Interaction between tasks

- Only reading of shared data (read-only)
- Reading and writing of shared data (read-write)
- One-way interaction initiated and completed by one task without any other tasks being involved or interrupted
  - Can be handled by ”shared memory”-paradigms
- Two-way – ”producer and consumer”-scenario
  - The natural model for ”distributed memory”-paradigms
  - Also used in ”shared memory”

Mapping techniques for load balancing

- Given a set of tasks, how do we map these onto processes to minimize overhead?
  - Minimize communication (interaction, synchronization)
  - Minimize idle time
  - These two goals are often in conflict – find an acceptable compromise
- Simplification: mapping techniques – static or dynamic

Static mapping

- Static mapping distributes tasks between processes before execution
- Static mapping is often combined with data decomposition
  - Block distribution, higher dimension generally gives higher parallelism
    - One-dimensional. E.g.: column block mapping of 2-dim array
    - Multi-dimensional. E.g.: mapping of 2-dim array in both row and column block
  - Block-cyclic distribution, many more blocks than processors
    - Good load balance
    - Suitable when different parts of the data generates different amount of work, e.g.: LU
    - 2D is used in ScaLAPACK
  - Cyclic distribution
    - By row or column
    - Perfect load balance but lack of locality can give lower performance

---

Static mapping

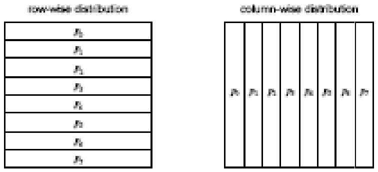


Figure 3.24 Examples of one-dimensional partitioning of an array among eight processes.

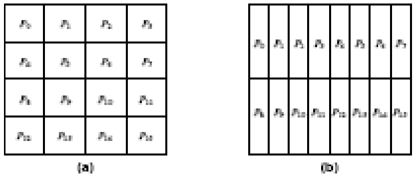
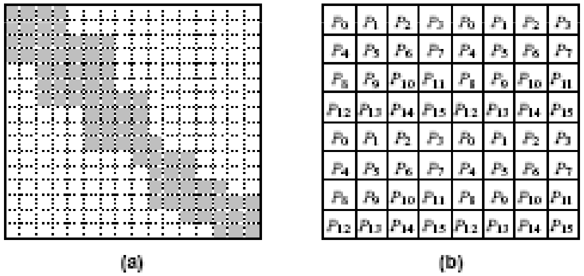


Figure 3.25 Examples of two-dimensional distributions of an array, (a) on a 4 × 4 process grid, and (b) on a 2 × 8 process grid.

Static mapping

- Static mapping can also be combined with random block distribution
  - Many more blocks than processes
  - The blocks are distributed randomly
  - Can be better than e.g. block cyclic on sparse matrices

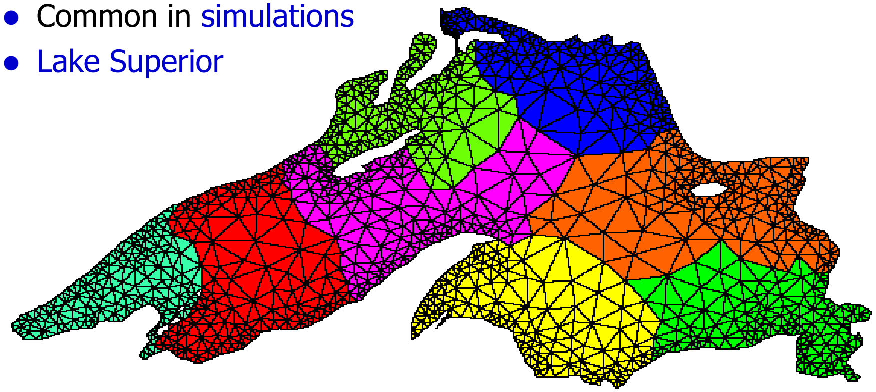


P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}	P_{12}	P_{13}	P_{14}	P_{15}
P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}	P_{12}	P_{13}	P_{14}	P_{15}

Figure 3.31 Using the block-cyclic distribution shown in (b) to distribute the computations performed in array (a) will lead to load imbalances.

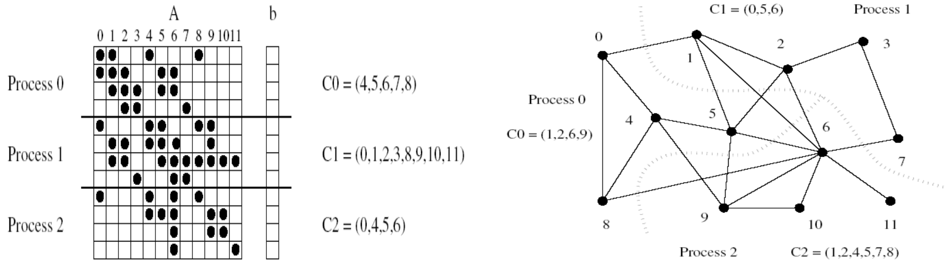
Static dependency graph partitioning

- Split the data into parts such that the contact areas (=communication) are as small as possible
- The contact areas are e.g. determined by a sparse matrix
- Common in simulations
- Lake Superior



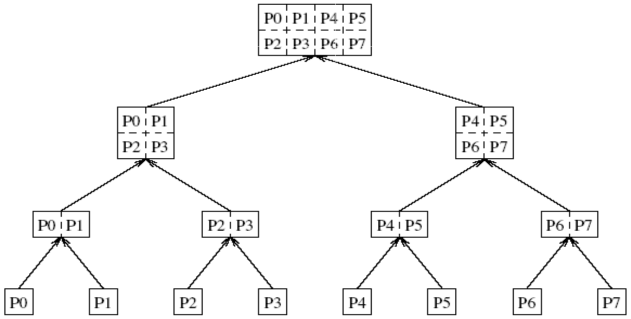
Static task partitioning

- Split the task graph into parts such that the contact areas (=communication) are as small as possible
- E.g.: sparse matrix-vector multiplication



Static hierarchical mapping

- Example: use recursive mapping to build a task graph, and data partitioning to further partition the ”feta” nodes



Dynamic mapping

- Necessary when static mapping results in imbalance of the work load between different processes
- Another name: dynamic load balancing
  - Centralized
    - Master-Slave, centralized work pool
    - Does not scale well: the master processes becomes a bottle neck
    - ”Chunk scheduling” can relieve the pressure
  - Distributed
    - Which pairs of processes shall exchange work?
    - Who takes the initiative? Sender or receiver?
    - How much work will be sent in each communication?
    - When shall work be exchanged? When there is no more work? When you want to get rid of some?
- Can be implemented in most paradigms

Methods for hiding interaction

- Maximize data locality
  - Minimize total amount of data-exchange
    - Select appropriate decomposition and mapping
    - E.g.: distributions of higher dimensions often better
  - Minimize frequency of data-exchange
    - Restructuring of the parallel algorithm may be necessary
    - Communicate larger chunks of data on fewer occasions
- Minimize bottle necks
  - Contension
  - Communication
  - Redesign of the parallel algorithm such that all computations needed for a block are being done at the same time

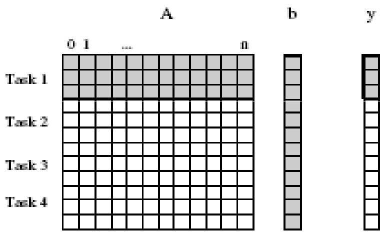
---

Methods to hide interaction

- Overlap computation with communication
  - Very common in numerical computations
  - Disadvantage: decreases the granularity of tasks
  - Demands support of lower layer software and/or hardware
    - Distributed memory – compute and communicate at the same time
    - Shared memory – data prefetching
- Use collective communication operations
  - Broadcast, scatter, reduce, gather, all-to-all
  - Often highly optimized implementations, e.g. *recursive doubling*
- Overlap interactions with other interactions

Methods to hide interaction

- Replicate data
  - Demands extra memory
  - E.g.: matrix-vector y = Ab, all processes have the whole vector b

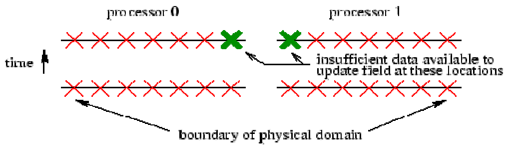


- Perform extra computations (redundant computing)
  - E.g.: ghost cells in compute mesh for time dependant PDE,
  - Need to communicate only every other time step

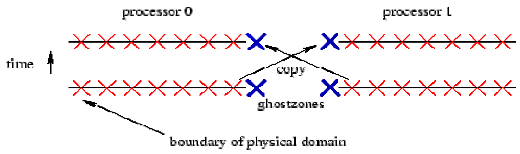
Methods to hide interaction, example

- To update values on an inner border you need values from other processors
- You can exchange O(n) data points in each time step...
- You can also choose to exchange O(2n) data points every other time step and compute the O(n) values needed for the next step on both sides of the border
- The extra data is saved in "ghost zones". Saves communication – performs extra work (often negligible if the decomposition is coarse grained)

Distributed wave equation with no ghostzones



Distributed wave equation with ghostzones



Parallel algorithm models

- Data-parallel model
  - Identical operations performed in parallel on large data sets
  - Data- partitioning, static mapping, regular interactions
  - E.g.: apply a compute stencil for PDE over a discretized area
  - Large problems can be solved efficiently
- Task-parallelism-model
  - Partitioning of dependency graph
  - Static/dynamic partitioning and mapping
  - E.g.: compile independent subroutines, call independent subroutines
- Work pool model
  - Dynamic mapping for good load balance
  - Centralized or decentralized (see earlier)
  - E.g.: solve game- or graph problems by searching in the solution space

---

Parallel algorithm models

- Master-slave
  - One process generates work and distribute to workers
  - Task may be generated and distributed statically beforehand if it is possible
  - The master can of course became a bottle neck
  - E.g.: distribution of independent iterations of a loop on an SMP-machine
- Pipeline
  - Stream of data passes through different processes
  - Different processes do different things with the data in parallel
  - Chain of producer-consumer
  - E.g.: Parallel LU-factorization
  - See also the *case studie*
- Hybrid models
  - Different models are applied hierarchically or sequentially on one problem

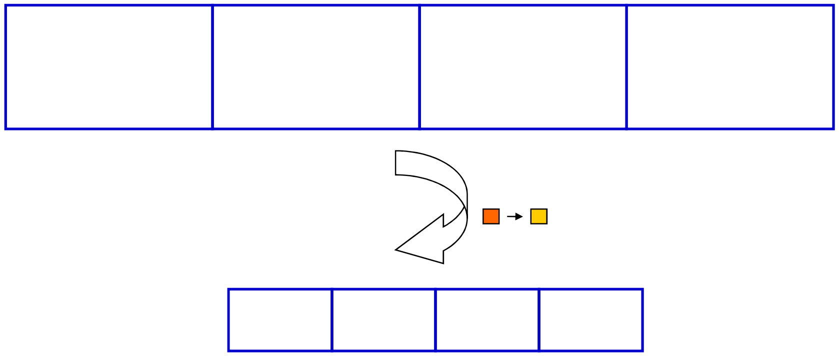
---

Case study: MPEG-2 to MPEG-1 transcoding

To be performed:

- An MPEG-2-stream shall be transformed into an MPEG-1-stream
- Certain correction of the colors should be done
- The MPEG-1-resolution is *half of* the MPEG-2-resolution
  - From full TV-resolution to ”junk video”
- Both in and out-format is ”long-GOP” (Group Of Pictures)
  - Simplified:
  - Every 12th frame is a reference frame (I)
  - All other frames are constructed by difference from previous I-frame (P-frames, significantly less than I-frames, P = prediction)
  - In real life there are also B-frames: differences that points backwards to the closest I or P or forward to the closest I or P, B = bi-directional. B-frames makes it possible to even more compression
- Typical storage of the stream: IBBPBBPBBPBBIBBPBBPBBPBBI...

Case study: MPEG-2 to MPEG-1 transcoding



Case study: MPEG-2 to MPEG-1 transcoding

Decomposition:

- GOP
- Frame
- Block
- Hierarchical
- Task
- ...?

Case study: MPEG-2 to MPEG-1 transcoding

- Idea to solution:
  - Pipeline: all frames streams through the pipeline where different processers have responsibility for different steps
    - Unpacking – construct the picture from I-, P- and B-frames
    - Rescaling
    - Color correction
    - +some other operations (e.g. convert/mix sound)
    - Repacking – pull the picture apart into I-, P- and B-frames
  - Possibly use several processes in one of the steps if it is too costly compared to the other steps
    - Frames are split up block wise in rectangles (1-dim data partitioning)
  - Task-parallelism
  - Suitable for SMP or multi(dual/quad)-core machines
    - Support for both heavy processes (fork) and light weight threads

---

Summary – check box

Decomposition	Recursion	Data	Exploratory	Speculative
Tasks	How create	Size	Knowledge about size	Size of data
Interaction between tasks	Static/ dynamic	Regular/ irregular	Reading/ writing	Sending/ exchange
Mapping	Static:	Array/block/ cyclic/random	Graph	Task/ hierarchical
	Dynamic:	Centralized	Decentralized	
Deal with overhead	Maximize data locality	Minimize bottle necks	Replicate data/comp.	Overlap/ group comm.
Model	Data parallel	Task graph	Work pool/ master-slave	Pipeline