# A Generic Software-Framework for Distributed, High-Performance Processing of Multiview Video

Dirk Farin and P.H.N. de With
d.s.farin@tue.nl

University Eindhoven (TU/e)
Video Coding & Architectures Group

LG 0.10, P.O. Box 513
5600 MB Eindhoven
The Netherlands

# Presentation Outline

- Why parallel processing?
- Design criteria for a software framework.
- Description of our proposed framework.
- Example application.
- Conclusions

# Whether to Do Parallel Processing or Not

- Why parallel processing?
  - High complexity of many computer-vision algorithms,
  - hardware limitations
    (e.g., maximum number of camera ports per computer),
  - CPU speed will not increase significantly in the future.
    Instead, we will have multi-core CPUs in the future.

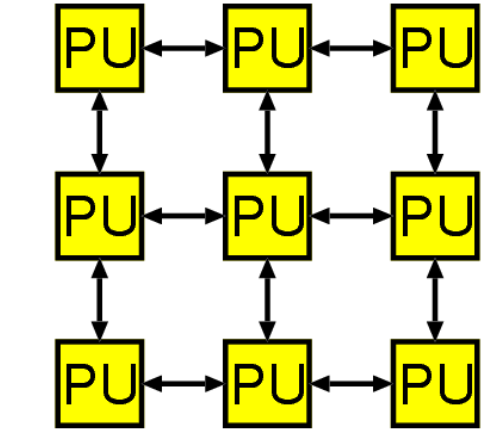
- Why **no** parallel processing?
  - More complicated software development,
  - special algorithms are complex,
  - developers have to understand both:
    *computer vision* and *parallel processing*.

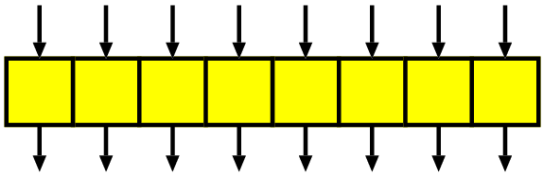
# Generic Framework for Parallel Processing

- Observation:
  - Research on parallel processing concentrates on finding parallelized algorithms.
  - These are hard to understand and complex to implement.
- However:
  - Practical applications often consist of a set of processing steps.
  - Better approach would be to keep algorithms simple (non-parallelized), but distribute the set of algorithms.
- Proposal:
  - A framework into which existing algorithms can be integrated easily, and
  - which enables to run the software system on a cluster.

# Design Considerations 1/5

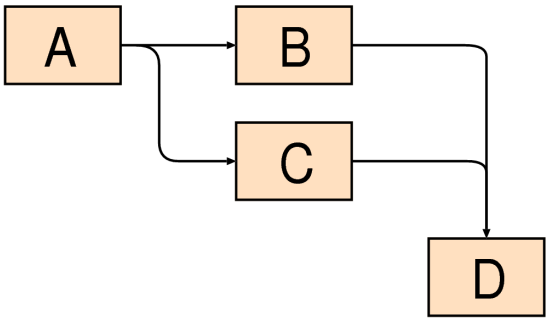
- Fine-grained parallelization
  - data parallelism
  - aims at accelerating a single algorithm
  - SIMD or massive-parallel processors
  - compiler support or low-level programming (e.g., assembler or GPU shaders)



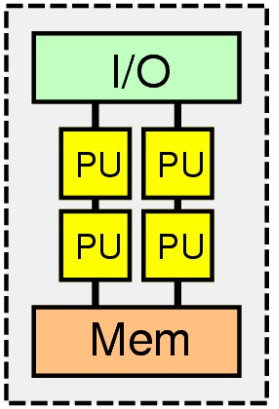
*vs.*

- Coarse-grained parallelization
  - task-level parallelism
  - aims at accelerating a complex system
  - computing clusters or multi-core processors



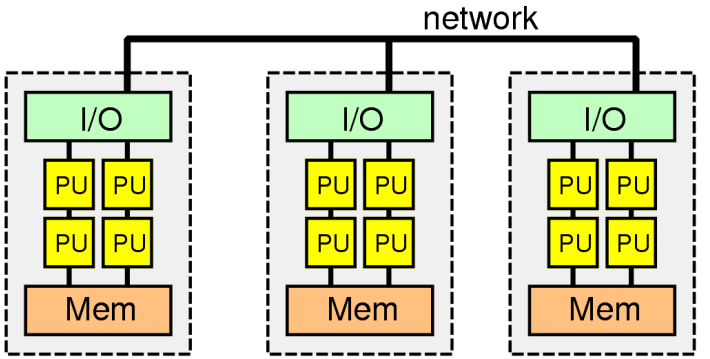
# Design Considerations 2/5

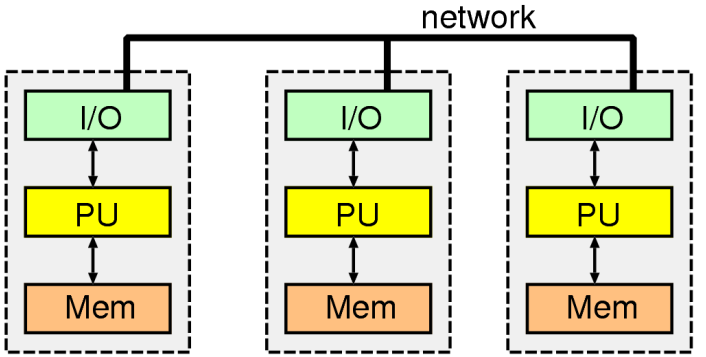
- Symmetric Multi-Processing (SMP)
  - shared memory, no explicit data-transfer
  - memory bandwidth and IO are limited
  - number of cores is limited




*vs.*

- Cluster of computation nodes
  - explicit data-transfer over network
  - limited network bandwidth
  - processing delay because of data-transfer



# Design Considerations 3/5

- Automatic parallelization
  - computer decides which tasks are computed on which node
  - knowledge about computational behaviour required (difficult to define)
  - further constraints about special hardware, IO connections

*vs.*

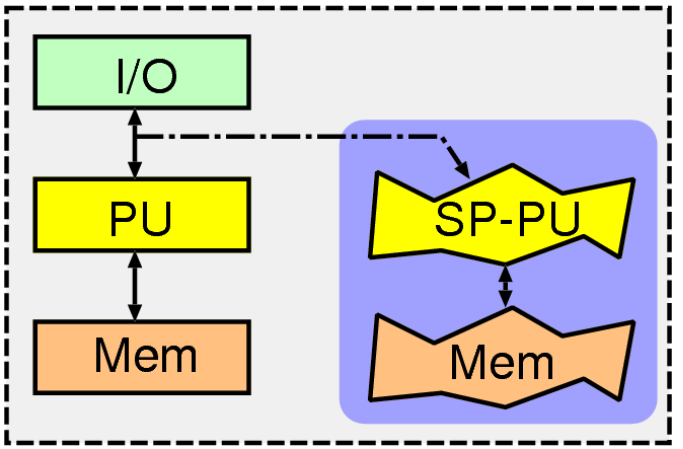
- Manual splitting of tasks
  - complex on large systems with complicated data-flow
  - practically not so difficult if
    - systems are static (no dynamic change of processing graph),
    - computation behaviour is easy to understand intuitively,
  - easier than to describe resource constraints formally.

# Design Considerations 4/5

- Architectural flexibility
  - Avoid to impose a fixed processing architecture, like
    - pipelining
      (severely limits applications), or
    - data-parallel processing
      (introduces delay and does not allow iterative algorithms).
  - Strengthen reuse of algorithms between applications
    - allow for a varying number of inputs (e.g., for multi-view)
    - allow for special, optional inputs (e.g., algorithm hints)
    - allow for special, optional outputs (e.g., for visualization)

# Design Considerations 5/5

- Special-purpose processor utilization
  - Low-level vision algorithms can be implemented efficiently on special hardware (FPGAs or GPUs).
  - Special hardware often comes with its own memory.
  - Image-transfers between memories are costly and should be minimized.




- Problem for the framework:
  it has to manage image data that is not in main memory.
- Data transfers between cluster nodes gets more complex and inefficient.

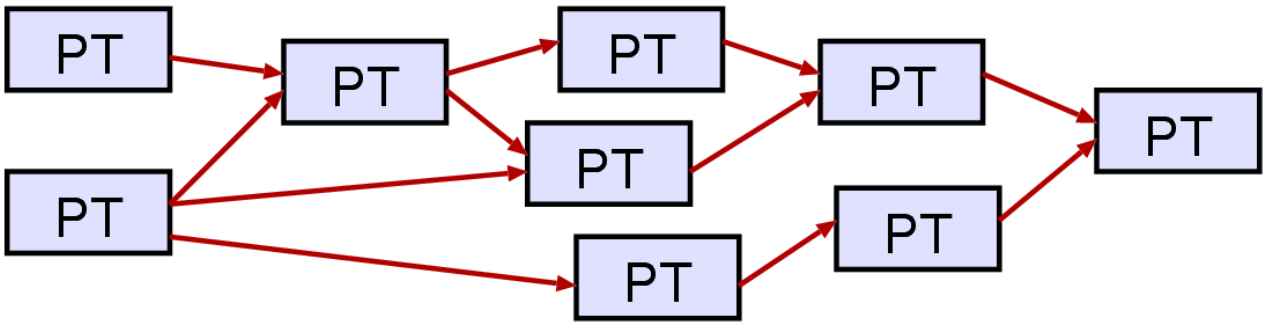
# Framework Support for Different Levels of Parallelization

- Four levels of parallelization

| | |
|---|---|
| • distribution over cluster of computers | enabled by our framework in a unified approach |
| • SMP parallelization | |
| • using specialized co-processors (e.g., GPUs) | supported, but not transparently |
| • SIMD via processor support | applicable, but no special support |

# Framework Overview 1/2

- At the lowest level, we have:
  - Processing Tiles (PTs)
    - implement an algorithm,
    - operating on a set of input data, and
    - generating a set of output data.
  - Data is stored in Data-Unit objects.
    - arbitrary data-type definitions are possible
  - PTs are connected to graphs,
    Data-Units are passed over the graph edges.



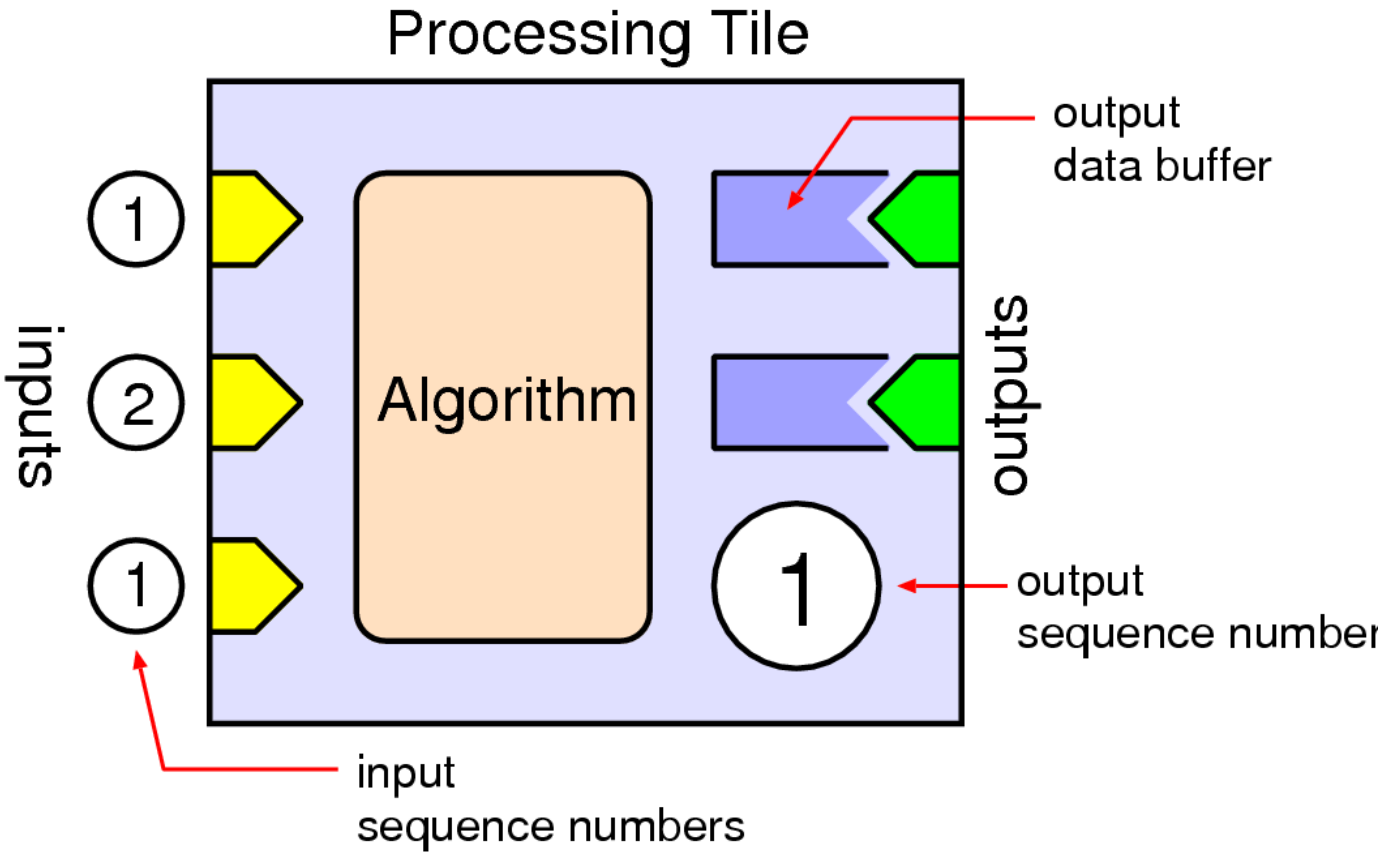
# Framework Overview 2/2

- On the next level: *Processing Graph Control* (PGC)
  - adds a scheduler to a graph of PTs,
  - uses pool-of-tasks concept to process the tiles with a set of threads.

- On the highest level: *Distributed Processing Graph* (DPG)
  - wraps PGC into a network interface,
  - nodes in the graphs of different PGCs can be connected, data-transfer on these edges is transparently realized as network transfer,
  - for the user, the network layer is transparent (but PTs can be realized on specific nodes if requested).

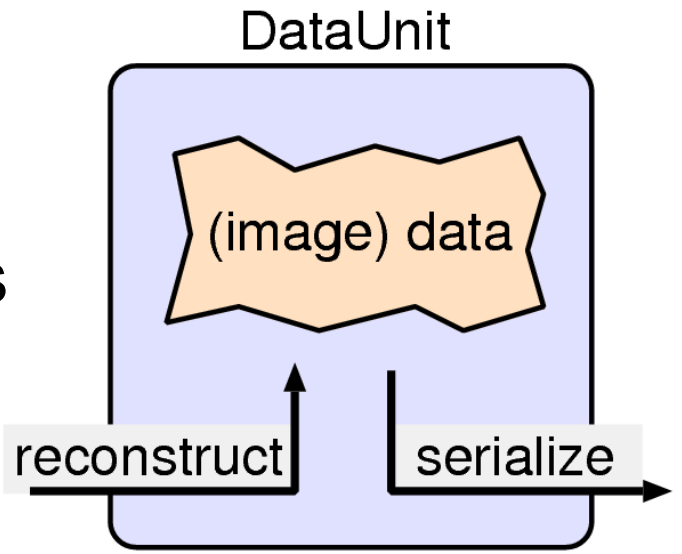
# Processing Tiles

- Algorithm takes input data from input plugs (no buffers).
- Output is stored in output buffers.
- Processing can only take place if *sequence numbers* of all inputs are equal.

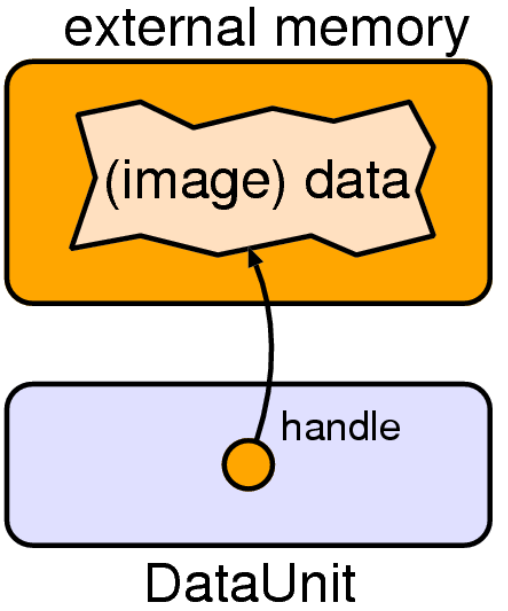


# Data Unit

- Encapsulates the image data.
- serialize() and reconstruct() methods transform to/from byte-stream representation.

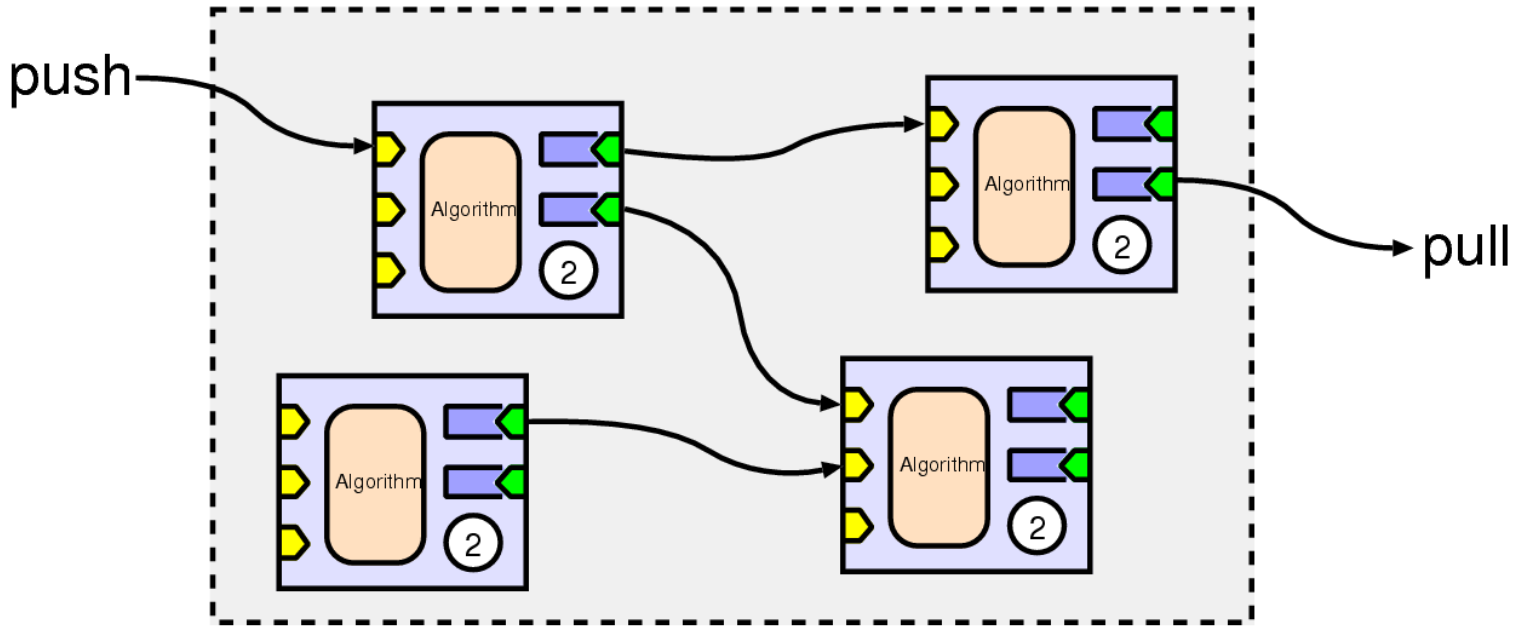



- Data that is stored in external memory (gfx-card, FPGA-board) is managed by DataUnit.
- Between PTs, only the handle to that data is transfered.
  No data copying is required.
- Data conversion with special PTs.



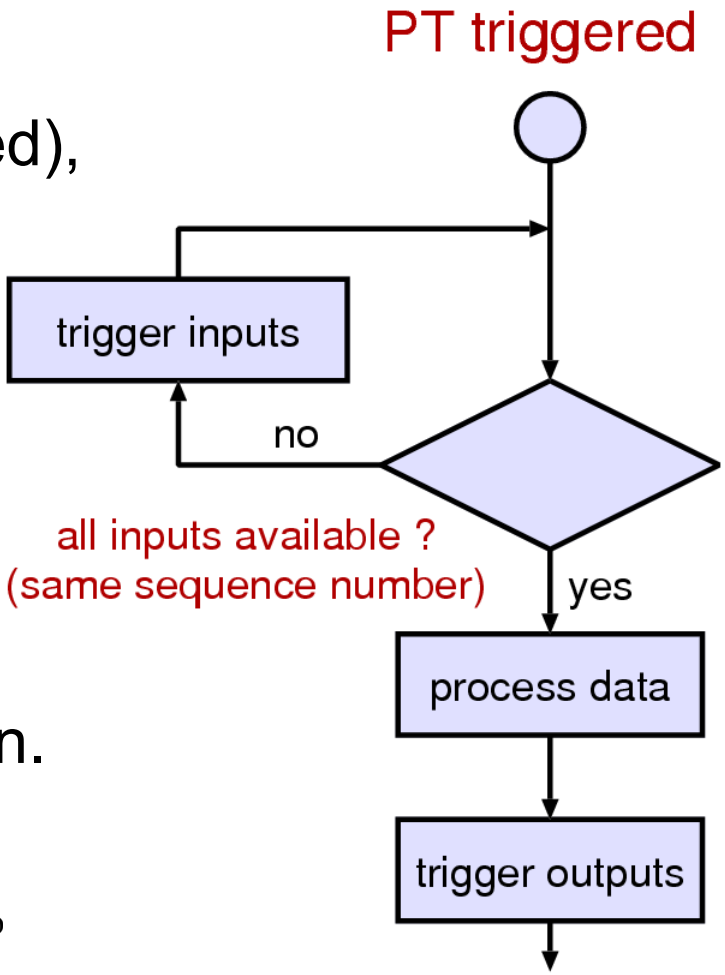
# Using the Framework on the Tile Level

- Each tile object represents one algorithm.
- Tiles can be connected and used without control object.
  (Each output can be connected to several inputs.)
- Processing is triggered by
  - pushing new data into the network, or by
  - pulling more data out of the network.

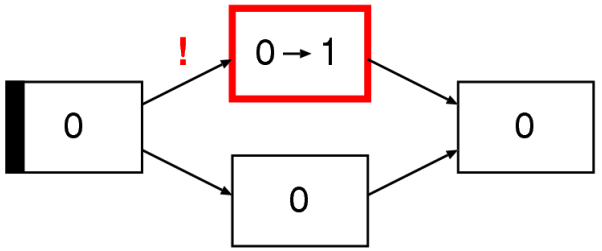


# Direct Tile-Interaction: Processing Model

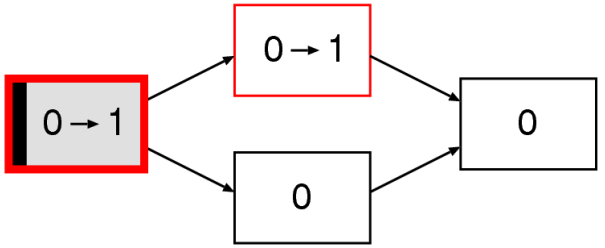
- When a PT is triggered, the following steps are carried out:
  - check if inputs are available (their *sequence numbers* should be the *current number +1*),
  - trigger inputs that are not up-to-date,
  - process the data (data for unconnected output-pins does not have to be generated),
  - trigger outputs to continue processing.
- With this process, **push** and **pull** semantics can both be applied.
- However:
  this usage does not lead to parallel execution.



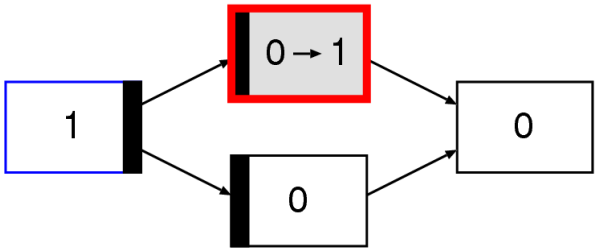
# Processing Example



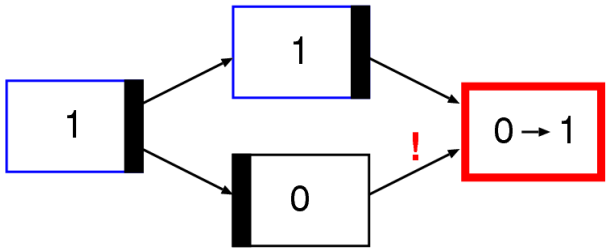

marked PT is triggered
input data is not available



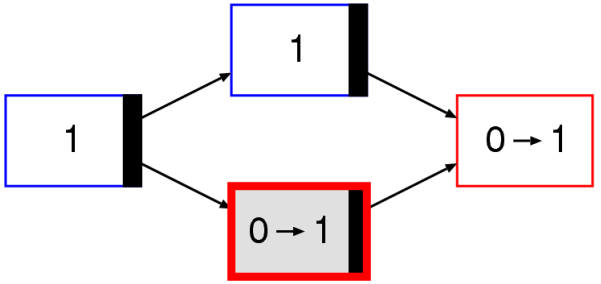

control is first given to
predecessor to generate input



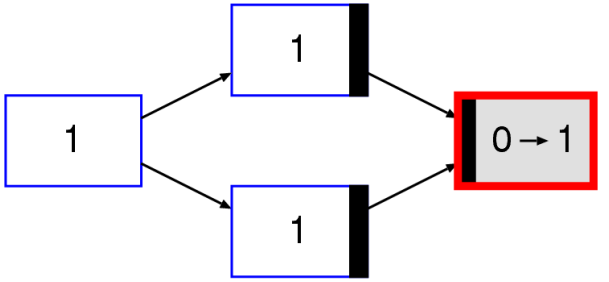

now the tile can
be processed




the output of the top tile is
triggered, but that PT needs
another input first




the predecessor can be directly
processed, because the input
is already available



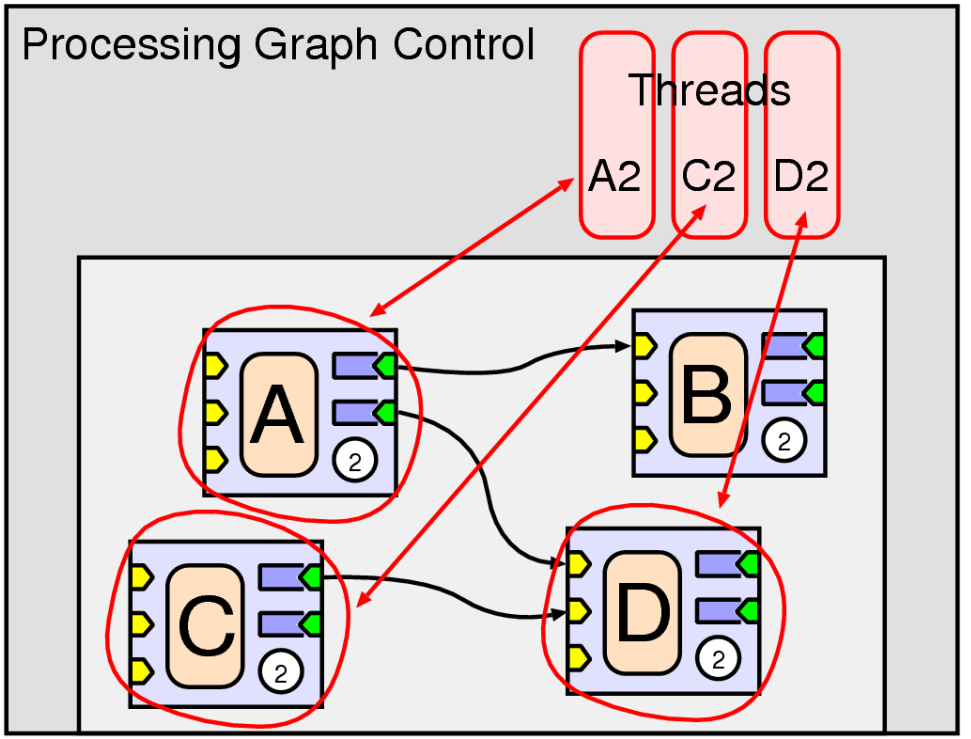

now the final output
can be generated

# Adding a *Processing Graph Control*

- *Processing Graph Control* (PGC) manages tiles internally.
- A set of worker threads are used to process data.
- Scheduling is automatic, no external triggering required.

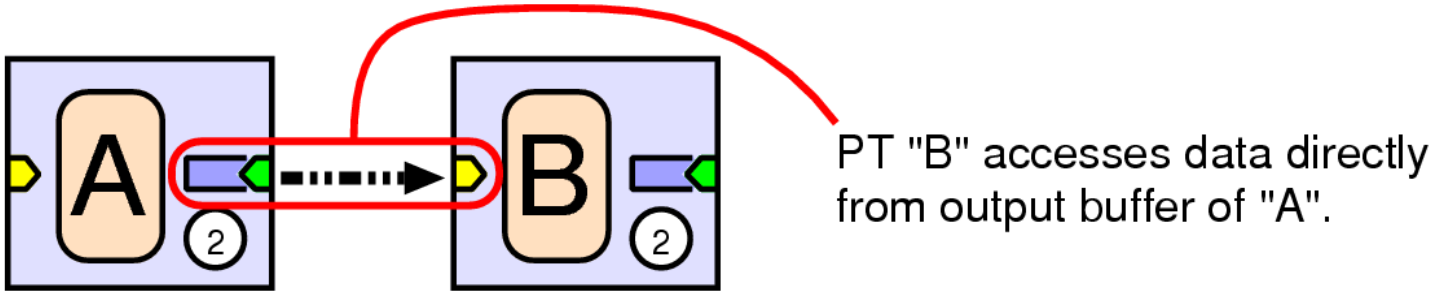
- <u>Scheduling algorithm:</u>
  - Generate tasks, being pairs of (*PT*, *sequence num.*)
  - Tasks with
    - inputs available and
    - no outputs depending on it

  are put into a *ready-to-run* set.
  - Free threads get new tasks from this set.

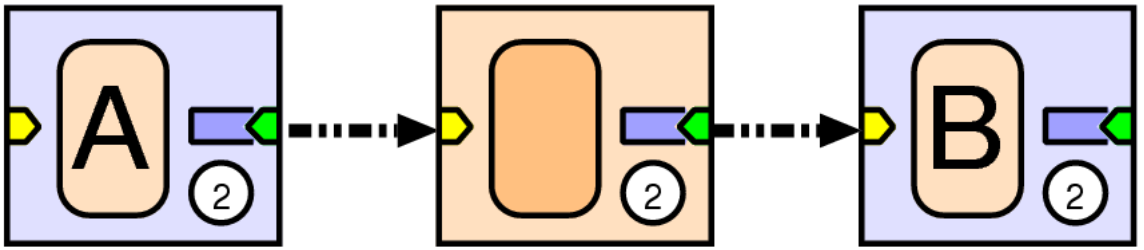


# Scheduling Properties

- Directly successive tasks cannot be executed in parallel, because output data-buffer of first PT is reused as input data-buffer of successive PTs.
  - Advantage: no unnecessary data-copies

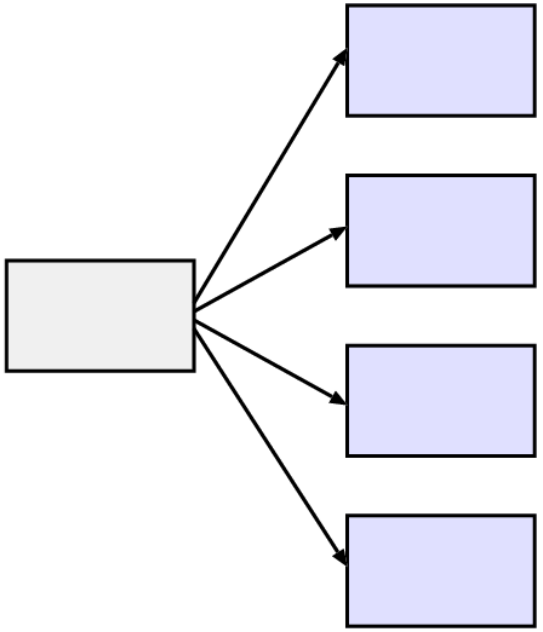



  - To enable concurrent processing of both PTs, a buffering-PT can be inserted between the two.



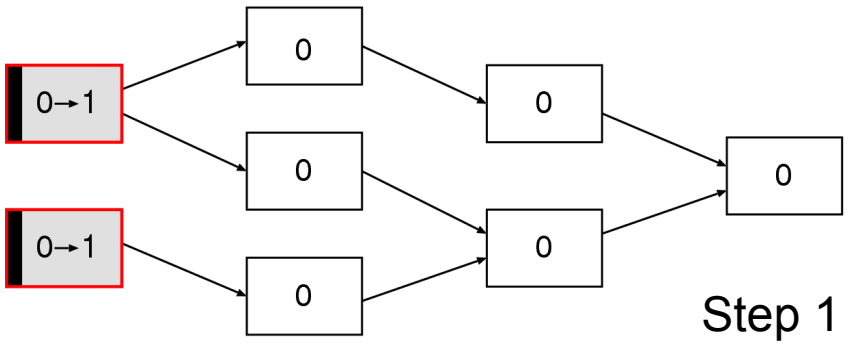
# Scheduling Properties

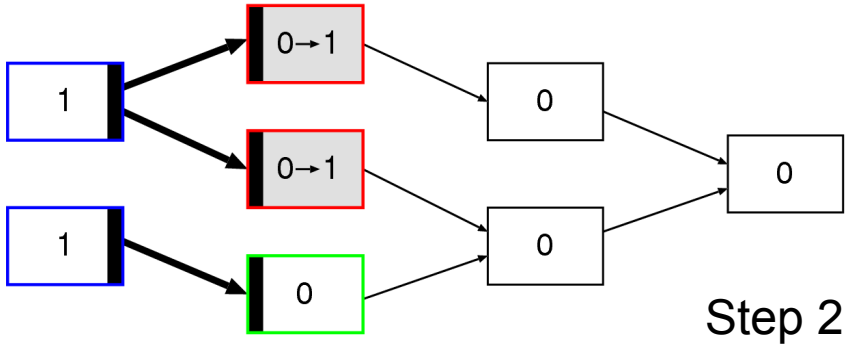
- Scheduling automatically supports both types of parallelism:
  - concurrency of independent tasks

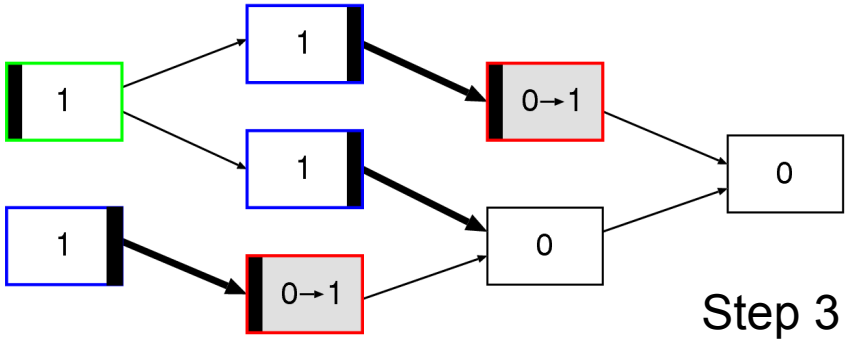


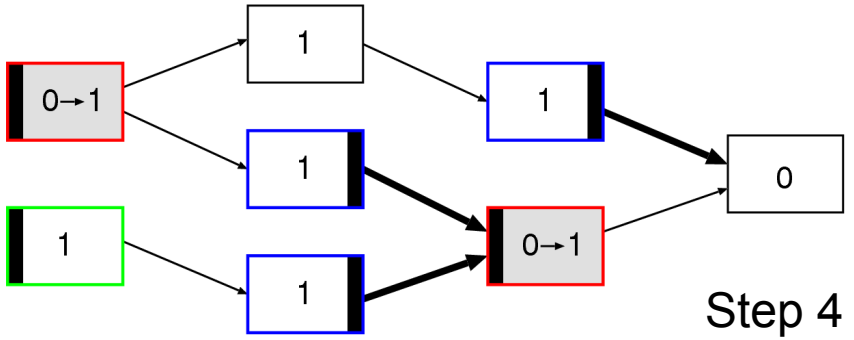
  - pipelining of temporally successive frames

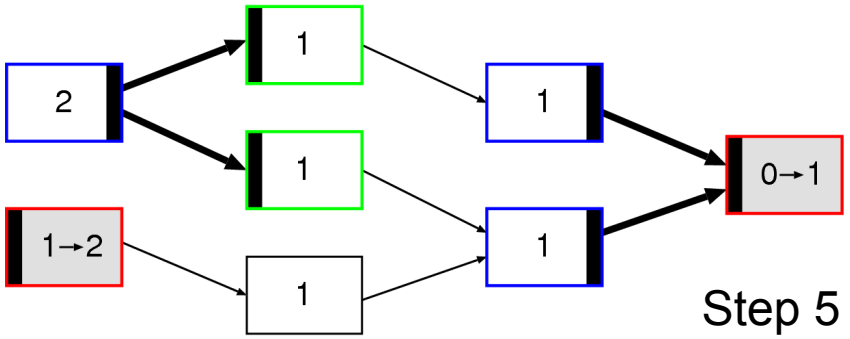
# Scheduling Example

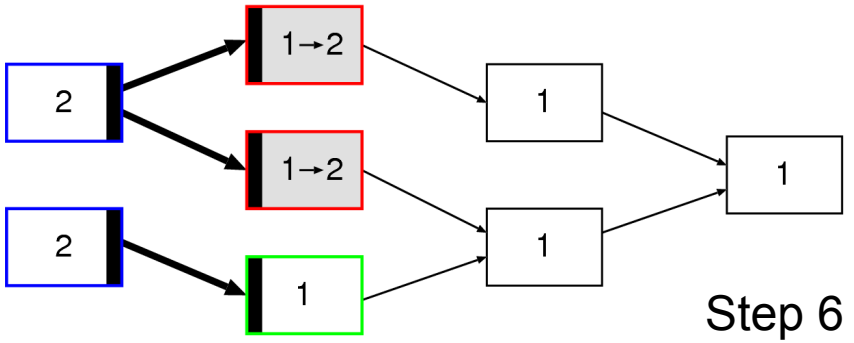


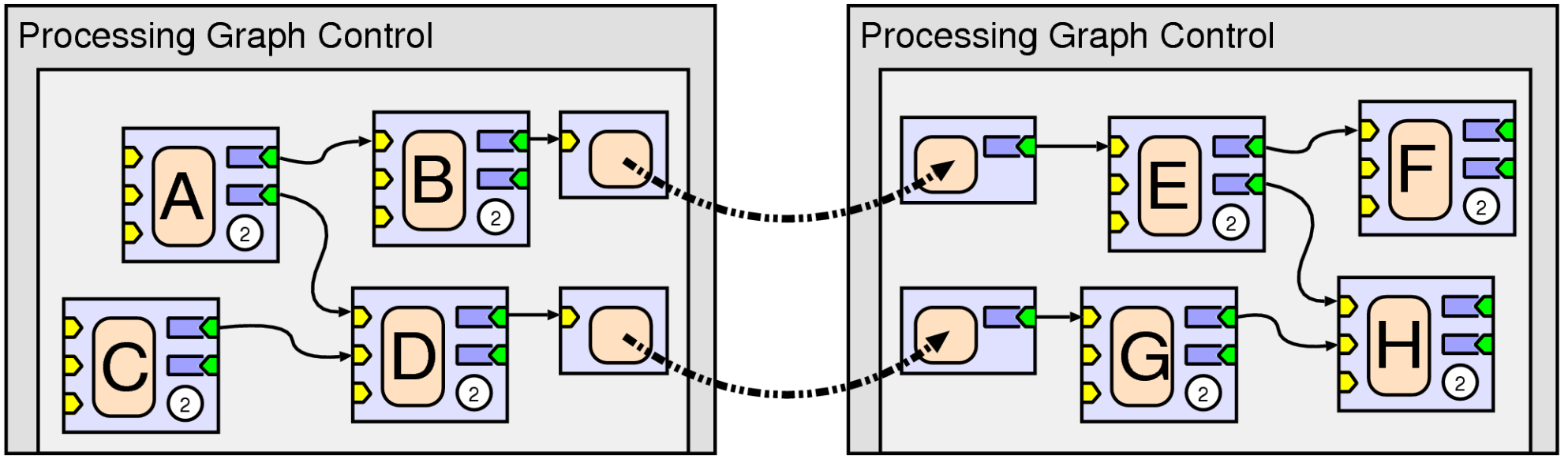




being processed / ready to run / blocked, because output it still required

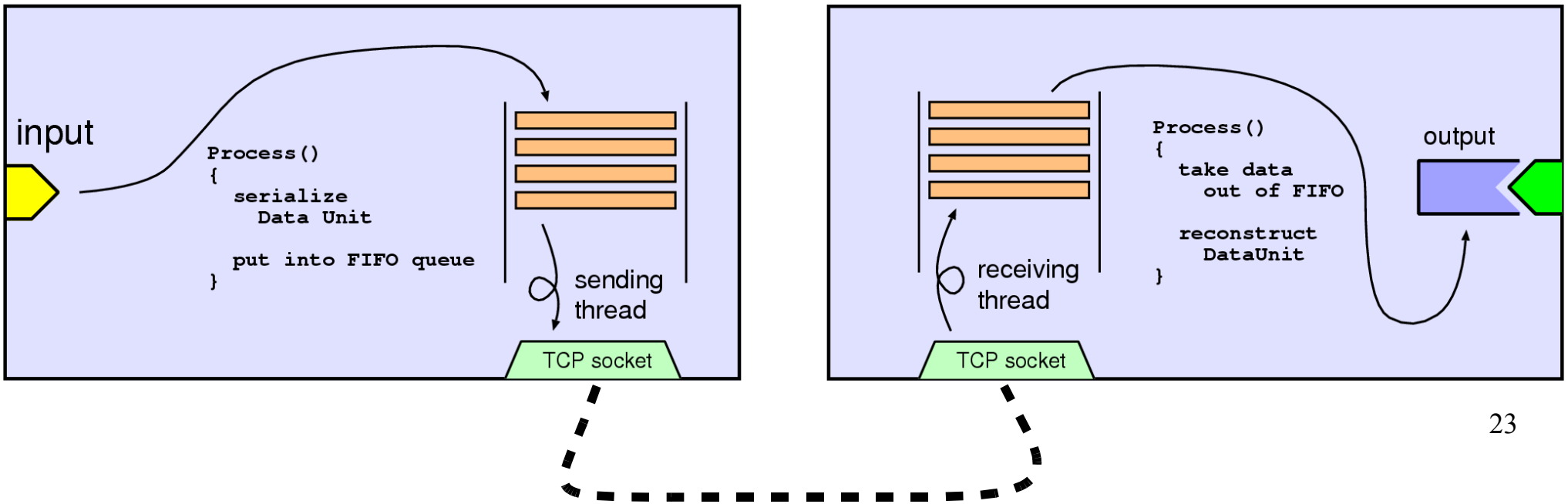
# *Distributed Processing Graphs* (DPG)

- Adds another level above the PGC.
- PGCs on several computers act transparently as a single PGC.
- DPGs are internally organized as a tree structure to distribute the control commands.
- To connect tiles on different nodes, network transmission and reception PTs are added transparently.



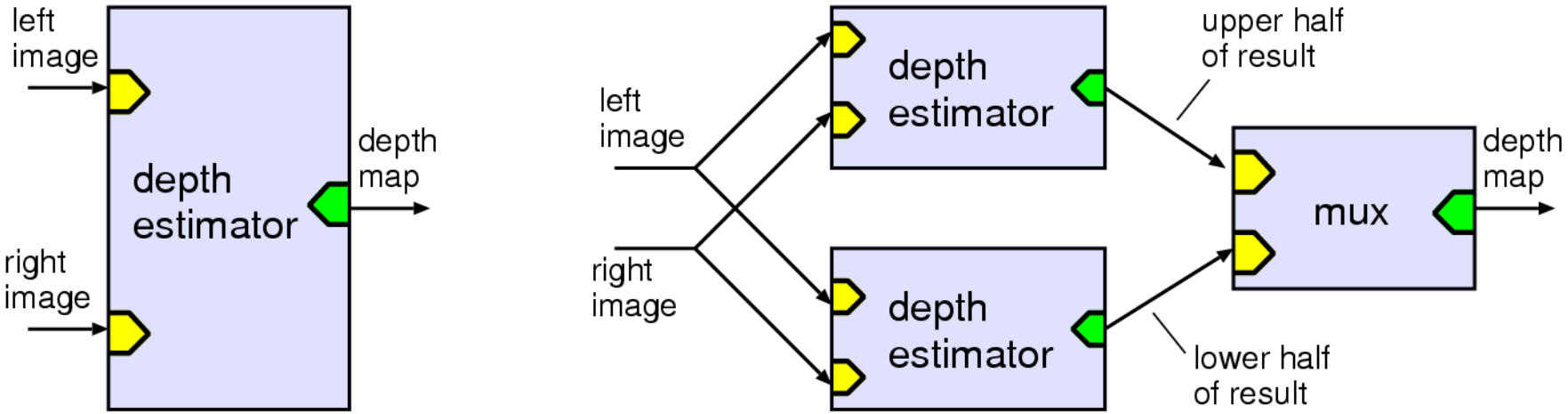
# Network-Transmission PTs

- Network-transmission PT
  - uses DataUnit serialization feature to generate byte-streams,
  - byte-streams are sent through TCP in separate thread.
- Network-receiver PT
  - receives data in a separate thread,
  - reconstructs data from byte-stream.



# Implementing Parallel Algorithms

- Framework can also be used for parallel algorithms:
  - Split the algorithm into separate PTs, each of which computes part of the result.
  - Use a PT that combines all intermediate results.
  - Optionally, a special DataUnit type can be defined for intermediate data.
  - Limitation: communication between PTs is not possible.



# Example Application: Multi-View Video

captured input images

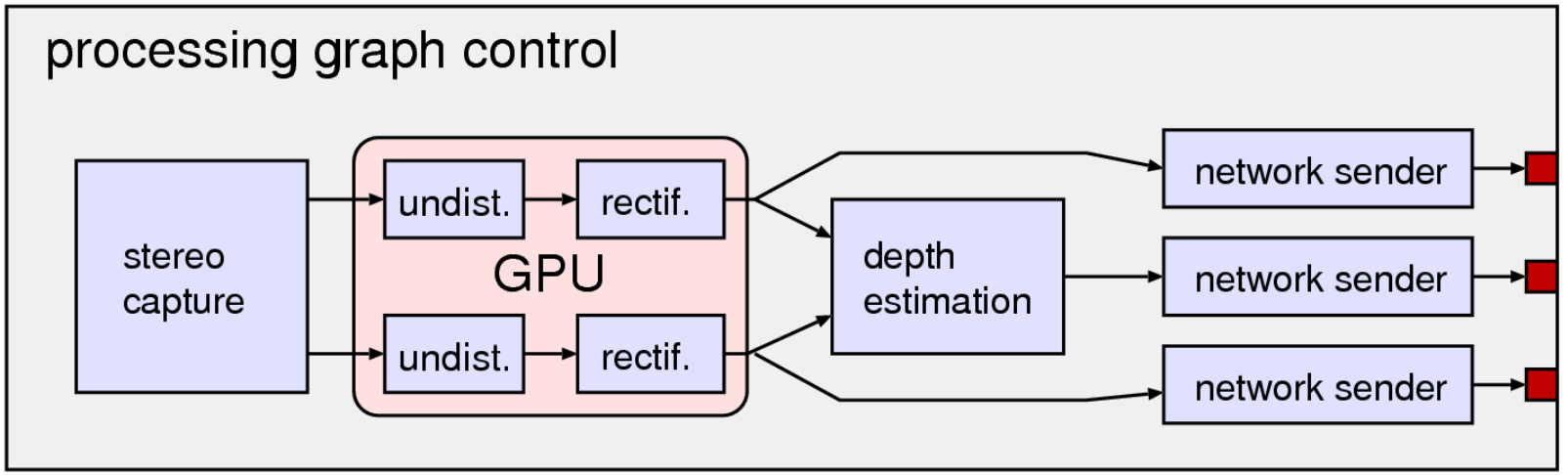





interpolated images

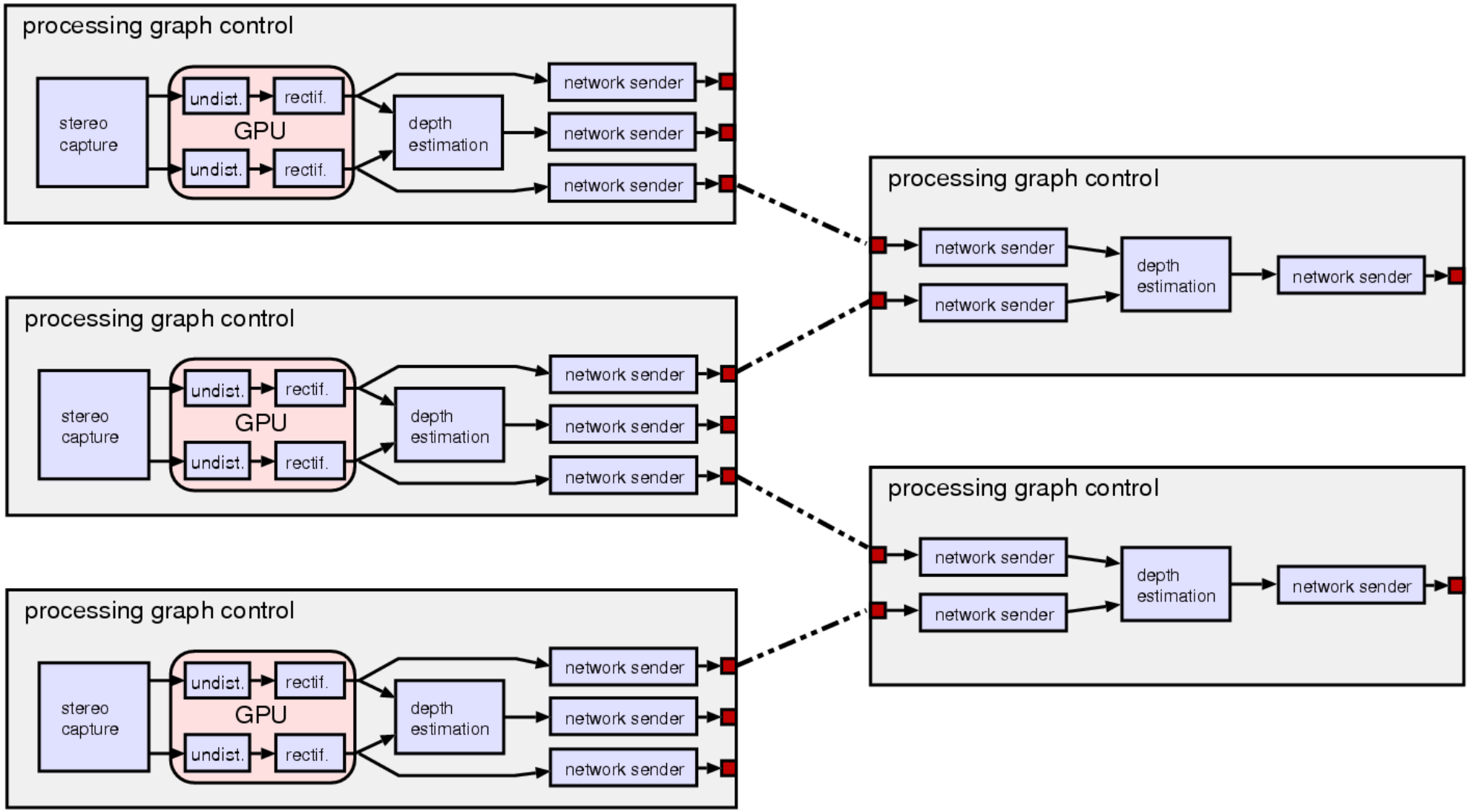
# Example Application: Multi-View Video

- Processing steps:
  1) capture from a set of cameras (max. 2 cameras per computer),
  2) correct lens distortion on each captured image (geometrical transformation, best performed on GPU),
  3) rectify image to canonical stereo (horizontal epipolar lines) (geometrical transformation, also on GPU),
  4) estimate depth image from stereo (most complex part),
  5) interpolate views between image pairs.



# Example Application: Multi-View Video

- Computation in cluster of computers (here: 6 cameras)



# Conclusions

- Generic framework for parallelization on *cluster of SMP computers*.
- Easy integration of existing algorithms.
  - Just wrap each algorithm into a Processing Tile.
  - Algorithm development stays simple.
  - Parallelization is independent from algorithm development.
- Framework can be used at three different levels:
  1) direct connection of PTs without parallelization
  2) apply SMP scheduler
  3) distribute in cluster
- Framework can also be used to implement parallel algorithms.