Master Program (Laurea Magistrale) in Computer Science and Networking

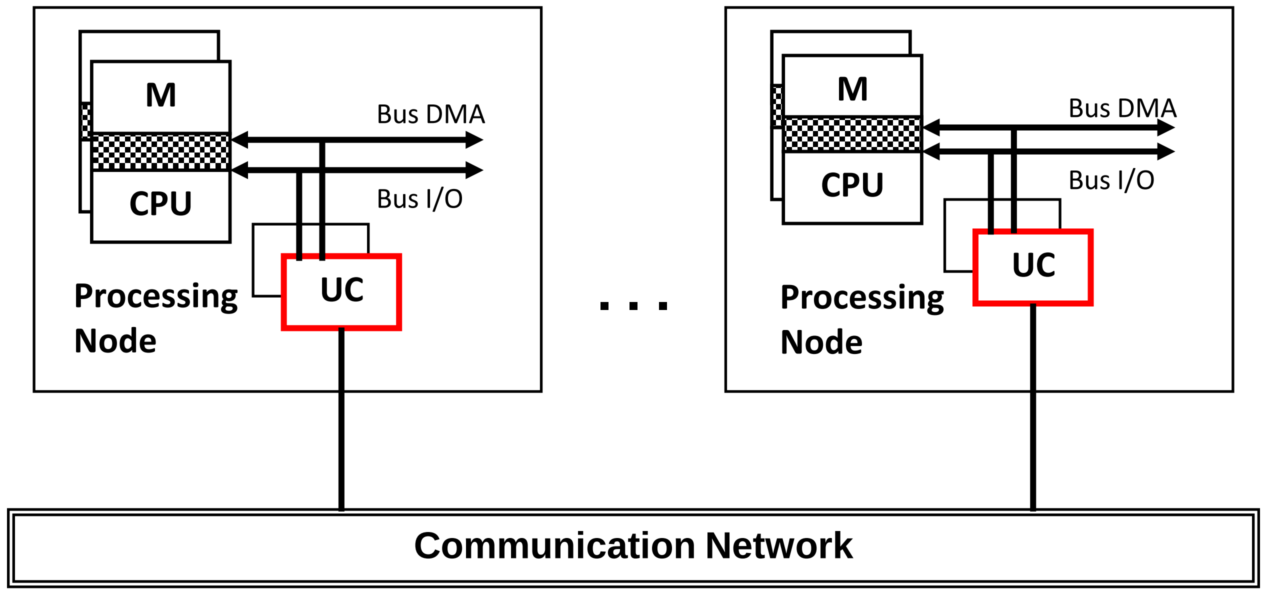
High Performance Computing Systems and Enabling Platforms

Marco Vanneschi

5. Distributed Memory Architectures

Distributed memory architectures

- Processing nodes are not able to share a physical memory space
  - a node cannot address the memory of another node
- I/O is the only primitive mechanism for node cooperation
  - cooperation by explicit value exchange
  - possibly, shared memory can be emulated



I/O unit(s) dedicated to node interfacing (UC):
Communication Unit,
Network Interface Unit,
Network Card,
…
Any architecture for Processing Nodes, e.g. multiprocessor

Kinds of distributed memory architectures

- PC/Workstation **Cluster**
- Multicluster
- Massively Parallel Processor (MPP)
- …
- Grid
- Data Center
- Server Farm
- Cloud
- …

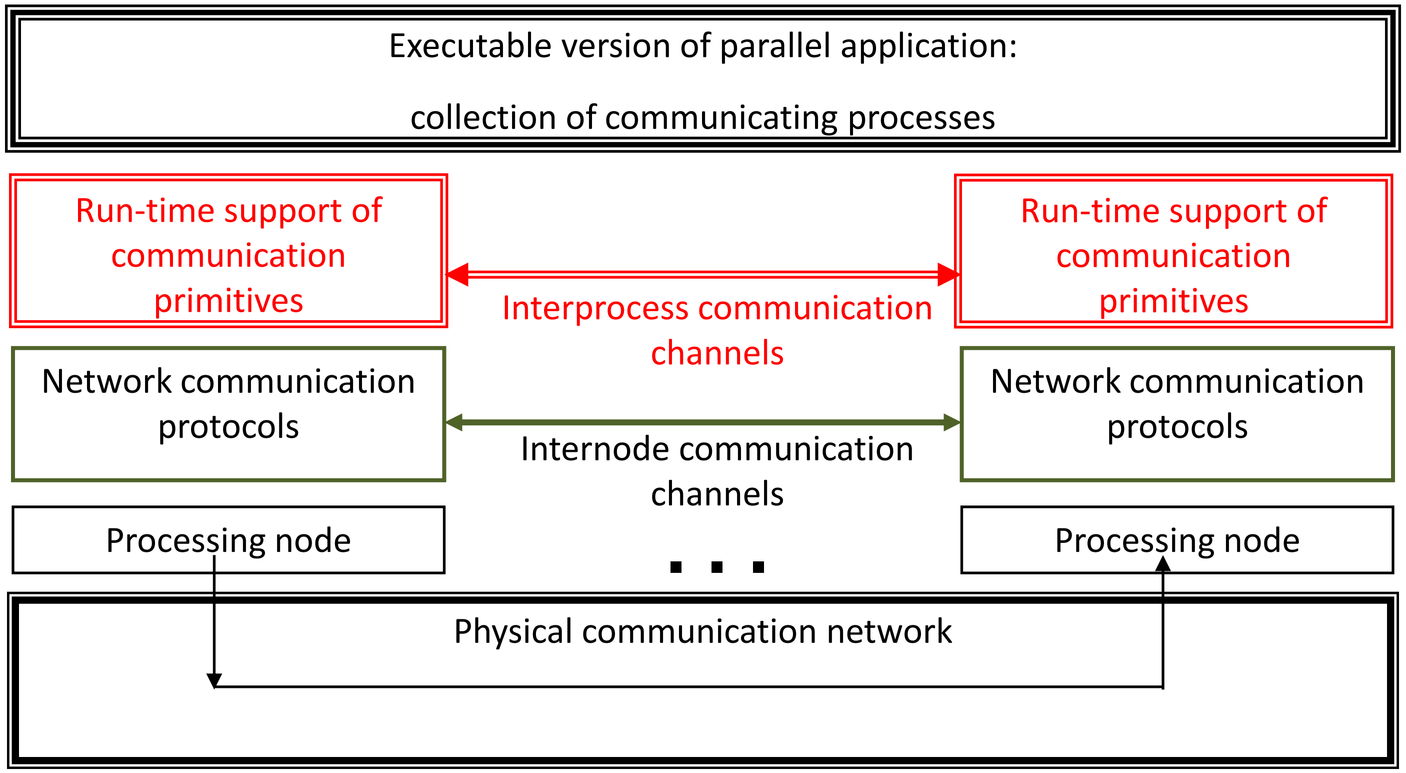
Dedicated processors architectures:

static allocation of processes to processing nodes,

possibly, dynamic reconfiguration

for load balancing or fault-tolerance reasons.

Interprocess communication support


Executable version of parallel application:
collection of communicating processes
Run-time support of communication primitives
Interprocess communication channels
Run-time support of communication primitives
Network communication protocols
Internode communication channels
Network communication protocols
Processing node
. . .
Processing node
Physical communication network

Run-time support exploits

- network communication protocols
- architectural features internal to processing nodes (notably, I/O mechanisms via shared memory: DMA and/or Memory Mapped I/O)

Communication networks

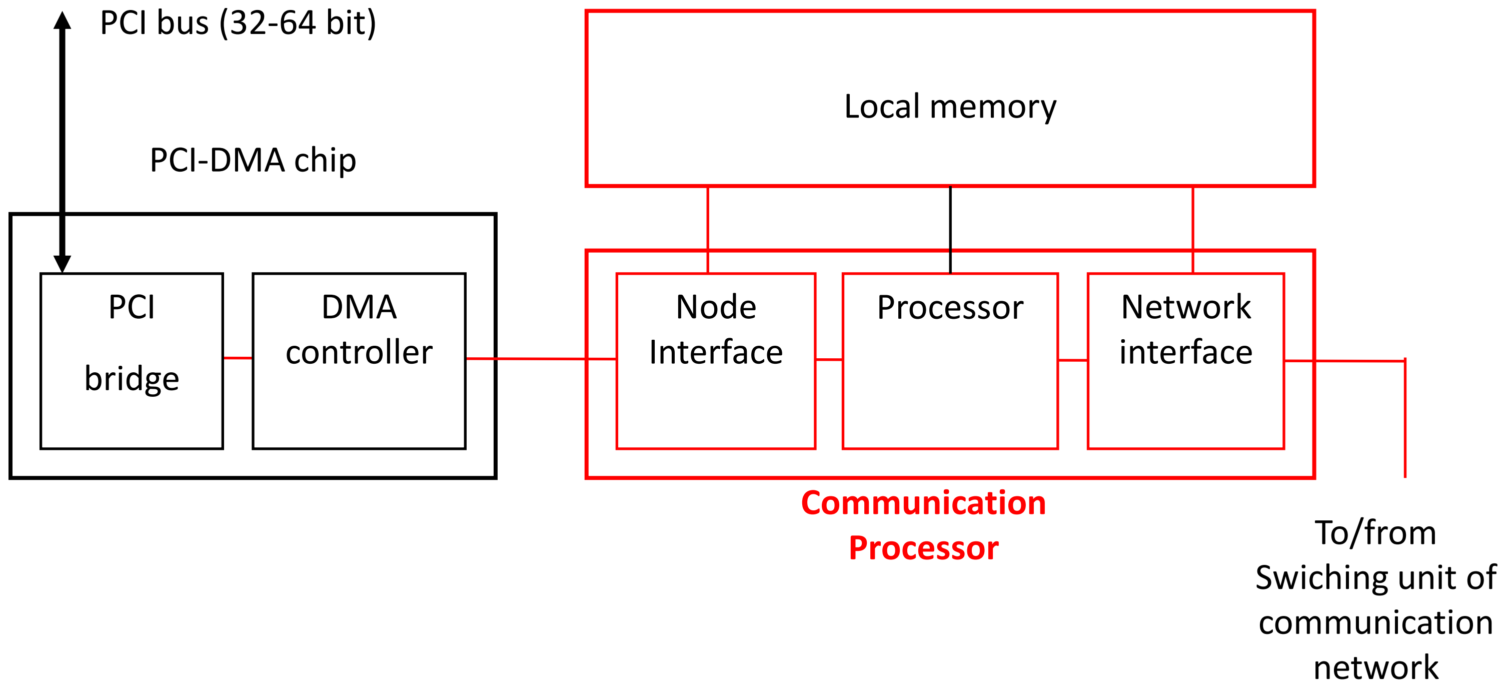
- Simple cases of network computers: usual network architectures (LAN / MAN /WAN) with serial links and standard IP protocol

- High performance architecutures: very local interconnection network (“Switch”) according to the structures studied for Shared Memory Architectures:
  - **multistage Fat Tree, Generalized Fat Tree**
  - **low dimensione cubes**
  - in the most powerful machines: **wormhole flow control**

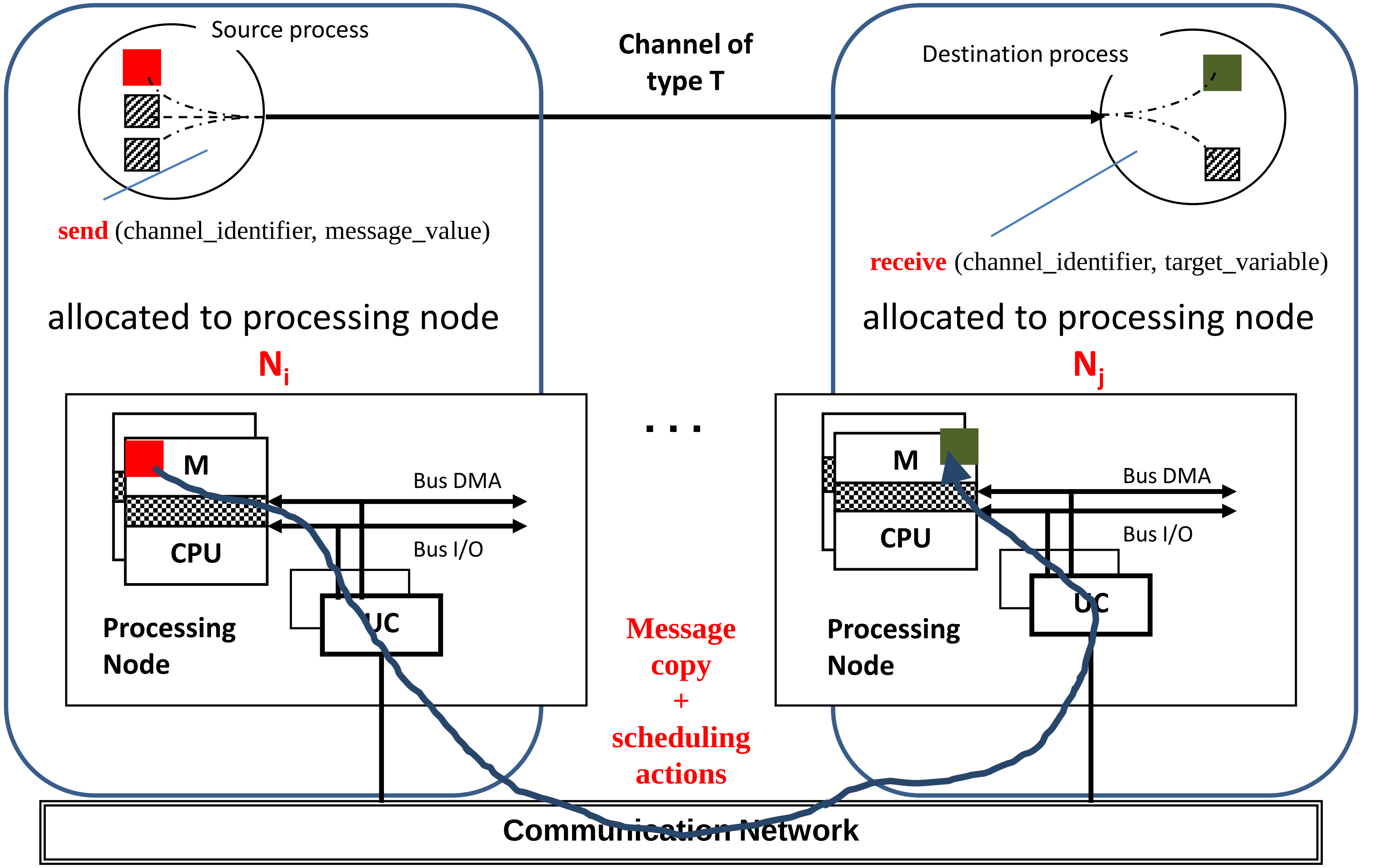
- **Fast Ethernet** (100 Mb/s)
- **Gigabit Ethernet** (1 Gb/s)
- **Myrinet** (1.28 Gb/s)
- **Infiniband** (till 10 Gb/s)
- Optical technology, fotonic networks are emerging (10 – 100 Gb/s)

Communication networks and communication processors

- Example: Myrinet
  - KP included in the network, connected as I/O unit to processing node
  - used for interprocess communication run-time support and/or Network Interface Unit (Network Card)



Interprocess communication run-time support



Source process
Channel of type T
Destination process
send (channel_identifier, message_value)
receive (channel_identifier, target_variable)
allocated to processing node
N_i
allocated to processing node
N_j
M
CPU
Bus DMA
Bus I/O
UC
Processing Node
. . .
Message copy + scheduling actions
M
CPU
Bus DMA
Bus I/O
UC
Processing Node
Communication Network

Distributed run-time support

Principles:

- Channel descriptor allocated in destination node N_j
- *receive* is executed locally by Destination process in Nj
- *send* call by Source process in N_i: *delegated* to destination node N_j
- Delegation consists in a firmware message from N_i to N_j via communication network, containing:

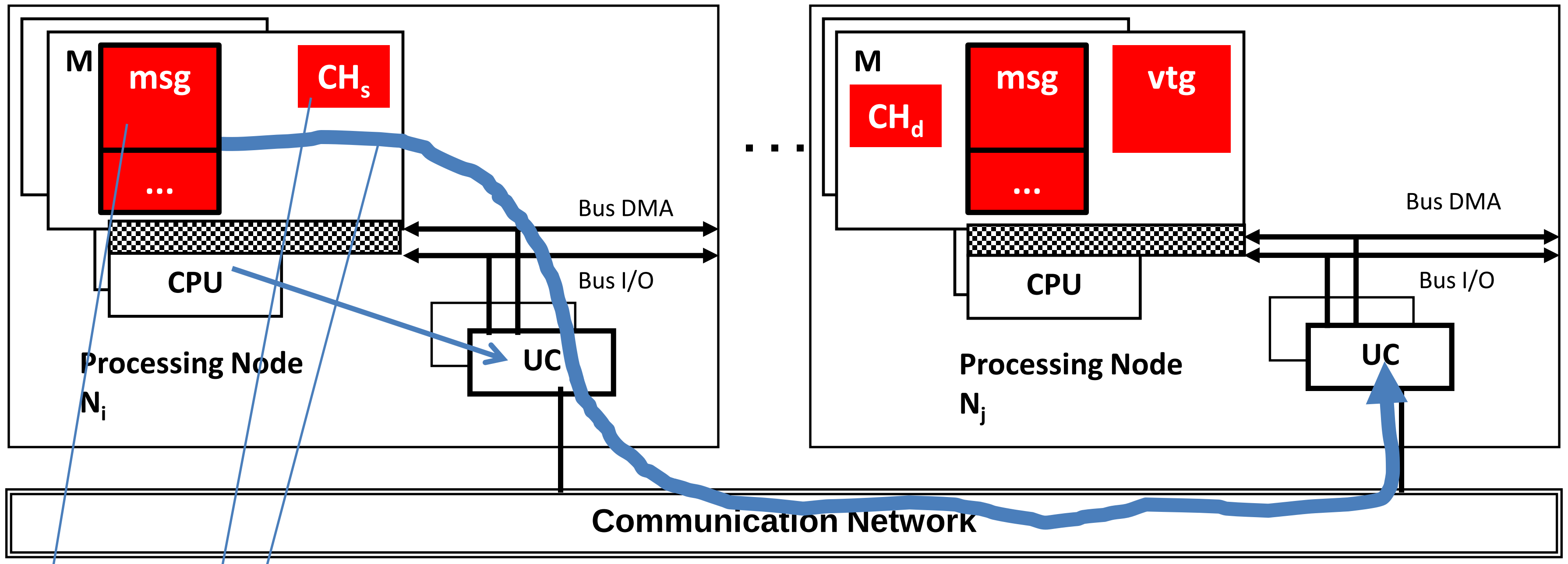
  FW_MSG = (header, channel identifier, message value, Source identifier)

- In Nj, this message is received by the network interface unit (UC_j) and transformed into an interrupt (for CPU_j or KP_j)
- The interrupt handler executes the *send* primitive locally, according to a shared memory implementation
  - and possibly returns an outcome to N_i (Source) via communication network: this action can be avoided according to the detailed implementation scheme

Implementation

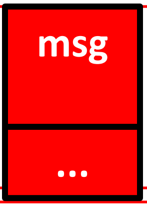
- Channel descriptor
  - data structure **CH_{source}** allocated in N_i: contains information about the *current number of buffered messages* and the *sender_wait* boolean
  - data structure **CH_{dest}** allocated in N_j: the "real" channel descriptor, with the usual structure for a shared memory implementation
- Send
  - verifies the asynchrony degree saturation and, if *buffer_full*, suspends the Source process
  - in any case, the interprocessor message FW_MSG is sent to UC_i, then to N_j via communication network
  - local execution of *send* on Nj, without checking buffer_full; no outcome is returned to N_i in this scheme
- Receive
  - causes the updating of the *number of buffered messages* in **CH_{source}** (interprocessor message to N_i)
  - In N_i, *sender_wait* is checked: if true, Source process is waked up.

send implementation – source node



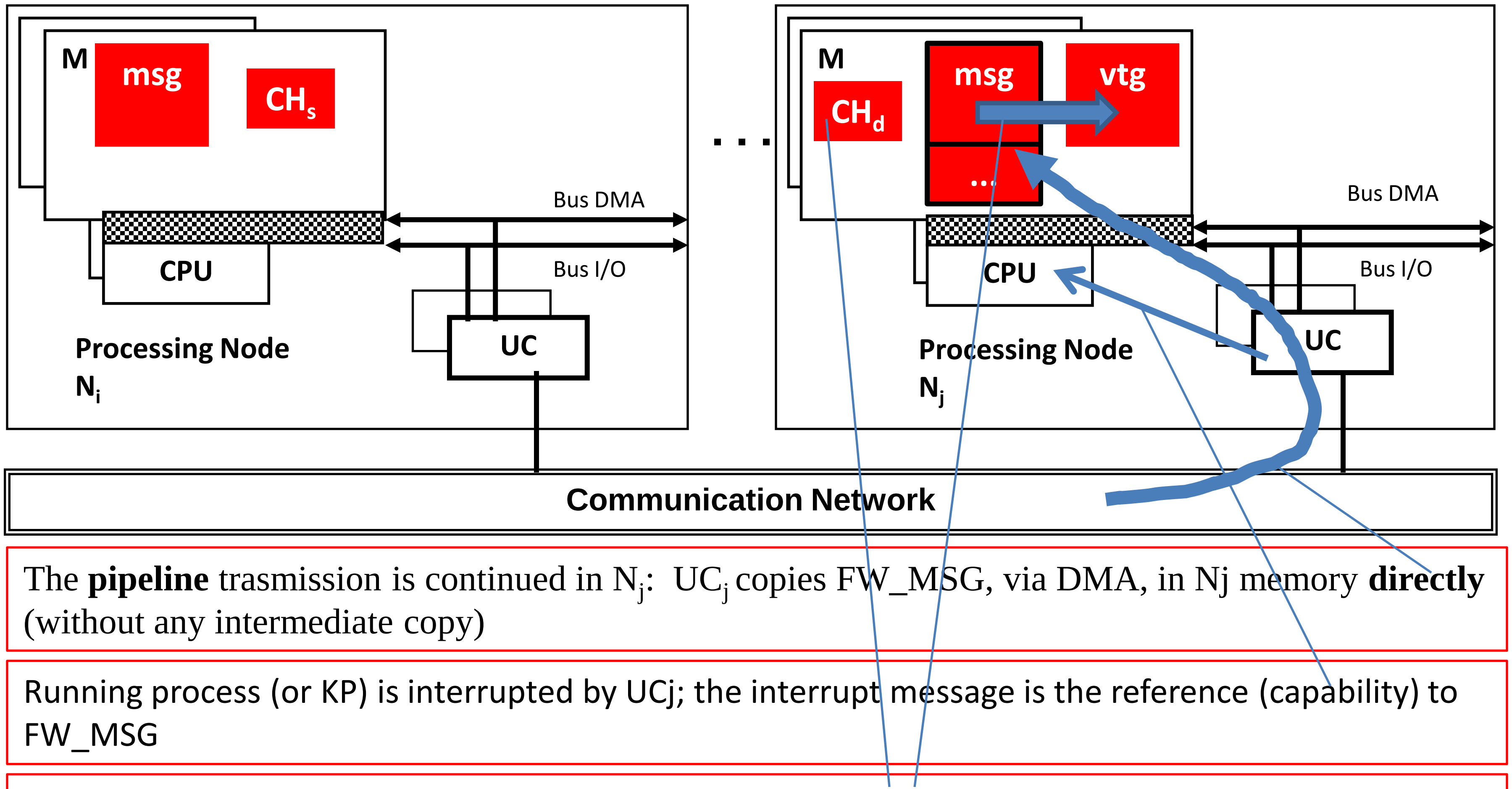
verifies the asynchrony degree saturation and, if *buffer_full*, suspends the Source process

in any case, the interprocessor message FW_MSG
(header, channel identifier, message value, Source process identifier)
is produced and passed to UC_i **by reference**



UC_i exploits **DMA** and transmits FW_MSG to UC_j, via network, **directly in pipeline** (flit by flit, without any intermediate copy in UC_i)

send implementation – destination node



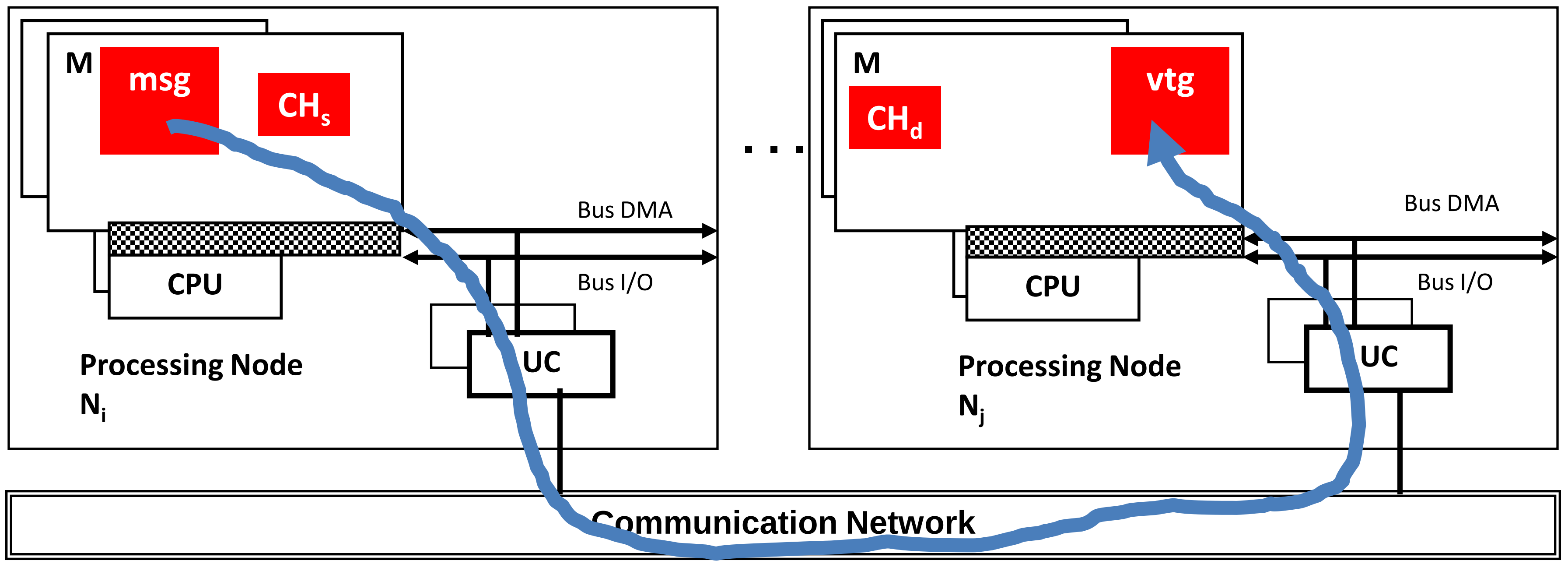
The **pipeline** trasmission is continued in N$_j$: UC$_j$ copies FW_MSG, via DMA, in Nj memory **directly** (without any intermediate copy)

Running process (or KP) is interrupted by UCj; the interrupt message is the reference (capability) to FW_MSG

Running process (or KP) acquires FW_MSG, then Ch$_{dest}$ and VTG, into its own addressing space. So, the *local send* can be executed (without checking buffer_full).

Optimization: UC is KP, thus the additional copy of FW_MSG is saved ! (on the fly execution)

send implementation – memory-to-memory copy



In practice, even in a distributed memory architecture, a memory-to-memory copy can be implemented (plus additional operations for low level scheduling),

provided that the communication network protocol is the **primitive**, firmware one.

If **IP** protocol is adopted, then several additional copies and administrative operations are done. IP overhead is prevailing, also compared to the network latency.

Implementation

- A key point for the *local send* execution on N_j is the addressing space of the process executing the interrupt handler.
- Any process should contain all possible channel descriptors, all possible target variables and process control blocks for all processes allocated on N_j.
- In practice, *static allocation of such objects is impossible*.
- Solution: *dynamic* allocation by means of Capability mechanism.

Interprocess communication cost model

- Base latency: takes into account
  - latency on $N_{i:}$
    - operations on CH_{source},
    - formatting of FW_MSG and delegation to KP_j or UC_j
    - operations in KP_j or UC_j
  - network latency (depending on network kind and dimension, routing and flow control strategies, link latency, link size, number of crossed units: SEE Shared Memory Arch.)
  - latency on N_j: latency of local *send* execution (SEE Shared Memory Arch.)
- Under-load latency:
  - resolution of a client-server model (SEE Shared Memory Arch.), where the destination node (thus, any node) is the server and the possible source nodes (thus, any node) are the clients
  - M/M/1 is a typical (worst-case) assumption
  - parameter *p*: average number of nodes acting as clients, according to the structure of the parallel program and to the process mapping strategies

Typical latencies

- The communication network is used with the *primitive firmware routing and flow-control protocol*:
  - similar result of shared memory run-time, for systems realized in a rack

    $$T_{setup} \sim 10^3 \tau, \; T_{transm} \sim 10^2 \tau$$

  - otherwise, for long distance networks, the transmission latency dominates, e.g.

    $$T_{setup} \sim 10^3 \tau, \; T_{transm} \sim 10^4 \tau \quad \text{till} \quad 10^6 \tau$$

- The communication network is used with the *IP protocol*, i.e., the application is IP-dependent
  - The network is exploited in the primitive way, however an additional overhead is paid due to the protocol actions (e.g., formatting, de-formatting) inside the nodes (+ transmission overhead on long distance networks):

    Rack: $T_{setup} \sim 10^5 \tau, \; T_{transm} \sim 10^4 \tau$

    Long distance: $T_{setup} \sim 10^7 \tau, \; T_{transm} \sim 10^8 \tau$

Exercizes

1. Describe the interprocess communication run-time support in details, in particular the actions inside the source and destination nodes.
2. Evaluate the interprocess communication latency in detail, according to the implementation scheme of Exercize 1.
3. Study the interprocess communication run-time support for clusters whose nodes are SMP or NUMA machines.