Scuola Superiore
Sant'Anna
di Studi Universitari e di Perfezionamento

UNIVERSITÀ DI PISA

Master Program (Laurea Magistrale) in Computer Science and Networking

High Performance Computing Systems and Enabling Platforms

Marco Vanneschi

3. Run-time Support of Interprocess Communications

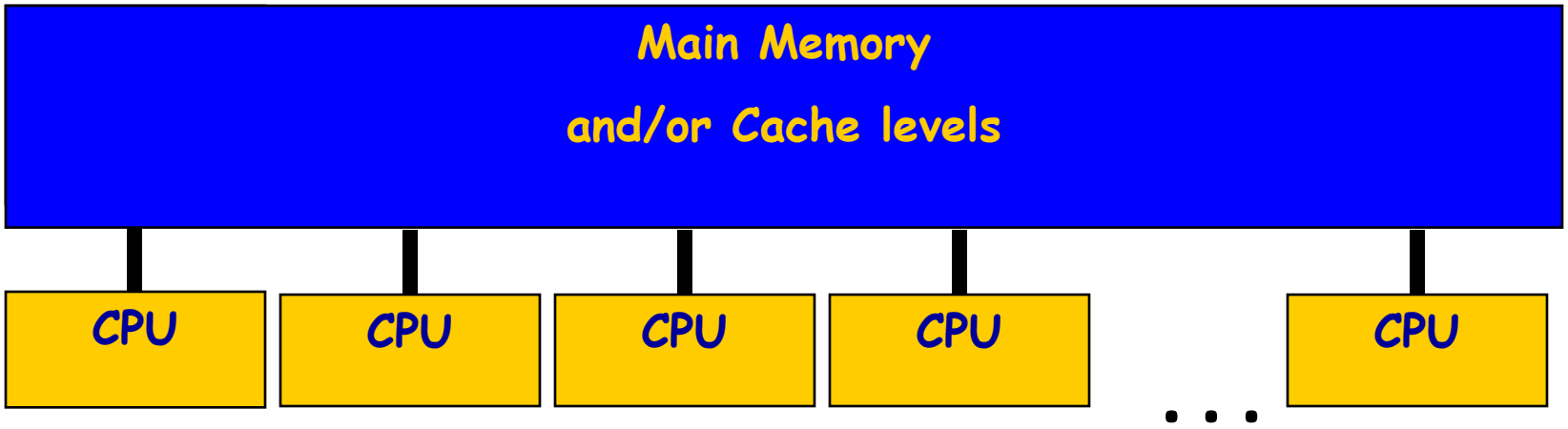
From the Course Objectives

- Provide a solid knowledge framework of concepts and techniques in high-performance computer architecture
  - Organization and structure of enabling platforms based on parallel architectures
  - Support to parallel programming models and software development tools
  - Performance evaluation (cost models)
- Methodology for studying existing and future systems
- Technology: state-of-the-art and trends
  - Parallel processors
  - Multiprocessors
  - Multicore / manycore / … / GPU
  - Shared *vs* distributed memory architectures
  - Programming models and their support
  - General-purpose *vs* dedicated platforms

HPC enabling platforms

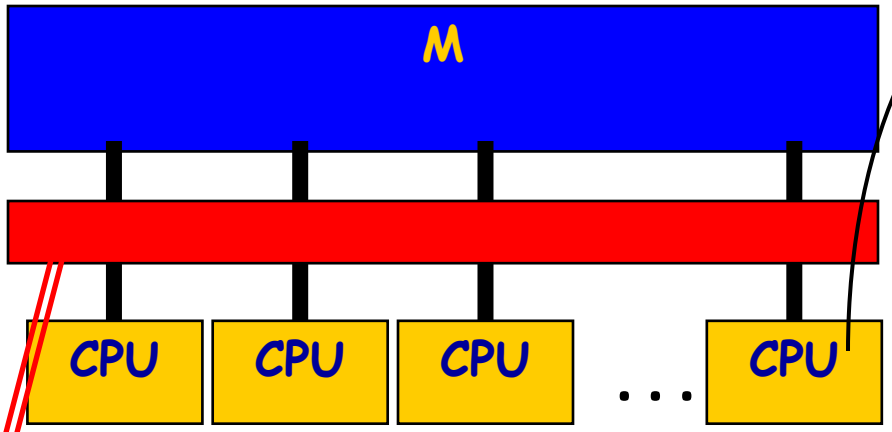
Shared memory multiprocessors

- Various types (SMP, NUMA, …)



- From simple to very sophisticated memory organizations
- Impact on the programming model and/or process/threads run-time support

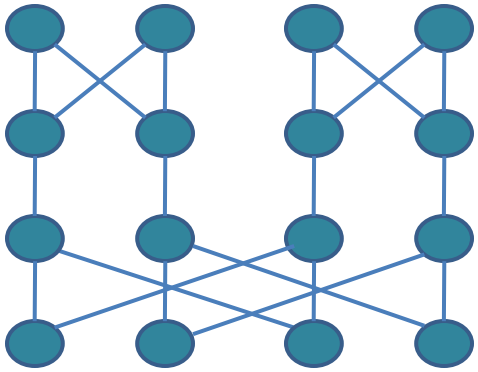
HPC enabling platforms: shared and distributed memory architectures

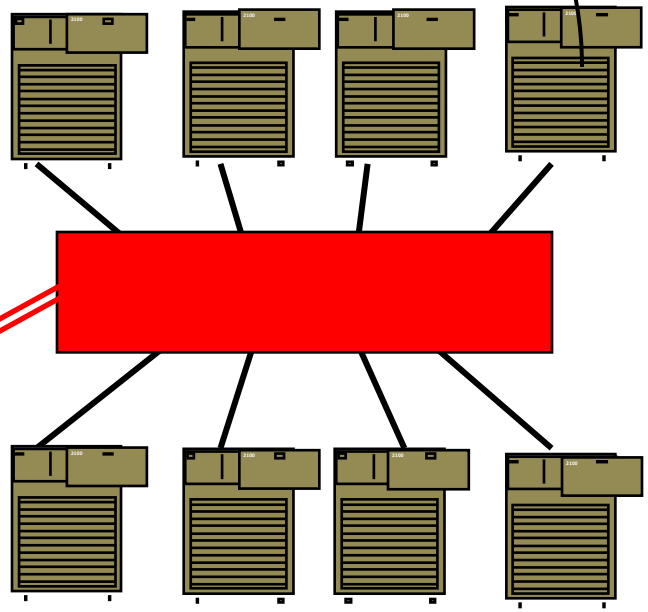


Shared memory multiprocessor

Instruction Level Parallelism CPU (pipeline, superscalar, multithreading, …)

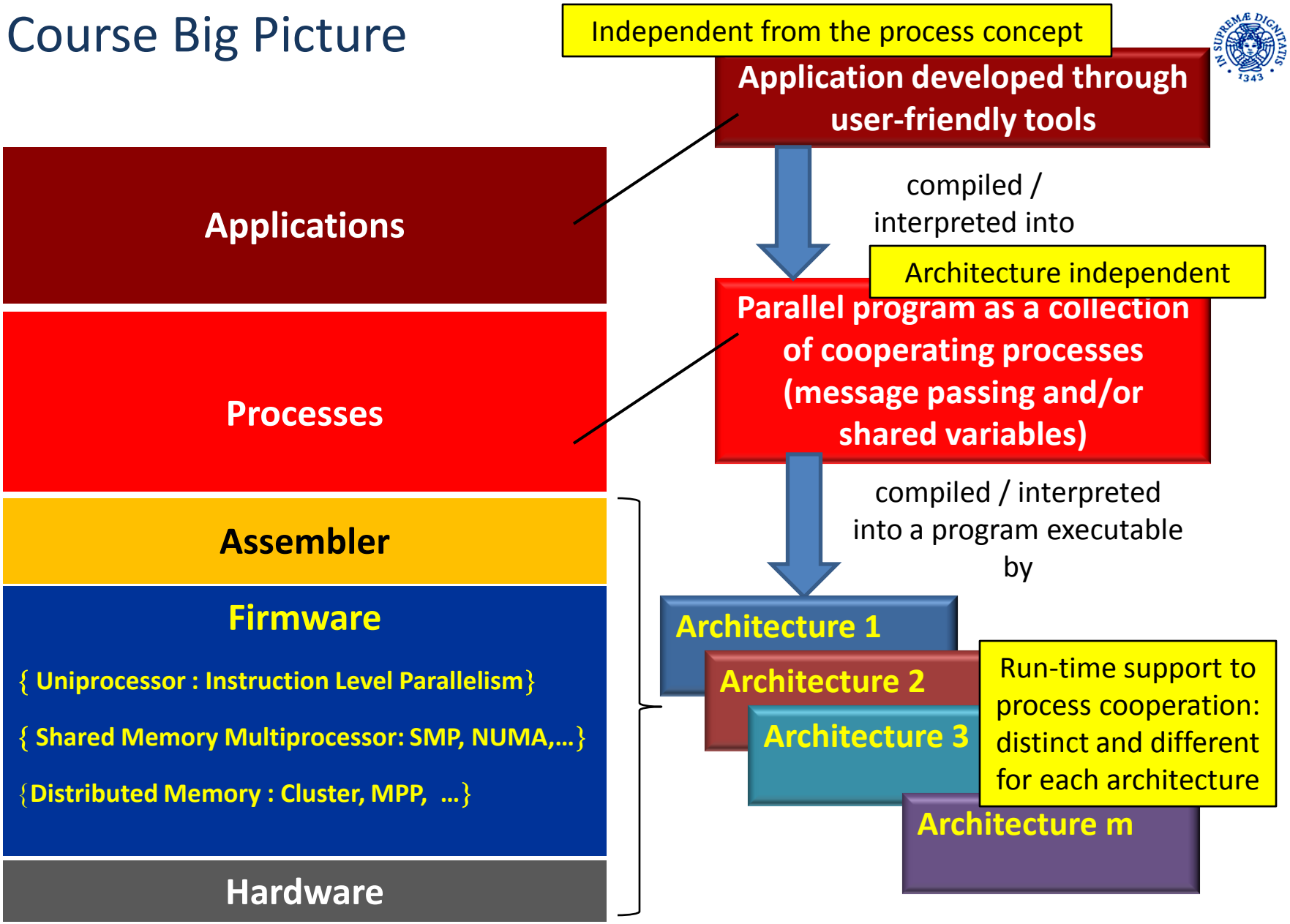
"Limited degree" Interconnection Network ("one-to-few")





Distributed memory multicomputer: PC cluster, Farm, Data Centre, …

Course Big Picture



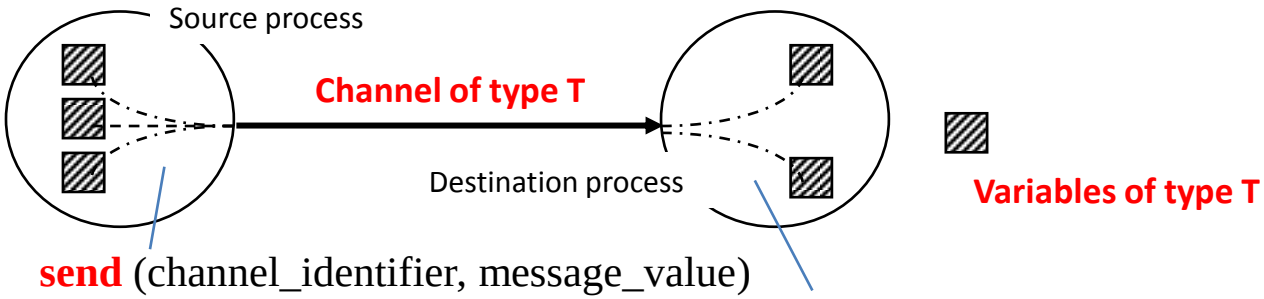
Interprocess communication model

- The parallel architecture will be studied in terms of:
  - **Firmware architecture and operating system impact**
  - **Cost model / performance evaluation**
  - **Run-time support to process cooperation / concurrency mechanisms**

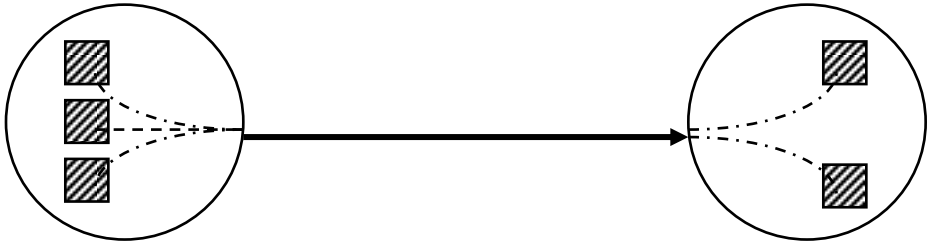
- Without losing generality, we will assume that the intermediate version of a parallel program is according to the:

message-passing model

Message passing model

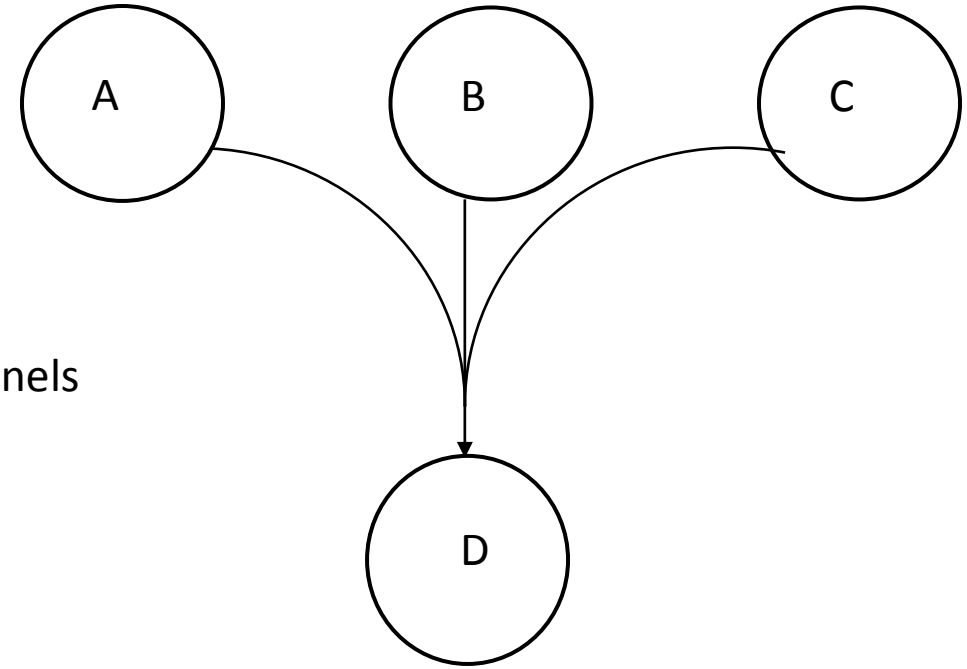


- **send**, **receive** primitives: commands of a concurrent language
- Final effect of a communication: target variable = message value
- Communications channels of a parallel program are identified by **unique names**.
- **Typed communication channels**
  - T = Type (message) = Type (target variable)
  - For each channel, the **message size (length) L** is known statically
- Known **asynchrony degree**: constant ≥ 0 (= 0: synchronous channel)
- Communication forms:
  - **Symmetric** (one-to-one), **asymmetric** (many-to-one)
  - **Constant** or **variable names** of channels



Symmetric channel

Asymmetric channel



Symmetric and asymmetric channels can be
synchronous or aynchronous

Example: farm parallel program

```
parallel EMITTER, WORKER[n], COLLECTOR; channel new_task[n];
EMITTER :: channel in input_stream (asynchrony_degree), worker_free (1); channel out var scheduled; item1 x;
           while true do
              { receive (input_stream, x);
                receive (worker_free, scheduled);
                send (scheduled, x )
              }
WORKER[i] :: channel in new_task[i] (1); channel out worker_free, result; item1 x, y;
           { send (worker_free, new_task[i]);
             while true do
               { receive (new_task[i], x);
                 send (worker_free, new_task[i]);
                 y = F(x);
                 send (result, y }
             }
COLLECTOR :: channel in result (1); channel out output_stream; item1 y;
           while true do
              { receive (result, y);
                send (output_stream, y)
              }
```

This program makes use of

- **symmetric channels (**e.g. input_stream, output_stream))
- **asymmetric channels** (e.g. worker_free, result)
- **constant-name channels** (e.g. input_stream)
- **variable-name channels** (e.g. scheduled)
- **array of processes** (WORKER[n])
- **array of channels** (new_task[n])

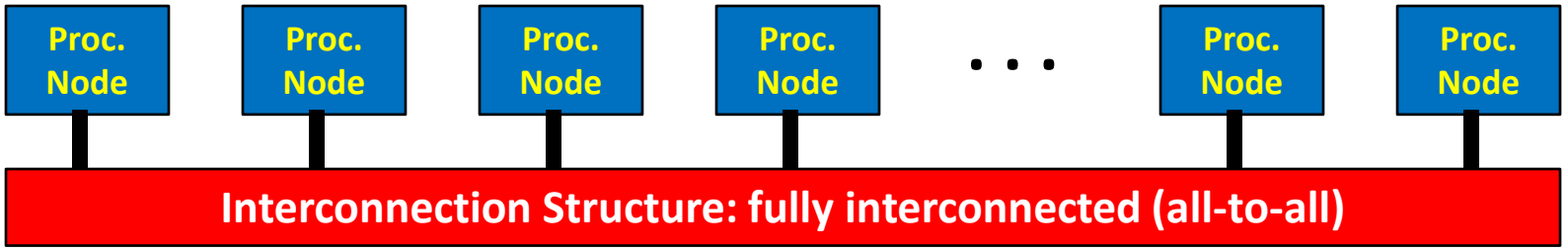
In the study of run-time support, the focus will be on the **basic case**:

symmetric, constant-name, scalar channels

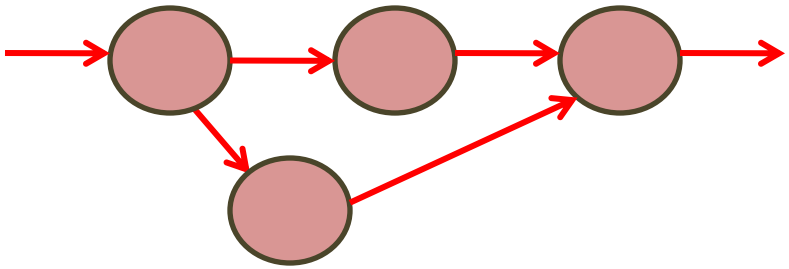
Cost models and abstract architectures

- Performance parameters and *cost models*
  - for each level, a cost model to evaluate the system performance properties
    - Service time, bandwidth, efficiency, scalability, latency, response time, …, mean time between failures, …, power consumption, …
- Static *vs* dynamic techniques for performance optimization
  - the importance of **compiler technology**
  - **abstract architecture** *vs* **physical/concrete architecture**
    - *abstract architecture*: a semplified view of the concrete one, able to describe the essential performance properties
  - **relationship between the abstract architecture and the cost model**
  - in order to perform optimizations, *the compiler "sees" the abstract architecture*
    - often, the compiler *simulates* the execution *on* the abstract architecture

Example of abstract architecture

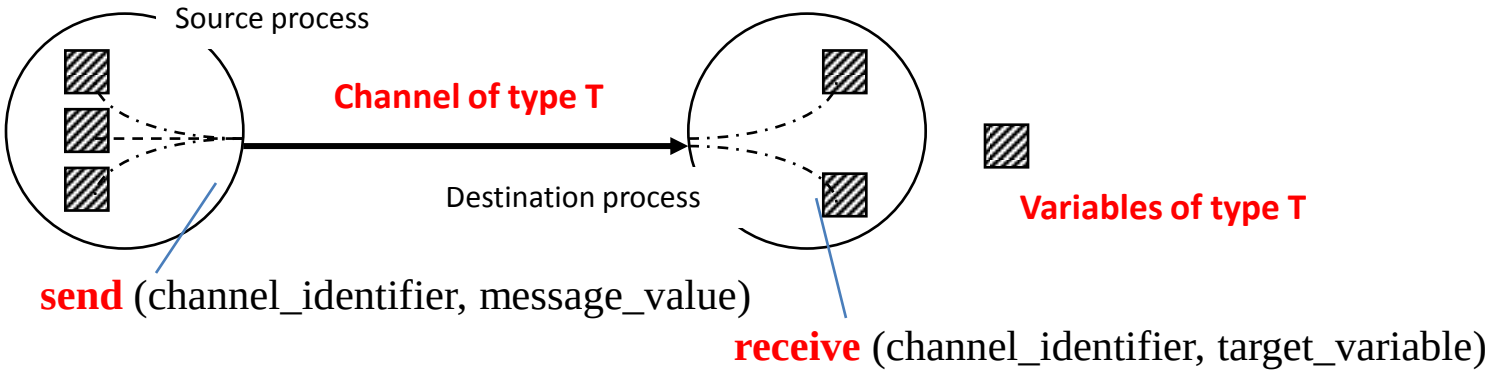


- **Processing Node**= (CPU, memory hierarchy, I/O)
  - Same characteristics of the concrete architecture node
- Parallel program allocation onto the Abstract Architecture: **one process per node**
  - Interprocess communication channels: one-to-one correspondence with the Abstarct Architecture interconnection network channels



Process Graph for the parallel program = **Abstract Architecture Graph** (same topology)

Cost model for interprocess communication



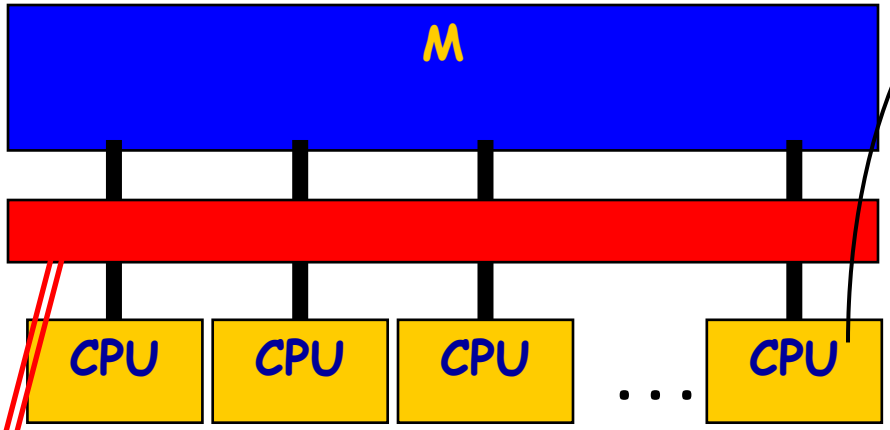
$$T_{send} = T_{setup} + L * T_{transm}$$

- T_{send} = **Average latency of interprocess communication**
  - delay needed for copying a message_value into the target_variable
- **L** = Message length
- T_{setup}, T_{transm}: known parameters, **evaluated for the concrete architecture**
- *Moreover,* the cost model must include the characteristics of possible **overlapping** of communication and internal calculation

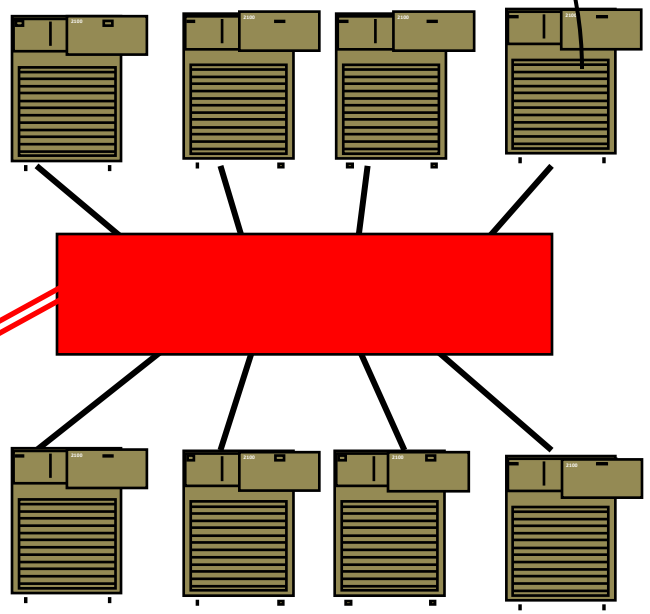
Parameters T_{setup}, T_{transm} evaluated as functions of several characteristics of the concrete architecture

Memory access time, interconnection network routing and flow-control strategies, CPU cost model, and so on

Instruction Level Parallelism CPU (pipeline, superscalar, multithreading, …)

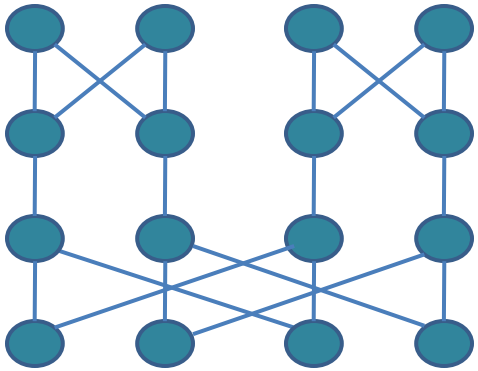


Shared memory multiprocessor

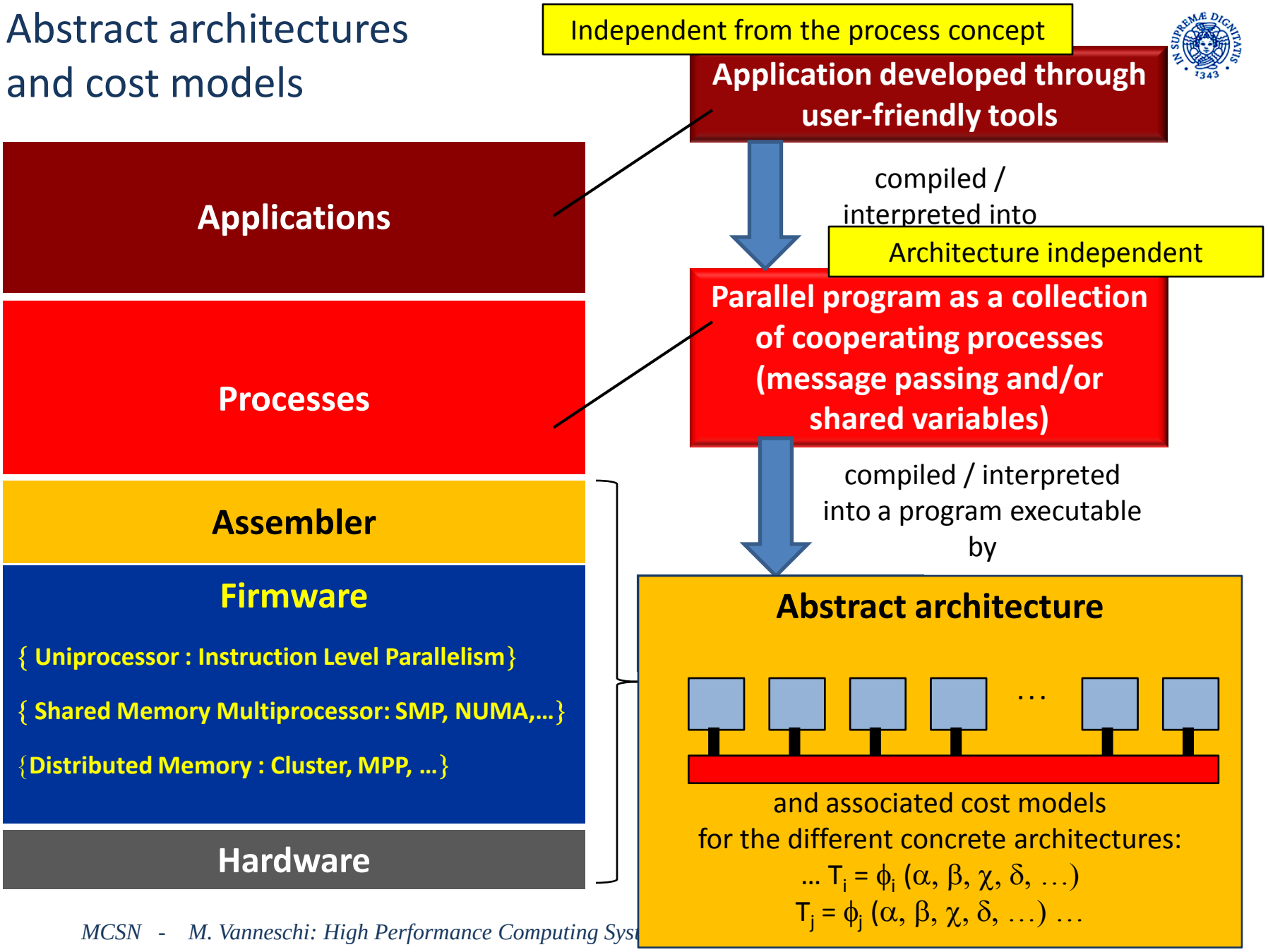


Distributed memory multicomputer: PC cluster, Farm, Data Centre, …

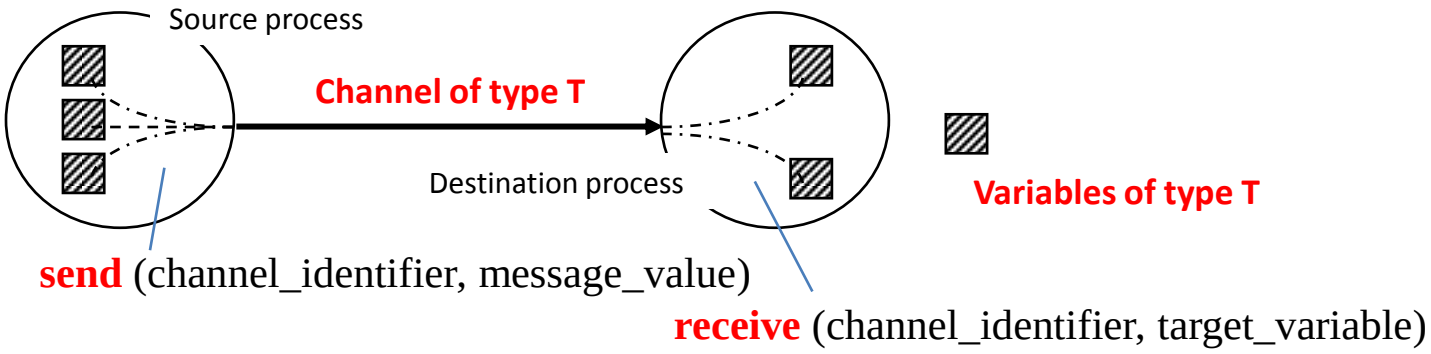
"Limited degree" Interconnection Network ("one-to-few")



Abstract architectures and cost models



Run-time support: shared variables, uniprocessor version

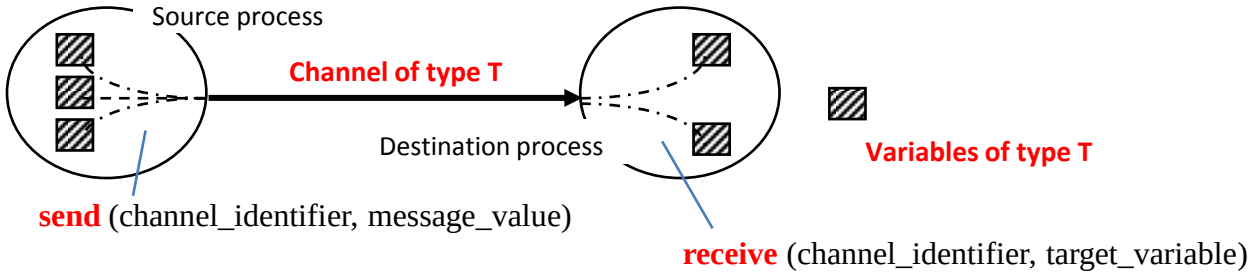


Run-time support implemented through **shared variables.**

Initial case: uniprocessor.

The run-time support for shared memory multiprocessors and distributed memory multicomputers will be derived through modifications of the uniprocessor version.

Run-time support: shared variables, uniprocessor version



Basic data structure of run-time support for send-receive:

CHANNEL DESCRIPTOR

sender_wait: bool (if true: sender process is in WAIT state)
receiver_wait: bool (if true: receiver process is in WAIT state)
buffer: FIFO queue of N = k + 1 positions of type T (k = asynchrony degree)
message_length: integer
sender_PCB_ref: reference to Process Control Block of sender process
receiver_PCB_ref: reference to Process Control Block of receiver process

Channel descriptor is **shared** *by* Sender process *and* Receiver process:

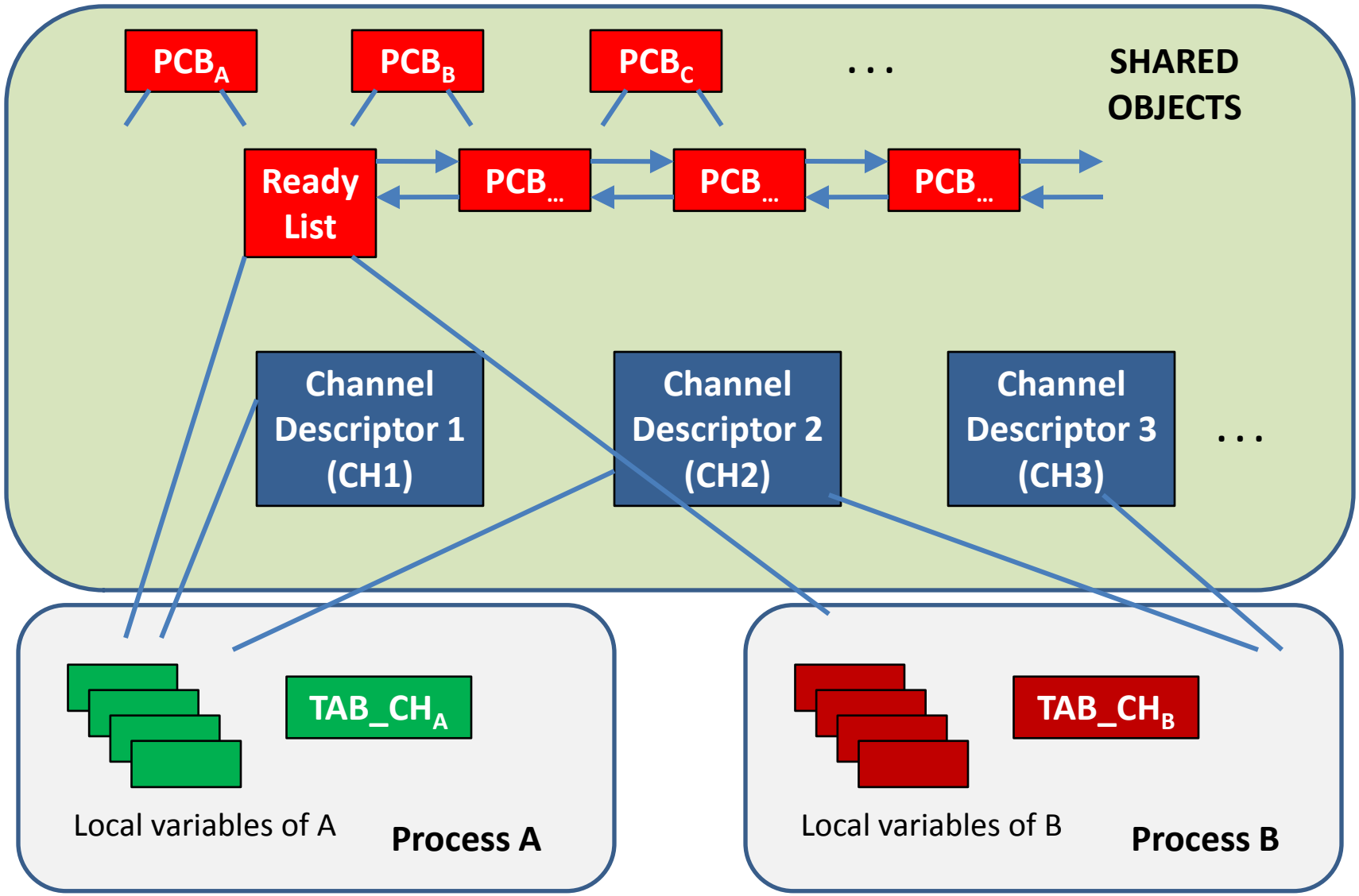
it belongs to the virtual memories (addressing spaces) of <u>both</u> processes

Run-time support variables

- **Channel Descriptor**: *shared*
- **Sender PCB**: *shared*
- **Receiver PCB**: *shared*
- **Ready List**: *shared*, list of PCBs of processes in READY state
- **Channel Table**: *private*, maps channel identifiers onto channel descriptor addresses

- *Note*: Sender process and Receiver process have *distinct* logical addresses of Channel Descriptor in the respective logical addressing spaces.

Shared objects for run-time support



First version of send-receive support

send (ch_id, msg_address) ::

CH address =TAB_CH (ch_id);

put message value into CH_buffer (msg_address);

if receiver_wait *then* { receiver_wait = false;

wake_up partner process (receiver_PCB_ref) };

if buffer_full *then* { sender_wait = true;

process transition into WAIT state: **context switching** }

receive (ch_id, vtg_address) ::

CH address =TAB_CH (ch_id);

if buffer_empty *then* { receiver_wait = true;

process transition into WAIT state: **context switching** }

else **get** message value from CH buffer and assign it to target variable (vtg_address);

if sender_wait *then* { sender_wait = false;

wake_up partner process (sender_PCB_ref) }

Notes

- *send*, *receive* are **indivisible** procedures
  - in uniprocessor machines: executed with *disabled interrupts*
- *send*, *receive* procedures provide to:
  - **pure communication between Sender and Receiver processes, and**
  - **low level scheduling of Sender and Receiver processes** (process state transitions, processor management)
- **wake_up** procedure: put PCB into Ready List
- Process transition into WAIT state (**context_switching**):
  - *send* procedure: Sender process continuation = the procedure *return* address
  - *receive* procedure: Receiver process continuation = *procedure address itself* (i.e., *receive* procedure is *re-executed* when the Receiver process is resumed)
- In this implementation, a communication implies two message copies (from message variable into CH buffer, from CH buffer into target variable)
  - assuming the most efficient implementation model: **entirely in user space**;
  - if implemented in *kernel space*: (several) additional copies are necessary for user-to-kernel and kernel-to-user parameter passing.

Cost model : communication latency

Communication latency: *latency needed to copy a message into the corresponding target variable*.

Send latency:

$$T_{send} = T_{setup1} + L * T_{transm1}$$

Receive latency:

$$T_{receive} = T_{setup2} + L * T_{transm2}$$

With good approximation:

$$T_{setup1} = T_{setup2} = T_{setup}$$

$$T_{trasm1} = T_{transm2} = T_{transm}$$

$L * T_{transm}$: actions for message copies into / from CH_Buffer;

all the other actions are independent of the message length, and are evaluated in T_{setup}

Communication latency:

$$L_{com} = T_{send} + T_{receive} = 2\,(T_{setup} + L * T_{transm})$$

“Zero-copy” communication support

- In user space, we can reduce to the minimum the number of message copies for a communication: **just one copy** (“zero-copy” stands for “zero *additional* copies”)
- *Direct copy of message value into target variable, “skipping” the channel buffer*

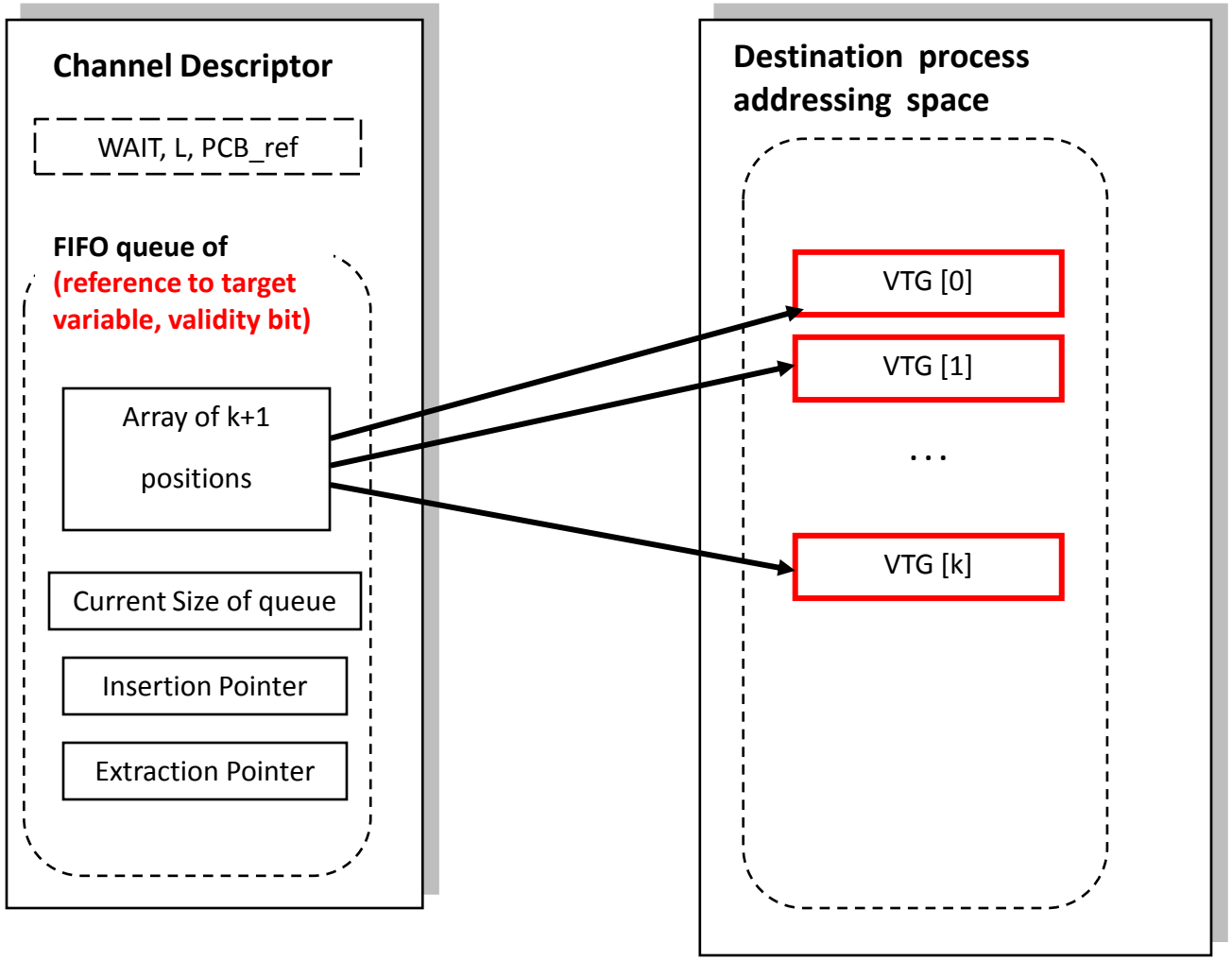
- Target variable becomes a *shared* variable (*statically* or *dinamically* shared)
- Channel descriptor doesn’t contain a buffer queue of message values: it contains all the synchronization and scheduling information, and ***references to target variables***
- Easy implementation of zero-copy communication for
  - ***synchronous*** channel,
  - ***asynchronous channel when the destination process is WAITing***;
  - in any case, Receiver process continuation = *return a*ddress of the *receive* procedure (as in the *send* procedure).

Zero-copy asynchronous communication

- FIFO queue of target variables implemented in the destination process addressing space
  - i.e., the Receiver process refers to *a different copy* of the target variable after every *receive* execution: Receiver is *compiled* in this way;
  - in a *k*-asynchronous communication, there is a queue of *(k + 1)* target variables which are statically allocated in the Receiver addressing space;
  - these target variables are shared with the Sender process (thus, they are allocated – *statically or dynamically* – in the Sender process addressing space too).

- Channel descriptor:
  - WAIT: bool (if true, Sender *or* Receiver process is WAITing)
  - message length: integer
  - Buffer: FIFO queue of (**reference to target variable**, **validity bit**)
  - PCB_ref: reference to PCB of WAITing process (if any)

Zero-copy implementation



Principle:

Sender copies the message into an instance of the target variable (in FIFO order),

on condition that it is not currently used by Receiver (**validity bit** used for mutual exclusion)

Zero-copy implementation

- **Send**: copies the message into the target variable referred by the CH_Buffer Insertion Pointer, *on condition that the validity bit is* ***set***, and modifies the CH_Buffer state accordingly.

- **Receive**: if CH_Buffer is not empty, ***reset*** the validity bit of the CH_Buffer position referred by the CH_Buffer Extraction Pointer, and modifies the CH_Buffer state accordingly
  - at this point, the Receiver process can utilize the copy of the target variable referred by the CH_Buffer Extraction Pointer *directly* (i.e., without copying it); when the target variable utilization is completed, the validity bit is ***set*** again in the CH_Buffer position.

During the target variable utilization by the Receiver, its value can not be modified by the Sender (validity bit mechanism).

Alternative solution for Receiver process compilation: *loop unfolding o*f Receiver loop (receive – utilize) and explicit synchronization.

Zero-copy implementation

send (ch_id, msg_address) ::

CH address =TAB_CH (ch_id);

if (CH_Buffer [Insertion_Pointer].validity_bit = 0) *then*

{ wait = true;

copy reference to Sender_PCB into CH.PCB_ref

process transition into WAIT state: **context switching** };

copy message value into the target variable referred by

CH_Buffer [Insertion_Pointer].reference_to_target_variable ;

modify CH_Buffer. Insertion_Pointer and CH_Buffer_Current_Size;

if wait *then*

{ wait = false;

wake_up partner process (CH.PCB_ref) };

if buffer_full *then*

{ wait = true;

copy reference to Sender_PCB into CH.PCB_ref

process transition into WAIT state: **context switching** }

Exercize

Write an equivalent representation of the following benchmark, using zero-copy communication on a *k*-asynchronous channel:

```
Receiver ::
    int A[N]; int v; channel in ch (k);
    for (i = 0; i < N; i++)
                { receive (ch, v);
                  A[i] = A[i] + v }
```

The answer includes the definition of the *receive* procedure, and the Receiver code before and after the *receive* procedure for a correct utilization of zero-copy communication (*send* procedure: see previuos page).

Cost model for zero-copy communication

- Zero-copy implementation:
  - the *receive* latency is negligible compared to the *send* latency: the Receiver doesn't copy the message into the target variable;
  - the message copy into the target variable is almost entirely paid by the Sender;
  - the *receive* primitive overhead is limited to relatively few operations for synchronization.
- With good approximation, the latency of an interprocess communication is the *send* latency:

$$L_{com} = T_{send} = T_{setup} + L * T_{transm}$$

- Typical values for uniprocessor systems:
  - $T_{setup} = (10^2 – 10^3)\ \tau$
  - $T_{transm} = (10^1 – 10^2)\ \tau$
- One or more order of magnitude increase for parallel architectures, according to the characteristics of memory hierarchy, interconnection network, etc.

Zero-copy trade-off

- Trade-off of the zero-copy mechanism:
  - *receive* latency is not paid,
  - but some additional waiting phases can be introduced in the Sender behaviour (validity bit).
- Strategy:
  - increase the asynchrony degree in order to reduce the probability of finding the validity bit at “set” during a *send*
    - *parallel programs with good load balance are able to exploit this feature.*

Statically *vs* dinamically shared objects

- In order to reduce the addressing space size
  - e.g., the Sender addressing space as far as **target variables** are concerned

  (and also to improve generality of use and protection),

  we can provide mechanisms for the DYNAMIC allocation of VIRTUAL memories of processes
  - e.g., the Sender **acquires the target variable dynamically** (with "copy" rights only) in its own addressing space, and releases the target variable as soon as the message copy has been done.
- A general mechanism, called **CAPABILITY-based ADDRESSING**, exists for this purpose:
  - e.g., the Receiver passes to the Sender the *Capability* on the target variable, i.e. a "ticket" to acquire the target variable dynamically in a protected manner. The Sender loses the ticket after the strictly needed utilization.
- Capabilities are a mechanism to implement the *References* to indirectly shared objects (reference to target variables, PCB, … in the Channel Descriptor).

Indirectly-referred shared data structures

- The so called *"shared pointers" problem*
- Example:
  - *Channel descriptor* is a shared object
    - it is referred **directly** by Sender and Receiver process, each process with its own logical address.
  - *Target variable* (VTG), or *PCBs* (sender PCB, receiver PCB) are shared objects
    - They are referred **indirectly**: Channel Descriptor contains references (pointers) to VTGs and PBCs. Thus, *these references (pointers) are shared objects* themselves.
    - PROBLEM: How to represent them?
    - An apparently simple solution is: shared references (pointers) are *physical* addresses. Despite its popularity, many problems arise in this solution.
    - If we wish to use **logical addresses**, the PROBLEM exists.
- Other example: pointed PCBs in Ready List. Other notable examples will be met in parallel architectures.

Indirectly-referred shared data structures

Static solutions:

- Coinciding logical addresses: all the processes sharing an indirectly-referred shared data structure have the *same* logical address of such structure
  - the shared pointer is equal to the coinciding logical addresss
  - feasible, however there are problem of addressing space fragmentation
- Distinct logical addresses: each process has *its own* logical address for an indirectly-referred shared data structure
  - more general solution, no fragmentation problems
  - the shared pointer is a **unique indentifiers** of the indirectly referred shared object
  - each process transforms the identifier into its own logical address (private **table**)

Indirectly-referred shared data structures

Example of static solutions for PCBs in send-receive support:

- Coinciding logical addresses: all the PCBs have the same logical address in all the addressing spaces
  - CH fields *Sender_PCB_ref* and *Receiver_PCB_ref* contain the coinciding logical addresses of sender PCB and receiver PCB
- Distinct logical addresses:
  - sender PCB and receiver PCB have unique identifiers (determined at compile time): *Sender_PCB_id* and *Receiver_PCB_id*
  - these identifiers are contained in the CH fields *Sender_PCB_ref* and *Receiver_PCB_ref*
  - when the sender wishes to wake-up the receiver, get the *Receiver_PCB_id* from CH field *Receiver_PCB_ref*, tranforms it into a private logical address using a private table, thus can refer the receiver PCB
  - analogous: for the manipulation of sender PCB by the receiver process.

Capabilities

Dynamic solution to the shared pointers problem:

- The indirectly-referred shared objects are not statically allocated in all the addressing spaces
- They are allocated / deallocated dynamically in the addressing spaces of the processes that need to use them
- Example:
  - the Receiver PCB is statically allocated in the receiver addressing space only;
  - when the Sender process wishes to wake-up the receiver:
    - it **acquires dynamically** the Receiver PCB into its addressing space, and
    - manipulates such PCB properly,
    - finally **releases** the Receiver PCB again, deallocating it from its addressing space.

Capabilities

Capability implementation

- The shared pointer is an information (*capability*) that enables the acquiring process *to allocate space* and *to refer* the indirectly referred shared object by means of logical addresses
- Formally, a capability is a couple (object identifier, access rights) and a mechanism to generate the object reference
  - it is related to the concept of Protected Addressing Space
- In practice, the implementation of a shared pointer is the entry of the Address Relocation Table relative to the shared object
  - this entry is added to the Relocation Table of the acquiring process, thus allowing this process to determine the logical address of the shared object.

X: A's page to be allocated dynamically in B's space (e.g. PCB_A)

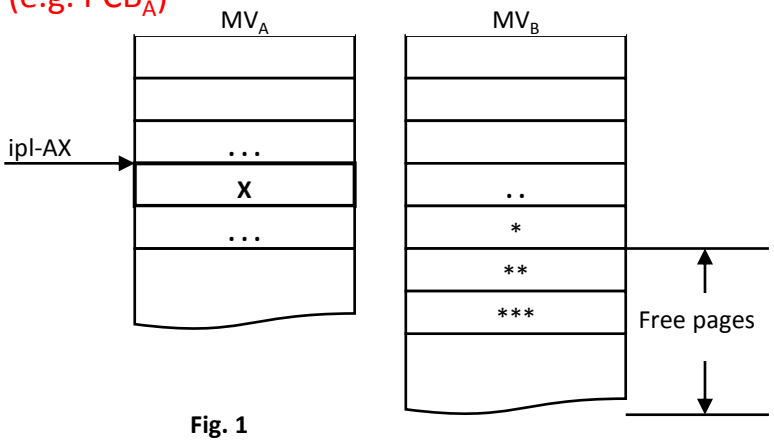


Fig. 1

A copies the X's Relocation Table entry into a shared structure S.

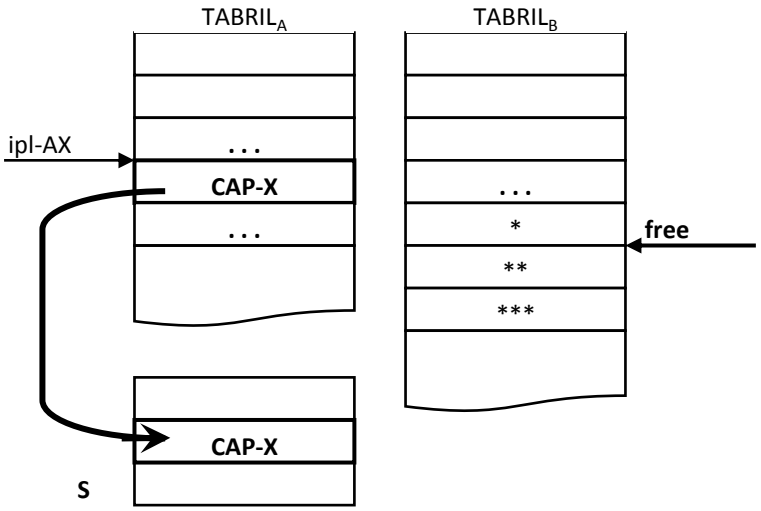


Fig. 2

S: shared structure through which X-capability is passed from A to B (e.g. Channel Descriptor, field Sender_PCB_ref)

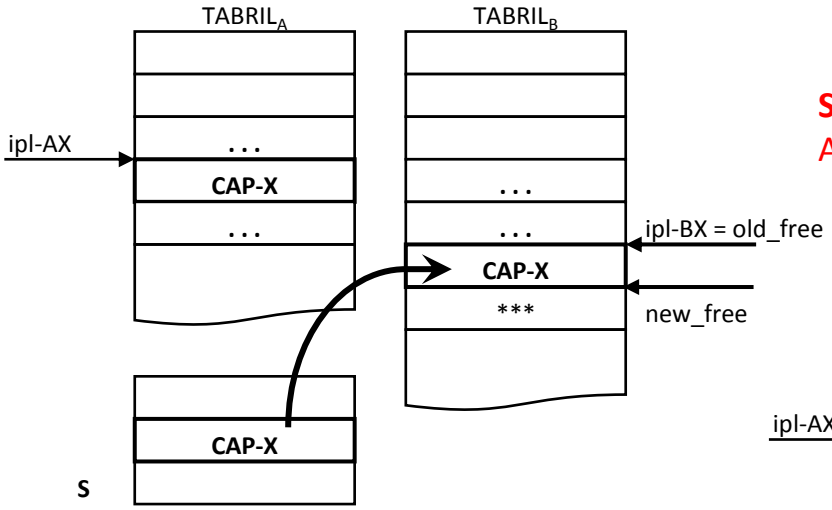


Fig. 3

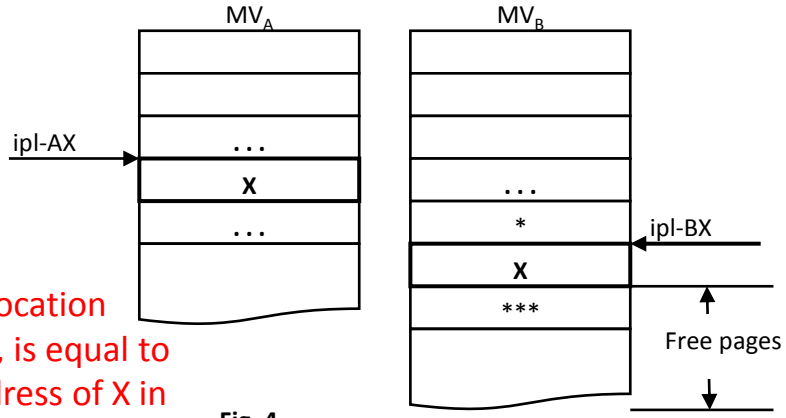


Fig. 4

B copies X-capability into a free position of its Relocation Table. The logical page identifier of X, in B's space, is equal to the Relocation Table position: thus the logical address of X in B's space is now determined.

Capabilities

- Several advantages
  - minimized addressing spaces,
  - increased protection,
  - solution of object allocation problems thar are very difficult / impossible to be solved statically (notable cases will be met in parallel architecture run-time support)
- at low overhead
  - few copies of capabilites, few operations to determine the logical address
  - comparable (or less) with respect to Distinct Logical Address technique and Physical Address technique.
- An efficient technique for implementation of *new/malloc* mechanisms in a concurrent context.
- Efficient alternative to kernel space solutions.