

# High Performance Computing – Programming Paradigms and Scalability

## Part 3: Foundations

PD Dr. rer. nat. habil. Ralf-Peter Mundani  
Computation in Engineering (CiE)  
Scientific Computing (SCCS)

Summer Term 2015

# Overview

- **terms and definitions**
- process interaction for MemMS
- process interaction for MesMS
- example of a parallel program
- load balancing
- state-of-the-art LB strategies

*A distributed system is the one  
that prevents you from working because of the failure  
of a machine that you had never heard of.*  
—Leslie Lamport

# Terms and Definitions

- **general design questions**
  - several considerations have to be taken into account for writing a parallel program (either from scratch or based on an existing sequential program)
  - standard questions comprise
    - which part of the (sequential) program can be done in parallel
    - where / how often does synchronisation become necessary
    - which programming model to be used
    - what kind of (data) structure to be used for parallelisation
    - what about load balancing strategies
    - what kind of architecture is the target machine
    - ...

# Terms and Definitions

- **dependence analysis**

- processes / (blocks of) instructions cannot be executed simultaneously if there exist dependencies between them
- hence, a dependence analysis of a given algorithm is necessary
- example

```
for_all_processes i ← 0 to N do
    a[i] ← i + 1
od
```

- what about the following code

```
for_all_processes i ← 1 to N do
    x ← i - 2*i + i*i
    a[i] ← a[x]
od
```

- as it is not always obvious, an algorithmic way of recognising dependencies (via the compiler, e.g.) would be preferable

# Terms and Definitions

- **dependence analysis (cont'd)**
  - BERNSTEIN (1966) established a set of conditions, sufficient for determining whether two processes can be executed in parallel
  - definitions
    - $I_i$  (input): set of memory locations read by process  $P_i$
    - $O_i$  (output): set of memory locations written by process  $P_i$
  - BERNSTEIN's conditions

$$I_1 \cap O_2 = \emptyset \quad I_2 \cap O_1 = \emptyset \quad O_1 \cap O_2 = \emptyset$$

- example

$$P_1: a \leftarrow x + y$$

$$P_2: b \leftarrow x + z$$

$$I_1 = \{x, y\}, O_1 = \{a\}, I_2 = \{x, z\}, O_2 = \{b\} \rightarrow \text{all conditions fulfilled}$$

# Terms and Definitions

- dependence analysis (cont'd)
  - further example

$$P_1: a \leftarrow x + y$$

$$P_2: b \leftarrow a + b$$

$$I_1 = \{x, y\}, O_1 = \{a\}, I_2 = \{a, b\}, O_2 = \{b\} \rightarrow I_2 \cap O_1 \neq \emptyset$$

- BERNSTEIN's conditions help to identify instruction-level parallelism or coarser parallelism (loops, e.g.)
- hence, sometimes dependencies within loops can be solved
- example: two loops with dependencies – which to be solved

loop A:

```
for i ← 2 to 100 do
    a[i] ← a[i-1] + 4
od
```

loop B:

```
for i ← 2 to 100 do
    a[i] ← a[i-2] + 4
od
```

# Terms and Definitions

- dependence analysis (cont'd)
  - expansion of loop  $B$

$$\begin{array}{ll}
 a[2] \leftarrow a[0] + 4 & a[3] \leftarrow a[1] + 4 \\
 a[4] \leftarrow a[2] + 4 & a[5] \leftarrow a[3] + 4 \\
 a[6] \leftarrow a[4] + 4 & a[7] \leftarrow a[5] + 4 \\
 \vdots & \vdots
 \end{array}$$

- hence,  $a[3]$  can only be computed after  $a[1]$ ,  $a[4]$  after  $a[2]$ , ...
  - computation can be split into two independent loops

```

a[0] ← ...
for i ← 1 to 50 do
  j ← 2*i
  a[j] ← a[j-2] + 4
od
    
```

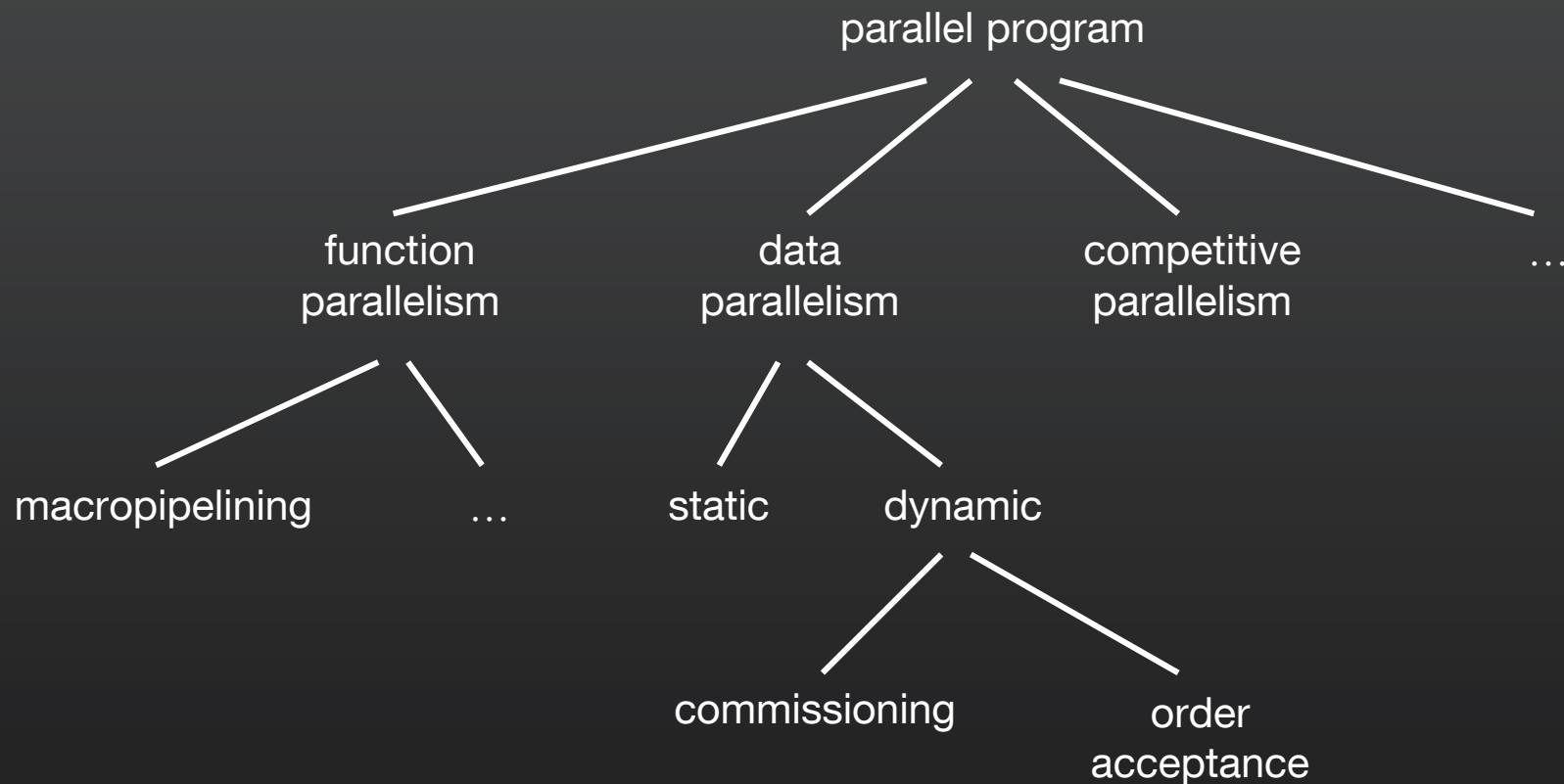
```

a[1] ← ...
for i ← 1 to 50 do
  j ← 2*i + 1
  a[j] ← a[j-2] + 4
od
    
```

- many other techniques for recognising / creating parallelism exist (see also Chapter 4: Dependence Analysis)

# Terms and Definitions

- structures of parallel programs
  - examples of structures





# Terms and Definitions

- **function parallelism**
  - parallel execution (on different processors) of components such as functions, procedures, or blocks of instructions
  - drawback
    - separate program for each processor necessary
    - limited degree of parallelism → limited scalability
  - macropipelining for data transfer between single components
    - overlapping parallelism similar to pipelining in processors
    - one component (producer) hands its processed data to the next one (consumer) → stream of results
    - components should be of same complexity (→ idle times)
    - data transfer can either be synchronous (all components communicate simultaneously) or asynchronous (buffered)

# Terms and Definitions

- **data parallelism**
  - parallel execution of same instructions (functions or even programs) on different parts of the data (SIMD)
  - advantages
    - only one program for all processors necessary
    - in most cases ideal scalability
  - drawback: explicit distribution of data necessary (MesMS)
  - structuring of data parallel programs
    - *static*: compiler decides about parallel and sequential processing of concurrent parts
    - *dynamic*: decision about parallel processing at run time, i.e. dynamic structure allows for load balancing (at the expenses of higher organisation / synchronisation costs)

# Terms and Definitions

- **competitive parallelism**
  - parallel execution of different processes (based on different algorithms or strategies) all solving the same problem
  - advantages
    - as soon as first process found the solution, computations of all subsequent processes are allowed to stop
    - on average, superlinear speed-up possible
  - drawback
    - lots of different programs necessary
  - nevertheless, rare case of parallel programs
  - examples: sorting algorithms, theorem proving within computational semantics

# Terms and Definitions

- **parallel programming languages**
  - explicit parallelism
    - parallel programming interfaces
      - extension of sequential languages (C, Fortran, e.g.) by additional parallel language constructs
      - implementation via procedure calls from respective libraries
      - example: MPI, PVM, Linda
    - parallel programming environments
      - parallel programming interface plus additional tools such as compiler, libraries, debugger, ...
      - most (machine dependent) environments come along with a parallel computer
      - example: MPICH

# Terms and Definitions

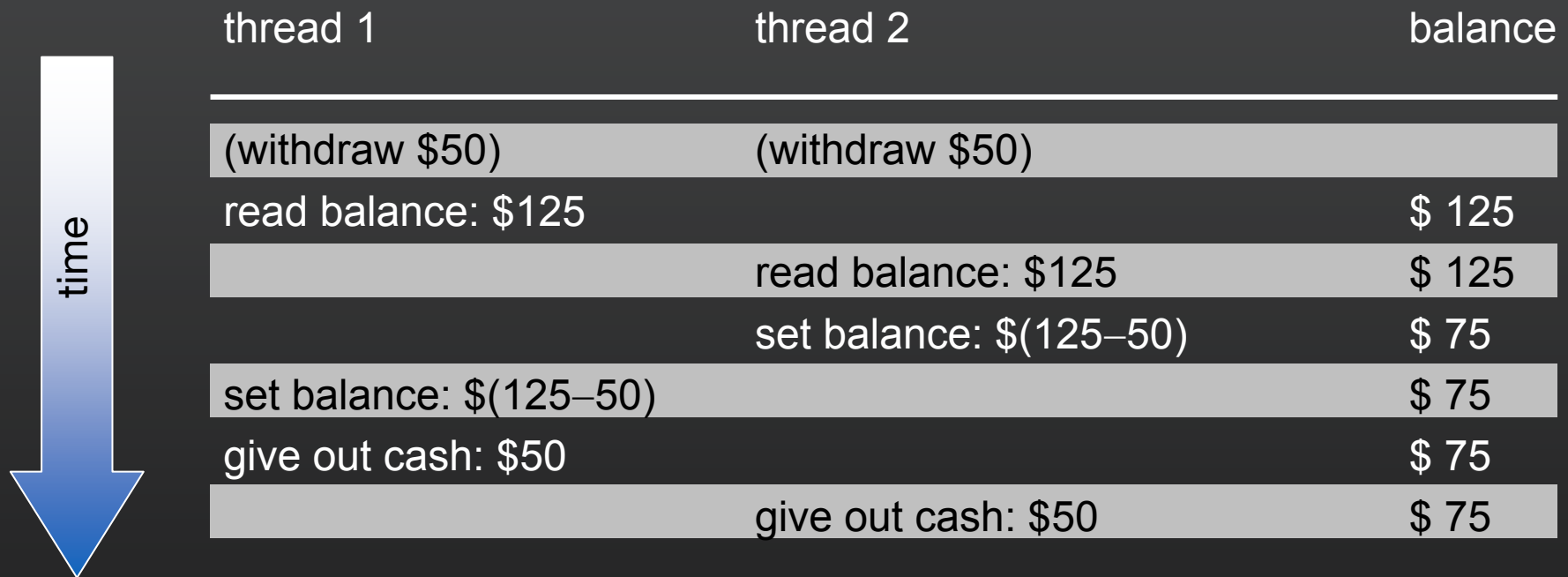
- **parallel programming languages (cont'd)**
  - implicit parallelism
    - mapping of programs (written in a sequential language) to the parallel computer via compiler directives
    - primarily for the parallelisation of loops
    - only minor modifications of source code necessary
    - level of parallelism
      - block level for parallelising compilers (→ threads)
      - instruction / sub-instruction level for vectorising compilers
  - example: OpenMP (parallelising), Intel compiler (vectorising)

# Overview

- terms and definitions
- **process interaction for MemMS**
- process interaction for MesMS
- example of a parallel program
- load balancing
- state-of-the-art LB strategies

# Process Interaction for MemMS

- **problem:** ATM race condition with two withdraw threads



# Process Interaction for MemMS

- **principles**
  - processes depend from each other if they have to be executed in a certain order; this can have two reasons
    - *cooperation*: processes execute parts of a common task
      - producer/consumer: one process generates data to be processed by another one
      - client/server: same as above, but second process also returns some data (result of a computation, e.g.)
    - *competition*: activities of one process hinder other processes
  - synchronisation: management of cooperation / competition of processes → ordering of processes' activities
  - communication: data exchange among processes
  - MemMS: realised via shared variables with read / write access

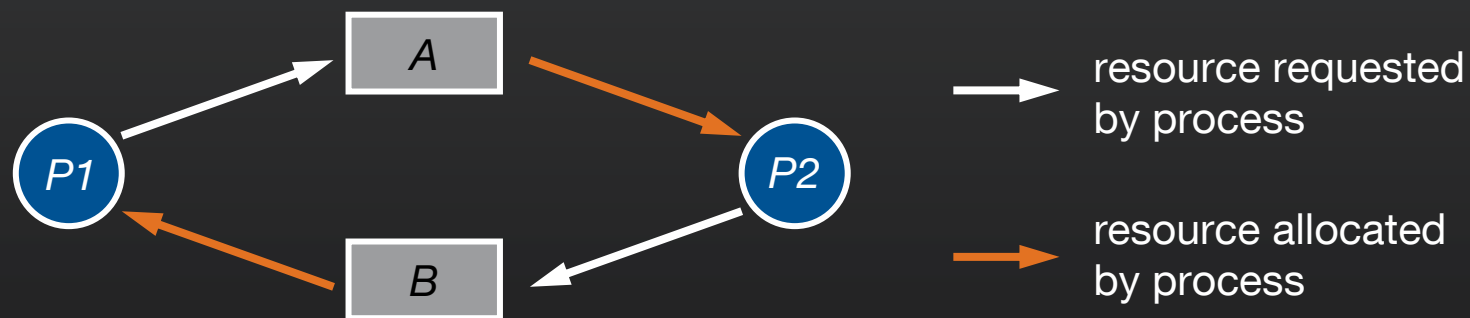


# Process Interaction for MemMS

- **synchronisation**
  - two types of synchronisation can be distinguished
    - *unilateral*: if activity *A2* depends on the results of activity *A1* then *A1* has to be executed before *A2* (i.e. *A2* has to wait until *A1* finishes); synchronisation does not affect *A1*
    - *multilateral*: order of execution of *A1* and *A2* does not matter, but *A1* and *A2* are not allowed to be executed in parallel (due to write / write or read / write conflicts, e.g.)
  - activities affected by multilateral synchronisation are *mutual exclusive*, i.e. they cannot be executed in parallel and act to each other atomically (no activity can interrupt another one)
  - instructions requiring mutual exclusion are called *critical sections*
  - synchronisation might lead to deadlocks (mutual blocking) or lockout (“*starvation*”) of processes, i.e. indefinable long delay

# Process Interaction for MemMS

- **synchronisation (cont'd)**
  - necessary and sufficient constraints for deadlocks
    - resources are only exclusively useable
    - resources cannot be withdrawn from a process
    - processes do not release assigned resources while waiting for the allocation of other resources
  - there exists a cyclic chain of processes that use at least one resource needed by the next processes within the chain



# Process Interaction for MemMS

- **synchronisation (cont'd)**
  - possibilities to handle deadlocks
    - deadlock detection
      - techniques to detect deadlocks (identification of cycles in waiting graphs, e.g.) and measures to eliminate them (rollback, e.g.)
    - deadlock avoidance
      - by rules: paying attention that at least one of the four constraints for deadlocks is not fulfilled
      - by requirements analysis: analysing future resource allocations of processes and forbidding states that could lead to deadlocks (HABERMANN's / banker's algorithm well known from OS, e.g.)

# Process Interaction for MemMS

- **methods of synchronisation**
  - lock variable / mutex
  - semaphore
  - monitor
  - barrier

# Process Interaction for MemMS

- **lock variable / mutex**
  - used to control the access to critical sections
  - when entering a critical section a process
    - has to wait until the respective lock is open, then enters and closes the lock, thus no other process can follow
    - opens the lock and leaves when finished
    - lock / unlock have to be executed from the same process
  - lock variables are abstract data types consisting of
    - a boolean variable of type mutex
    - at least two functions lock and unlock
    - further functions (Pthreads): init, destroy, trylock, ...
  - function lock consists of two operations “test” and “set” which together form a non interruptible (i.e. atomic) activity

# Process Interaction for MemMS

- **semaphore**
  - abstract data type consisting of
    - nonnegative variable of type integer (semaphore counter)
    - two atomic operations  $P$  (“*passeeren*”) and  $V$  (“*vrijgeven*”)
  - after initialisation of semaphore  $S$  the counter can only be manipulated with the operations  $P(S)$  and  $V(S)$ 
    - $P(S)$ : if  $S > 0$  then  $S \leftarrow S - 1$   
else the processes executing  $P(S)$  will be suspended
    - $V(S)$ :  $S \leftarrow S + 1$
  - after a  $V$ -operation any suspended process is reactivated (busy waiting); alternatives: always next process in queue
  - *binary semaphore*: has only values “0” and “1”
  - *general semaphore*: has any nonnegative number

# Process Interaction for MemMS

- **semaphore (cont'd)**
  - initial value of semaphore counter defines the maximum amount of processes that can enter a critical section simultaneously
  - critical section enclosed by operations  $P$  and  $V$

```
# mutual exclusion
(binary) semaphore  $s$ ;  $s \leftarrow 1$ 
execute  $p1()$  and  $p2()$  in parallel
```

```
begin procedure  $p1()$ 
  while (true) do
     $P(s)$ 
    enter_crit_section()
     $V(s)$ 
  od
end
```

```
begin procedure  $p2()$ 
  while (true) do
     $P(s)$ 
    enter_crit_section()
     $V(s)$ 
  od
end
```

# Process Interaction for MemMS

- **semaphore (cont'd)**
  - consumer/producer-problem: semaphore indicates difference between produced and consumed elements
  - assumption: unlimited buffer, atomic operations store and remove

```
# consumer/producer
(general) semaphore s; s ← 0
execute producer() and consumer() in parallel
```

```
begin procedure producer()
  while (true) do
    produce X
    store X
    V(s)
  od
end
```

```
begin procedure consumer()
  while (true) do
    P(s)
    remove X
    consume X
  od
end
```



# Process Interaction for MemMS

- **monitor**
  - semaphores solve synchronisation on a very low level → already one wrong semaphore operation might cause the breakdown of the entire system
  - better: synchronisation on a higher level with monitors
    - abstract data type with implicit synchronisation mechanism, i.e. implementation details (such as access to shared data or mutual exclusion) are hidden from the user
    - all access operations are mutual exclusive, thus all resources (controlled by the monitor) are only exclusively useable
  - monitors consist of
    - several monitor variables and monitor procedures
    - a monitor body (instructions executed after program start for initialisation of the monitor variables)

# Process Interaction for MemMS

- **monitor (cont'd)**
  - only access to monitor-bound variables via monitor procedures, direct access from outside the monitor is not possible
  - only one process can enter a monitor at each point in time, all others are suspended and have to wait outside the monitor
  - synchronisation via condition variables (based on mutex)
    - *wait(c)*: calling process is blocked and appended to an internal queue of processes also blocked due to condition *c*
    - *signal(c)*: if queue for condition *c* is not empty, the process at the queue's head is reactivated (and also preferred to processes waiting outside for entering the monitor)
  - condition variables are monitor-bound and only accessible via operations *wait* and *signal* (→ no manipulation from outside)

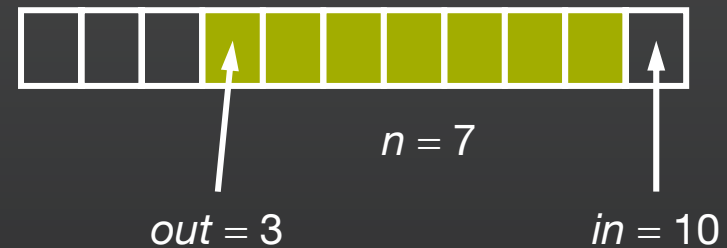
# Process Interaction for MemMS

- monitor (cont'd)
  - consumer/producer-problem with limited buffer (1)

```
# consumer/producer
const size ← ...

monitor limitedbuffer
    buffer[size] of integer
    in, out: integer
    n: integer
    notempty, notfull: condition

begin procedure store(X)
    if n = size then wait(notfull) fi
    buffer[in] ← X; in ← in + 1
    if in = size then in ← 0 fi
    n ← n + 1
    signal(notempty)
end
```



# Process Interaction for MemMS

- **monitor (cont'd)**
  - consumer/producer-problem with limited buffer (2)

```
begin procedure remove(X)
  if n = 0 then wait(notempty) fi
  X ← buffer[out]; out ← out + 1
  if out = size then out ← 0 fi
  n ← n - 1
  signal(notfull)
end

monitor body: in ← 0; out ← 0; n ← 0

begin procedure producer()
  while (true) do
    produce X
    store(X)
  od
end

begin procedure consumer()
  while (true) do
    remove(X)
    consume X
  od
end
```

# Process Interaction for MemMS

- **monitor (cont'd)**
  - some remarks
    - compared to semaphores, after once correctly programmed monitors cannot be disturbed by adding further processes
    - semaphores can be implemented via monitors and vice versa
    - nowadays, multithreaded OS use lock variables and condition variables instead of monitors
    - nevertheless, monitors are used within Java for synchronisation of threads

# Process Interaction for MemMS

- **barrier**
  - synchronisation point for several processes, i.e. each process has to wait until the last one also arrived
  - initialisation of counter  $C$  before usage with the amount of processes that should wait (init-barrier operation)
  - each process executes a wait-barrier operation
    - counter  $C$  is decremented by one
    - process is suspended if  $C > 0$ , otherwise all processes are reactivated and the counter  $C$  is set back to the initial value
  - useful for setting all processes (after independent processing steps) into the same state and for debugging purposes

# Process Interaction for MemMS

- simple case study



*“Program testing can be used to show the presence of bugs, but never to show their absence.”*

E.W. Dijkstra

# Process Interaction for MemMS

- **simple case study (cont'd)**
  - test case: reader-writer-problem
  - to be examined
    - *deadlock*: program will **never deadlock**
    - *mutual exclusion*: resource is **never used by both processes** at same time
    - *liveness*: resource **will eventually be used** by any process
- status variables
  - $x, pc_0, pc_1$
  - hence, 32 states possible (but 12 states not reachable)

```

boolean x ← 0

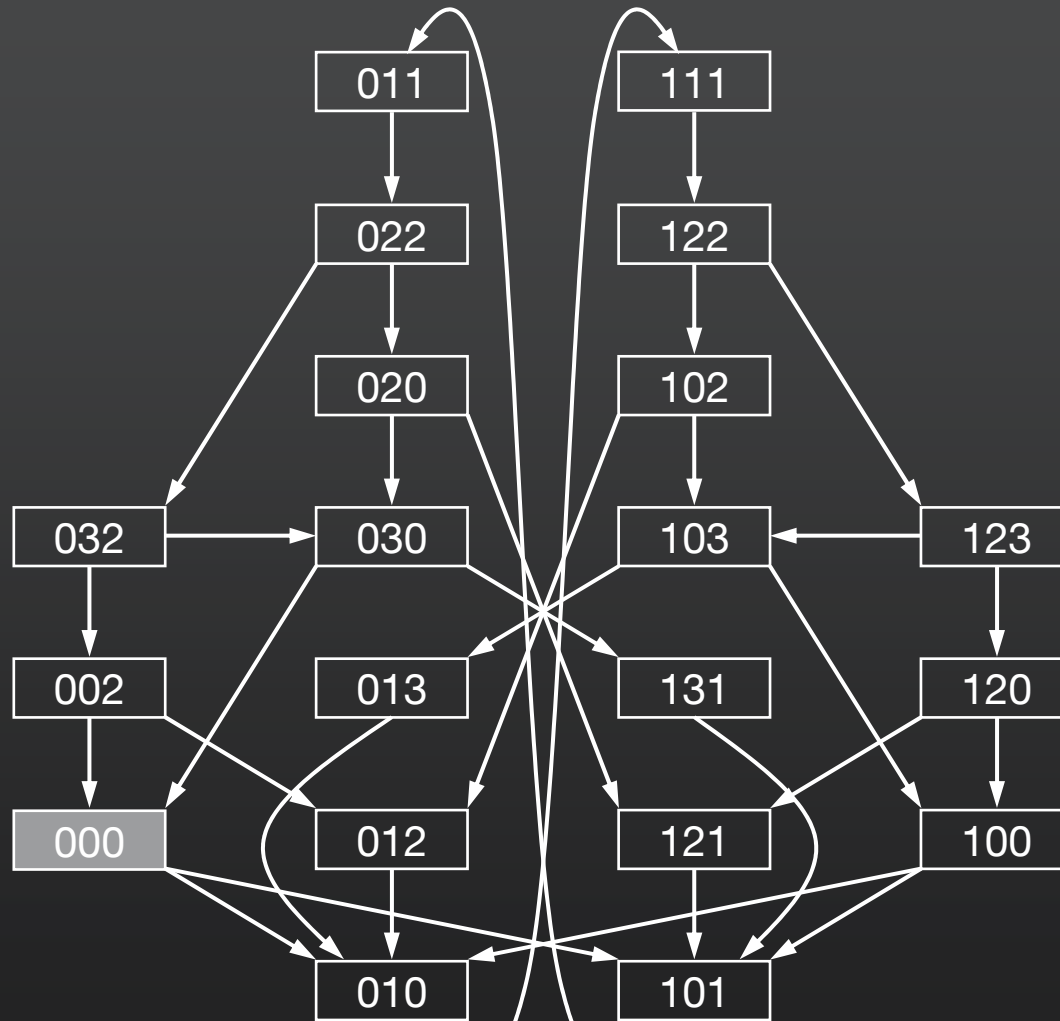
proc rw0 {
  while (true) {
0:     x ← 0
1:     sync ( )
2:     if (x = 0)
3:       use_resource
  }
}

proc rw1 {
  while (true) {
0:     x ← 1
1:     sync ( )
2:     if (x = 1)
3:       use_resource
  }
}
  
```

© S. Siegel, UD, USA



# Process Interaction for MemMS



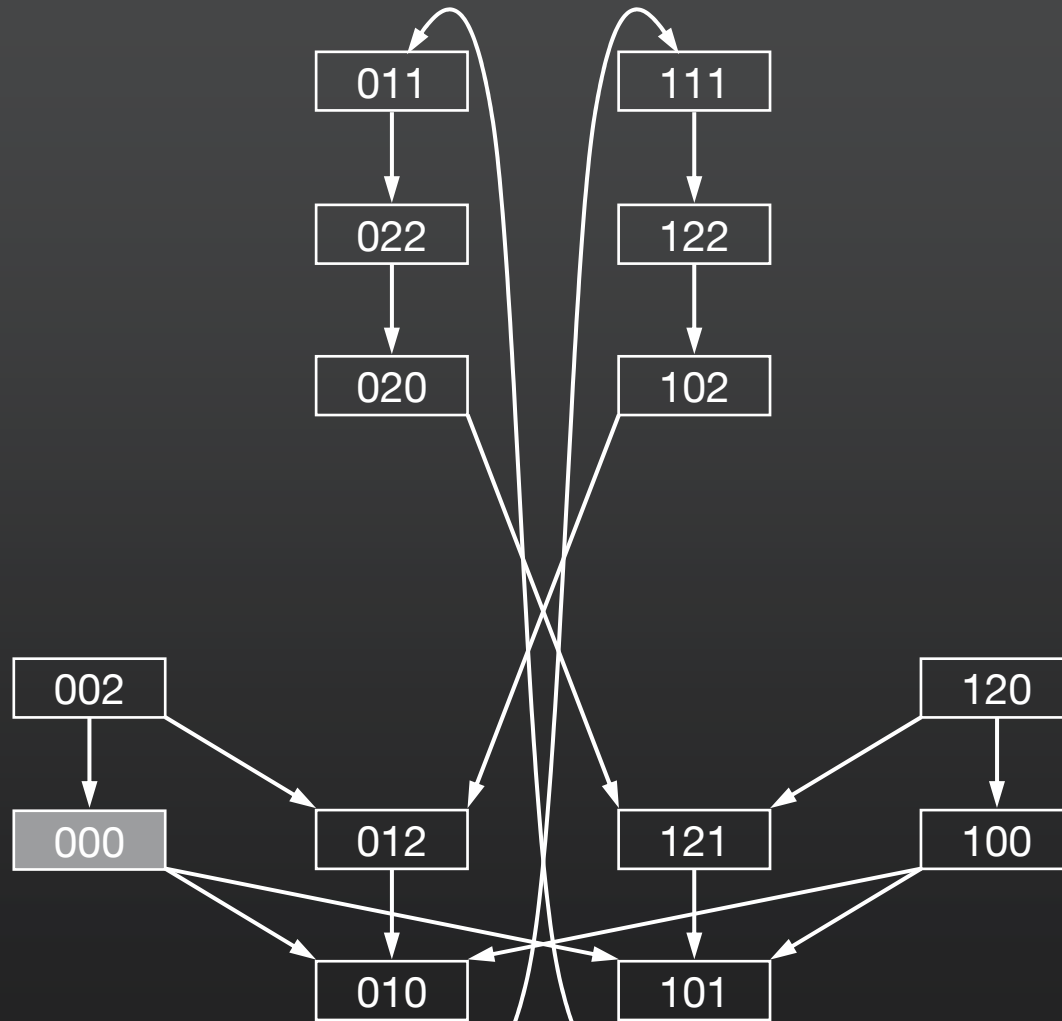
```

boolean x ← 0

proc rw0 {
  while (true) {
0:   x ← 0
1:   sync ( )
2:   if (x = 0)
3:     use_resource
  }
}

proc rw1 {
  while (true) {
0:   x ← 1
1:   sync ( )
2:   if (x = 1)
3:     use_resource
  }
}
  
```

# Process Interaction for MemMS



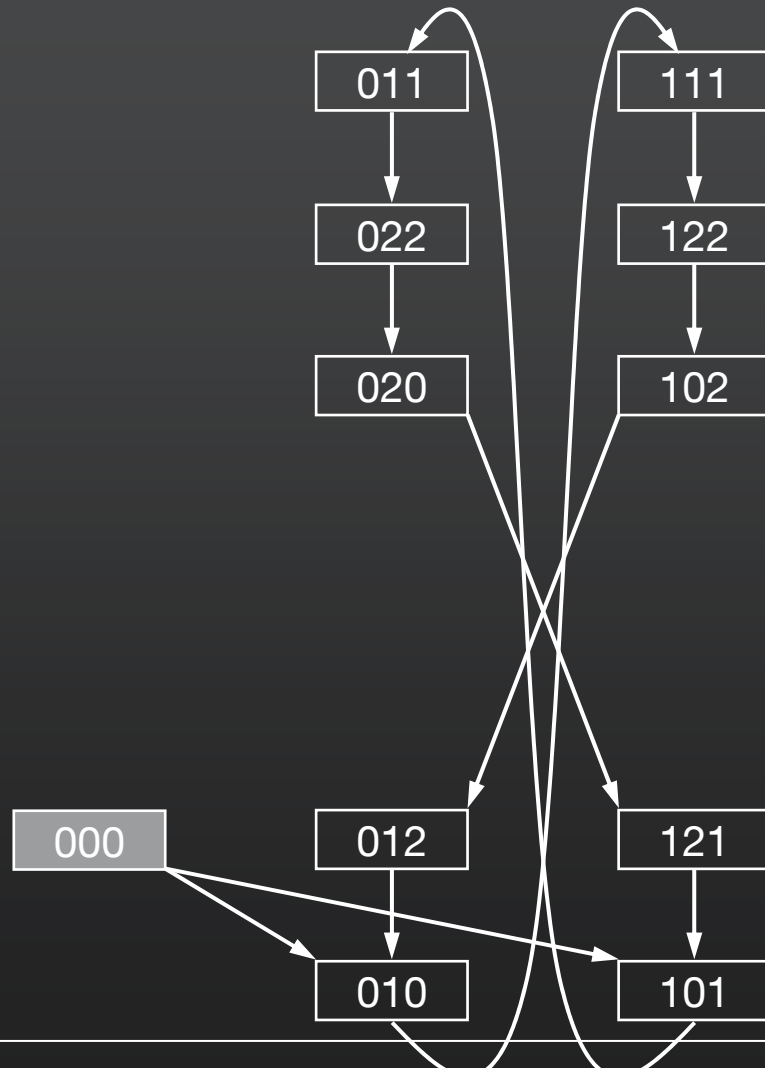
```

boolean x ← 0

proc rw0 {
  while (true) {
0:   x ← 0
1:   sync ( )
2:   if (x = 0)
3:     use_resource
  }
}

proc rw1 {
  while (true) {
0:   x ← 1
1:   sync ( )
2:   if (x = 1)
3:     use_resource
  }
}
  
```

# Process Interaction for MemMS



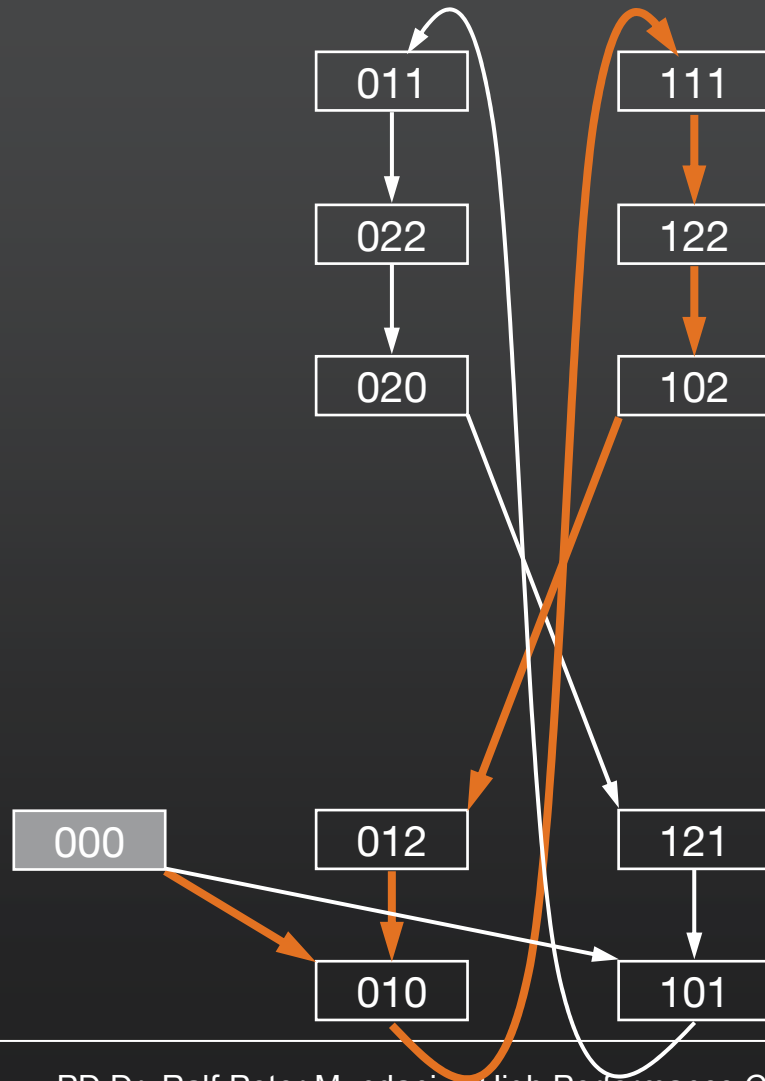
```

boolean x ← 0

proc rw0 {
  while (true) {
0:   x ← 0
1:   sync ( )
2:   if (x = 0)
3:     use_resource
  }
}

proc rw1 {
  while (true) {
0:   x ← 1
1:   sync ( )
2:   if (x = 1)
3:     use_resource
  }
}
  
```

# Process Interaction for MemMS



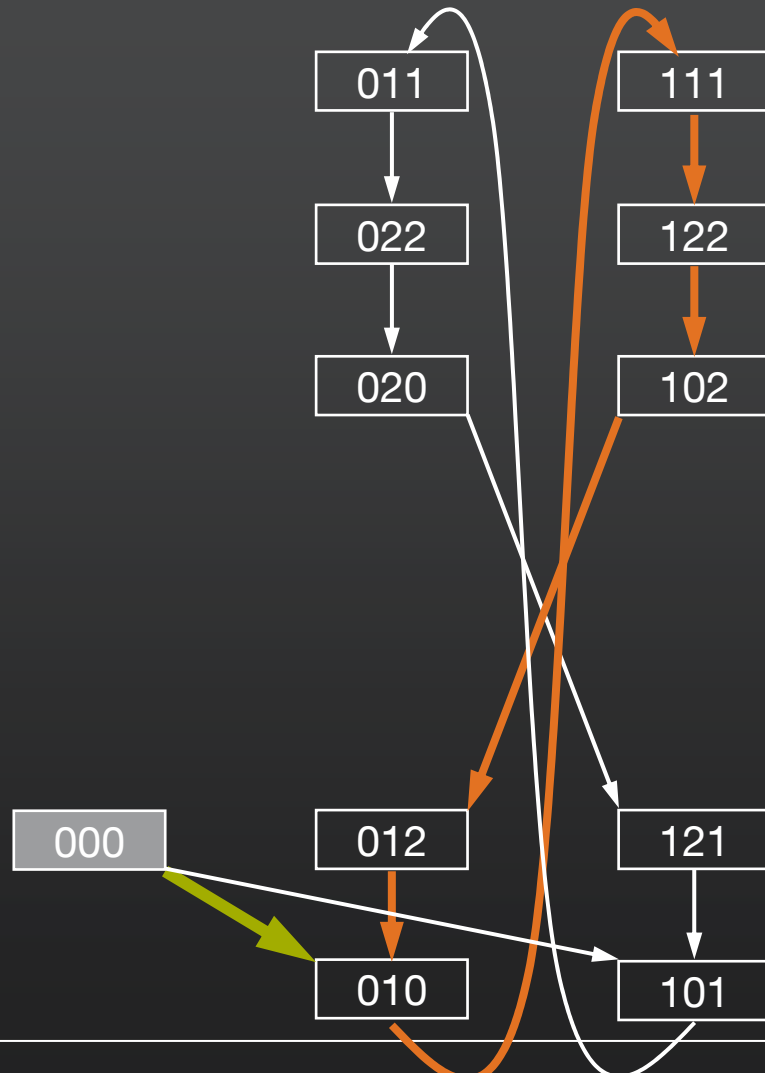
```

boolean x ← 0

proc rw0 {
  while (true) {
0:   x ← 0
1:   sync ( )
2:   if (x = 0)
3:     use_resource
  }
}

proc rw1 {
  while (true) {
0:   x ← 1
1:   sync ( )
2:   if (x = 1)
3:     use_resource
  }
}
  
```

# Process Interaction for MemMS



```

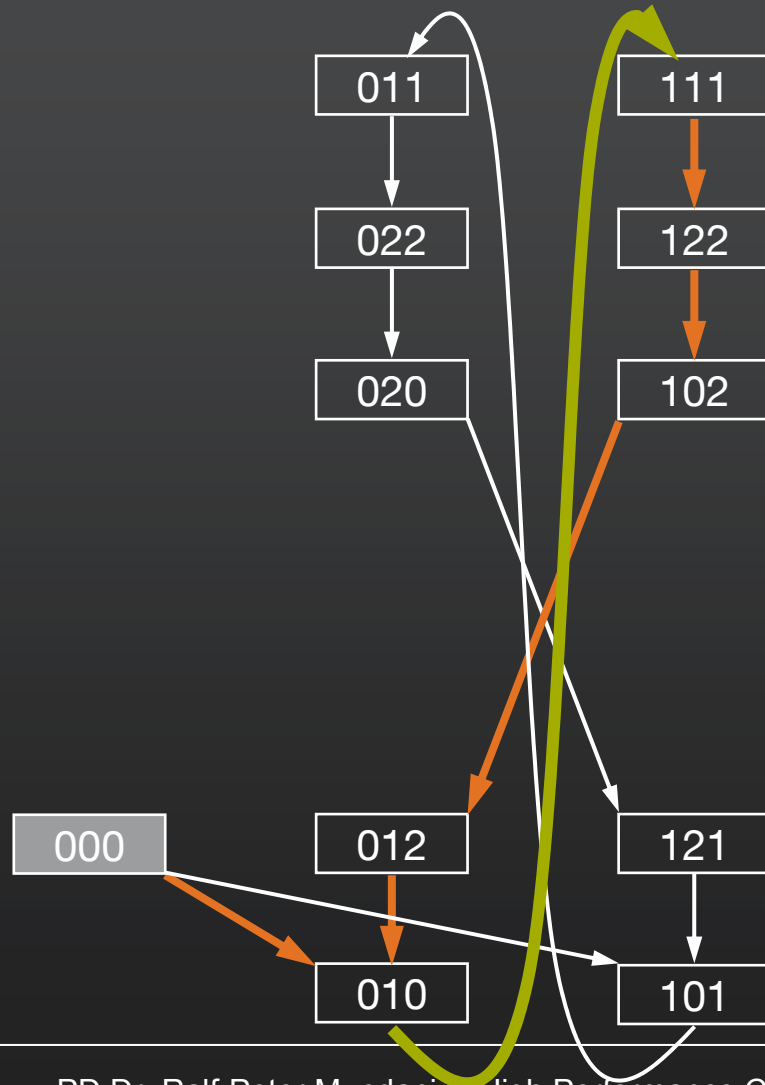
boolean x ← 0

proc rw0 {
  while (true) {
0:   x ← 0
1:   sync ( )
2:   if (x = 0)
3:     use_resource
  }
}

proc rw1 {
  while (true) {
0:   x ← 1
1:   sync ( )
2:   if (x = 1)
3:     use_resource
  }
}

```

# Process Interaction for MemMS



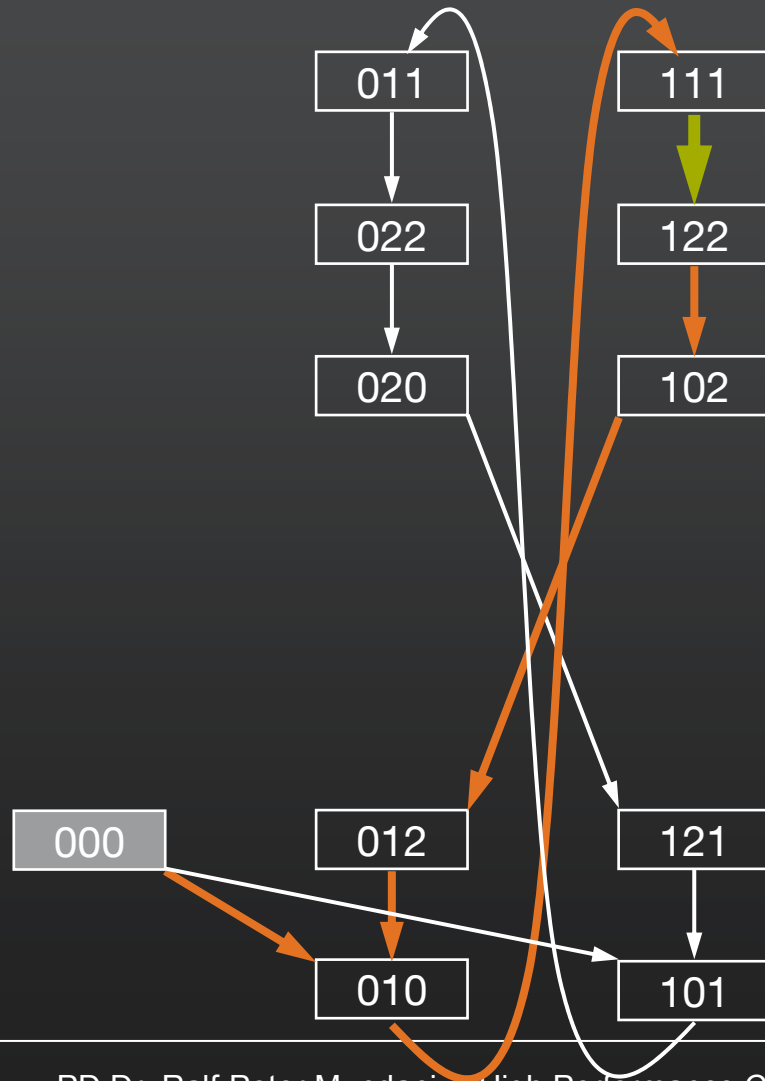
```

boolean x ← 0

proc rw0 {
  while (true) {
0:   x ← 0
1:   sync ( )
2:   if (x = 0)
3:     use_resource
  }
}

proc rw1 {
  while (true) {
0:   x ← 1
1:   sync ( )
2:   if (x = 1)
3:     use_resource
  }
}
  
```

# Process Interaction for MemMS



```

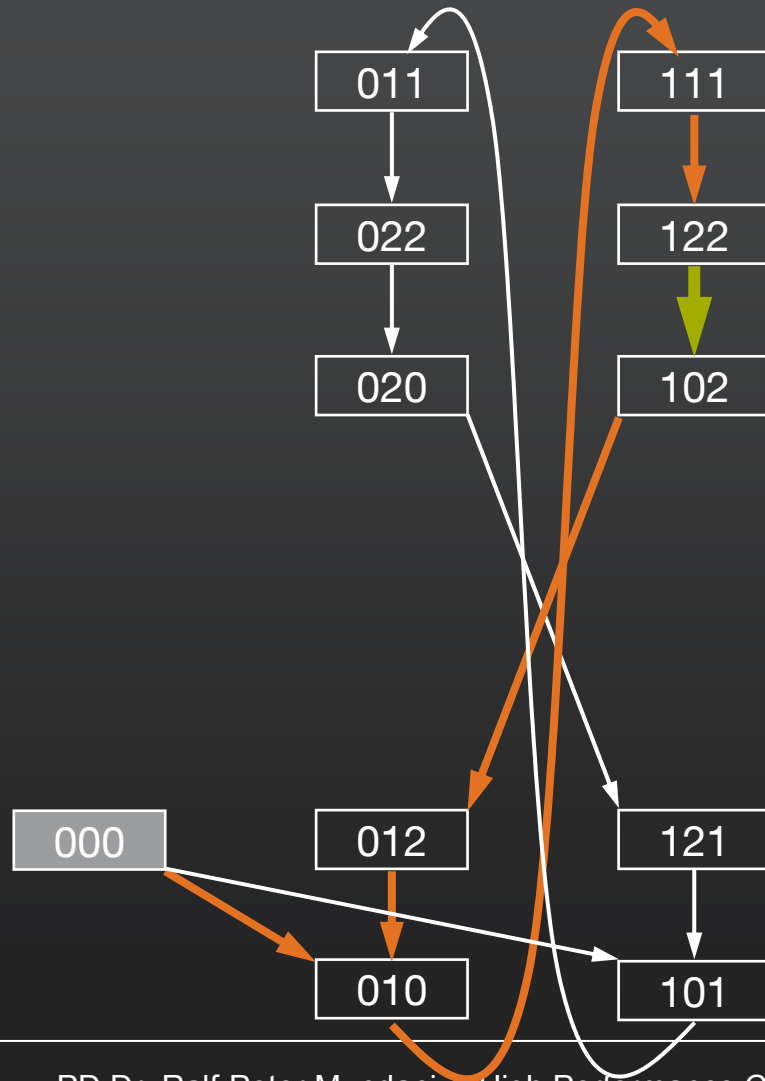
boolean x ← 0

proc rw0 {
  while (true) {
0:   x ← 0
1:   sync ( )
2:   if (x = 0)
3:     use_resource
  }
}

proc rw1 {
  while (true) {
0:   x ← 1
1:   sync ( )
2:   if (x = 1)
3:     use_resource
  }
}

```

# Process Interaction for MemMS



```

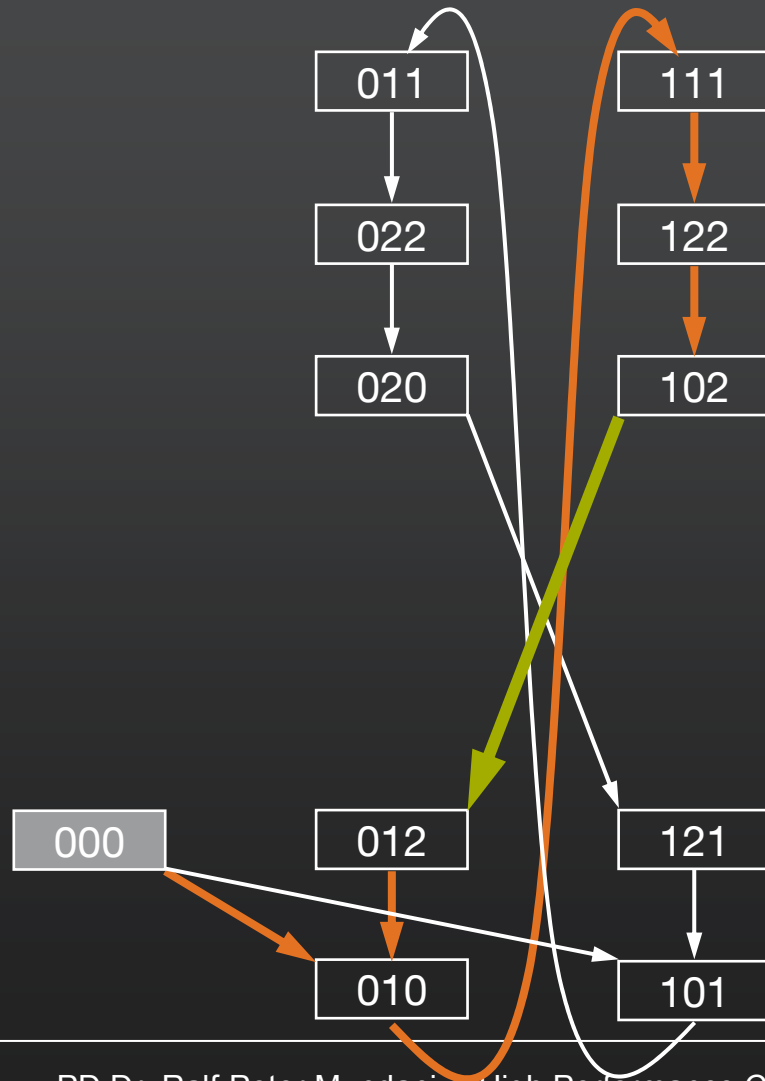
boolean x ← 0

proc rw0 {
  while (true) {
0:   x ← 0
1:   sync ( )
2:   if (x = 0)
3:     use_resource
  }
}

proc rw1 {
  while (true) {
0:   x ← 1
1:   sync ( )
2:   if (x = 1)
3:     use_resource
  }
}
  
```



# Process Interaction for MemMS



```

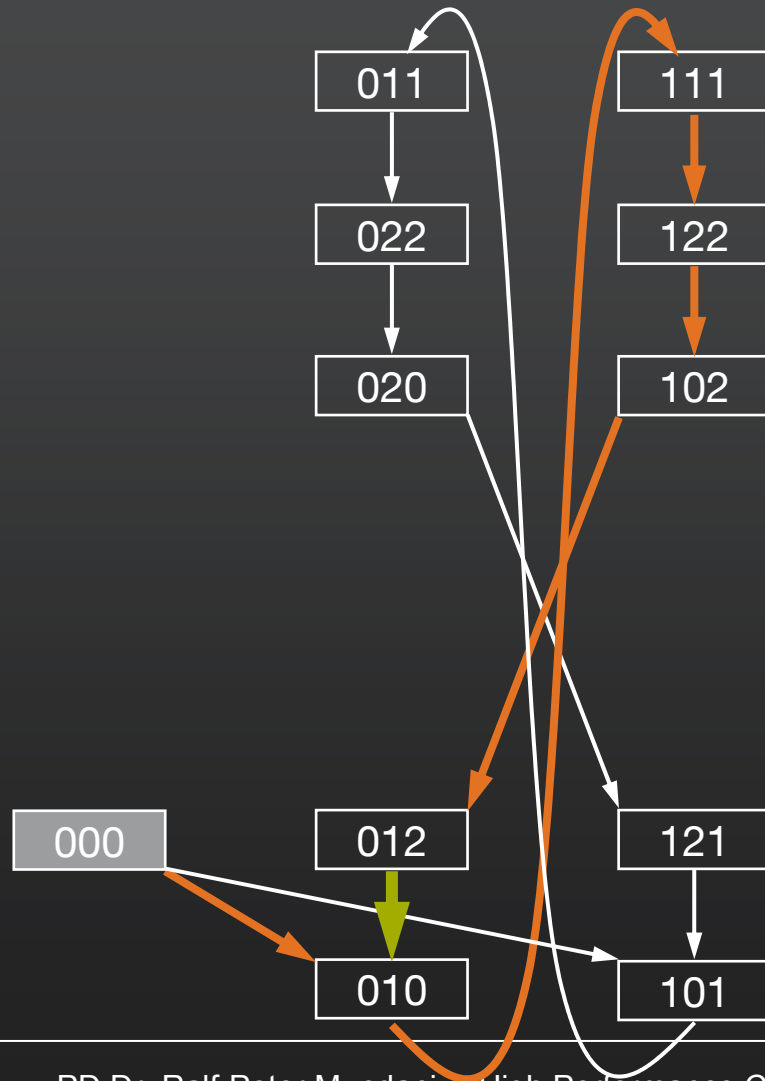
boolean x ← 0

proc rw0 {
  while (true) {
0:   x ← 0
1:   sync ( )
2:   if (x = 0)
3:     use_resource
  }
}

proc rw1 {
  while (true) {
0:   x ← 1
1:   sync ( )
2:   if (x = 1)
3:     use_resource
  }
}

```

# Process Interaction for MemMS



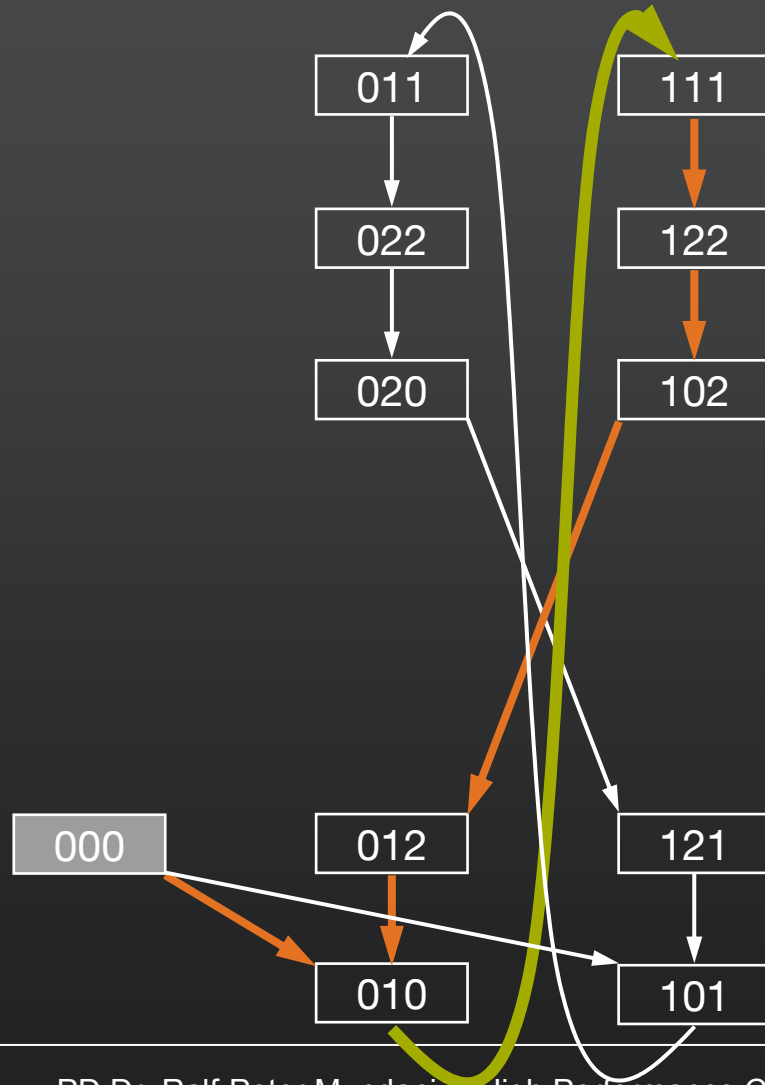
```

boolean x ← 0

proc rw0 {
  while (true) {
0:   x ← 0
1:   sync ( )
2:   if (x = 0)
3:     use_resource
  }
}

proc rw1 {
  while (true) {
0:   x ← 1
1:   sync ( )
2:   if (x = 1)
3:     use_resource
  }
}
  
```

# Process Interaction for MemMS



```

boolean x ← 0

proc rw0 {
  while (true) {
0:   x ← 0
1:   sync ( )
2:   if (x = 0)
3:     use_resource
  }
}

proc rw1 {
  while (true) {
0:   x ← 1
1:   sync ( )
2:   if (x = 1)
3:     use_resource
  }
}
  
```

# Overview

- terms and definitions
- process interaction for MemMS
- **process interaction for MesMS**
- example of a parallel program
- load balancing
- state-of-the-art LB strategies

# Process Interaction for MesMS

- **message passing paradigm**
  - no shared memory for synchronisation and communication
  - hence, transfer mechanism for information interchange necessary
  - message passing
    - messages: data units transferred between processes
    - send / receive operations instead of read / write operations
  - implicit (sequential) order during send / receive-stage
    - a message can only be received after a prior send
    - communication via message passing (independent from the transferred data) leads to an implicit synchronisation
    - synchronisation due to availability / unavailability of messages

# Process Interaction for MesMS

- **messages**
  - created whenever a process performs a send
  - necessary information to be provided from the sender
    - destination (process, node, communication channel, e.g.)
    - unique identifier of message (number, e.g.)
    - memory (so called send buffer) containing the data to be transferred with the message
    - data type and amount of elements within the send buffer
  - data type and amount of elements have to match for the receiver, otherwise a correct interpretation of the data is in doubt

# Process Interaction for MesMS

- **sending messages**
  - send operations can be
    - *synchronous / asynchronous*: a process performing a send is either blocked (synchronous) or not (asynchronous) until a respective receive operation is executed
    - *buffered / unbuffered*: a send operation may first copy the data from the send buffer to a system buffer (buffered) or directly perform the transfer (unbuffered → faster, but risk of overwriting the send buffer due to parallel execution of transfer (NIC) and next send operation (CPU) possible)
    - *blocking / non-blocking*: a send operation can either be blocked until the send buffer has been emptied (blocking) or immediately give control to the next instruction of the sending process (non-blocking → risk of overwriting send buffer)

# Process Interaction for MesMS

- **receiving messages**
  - a process has to specify which message to receive (via message identifier or wildcard) and where to store the data (so called receive buffer)
  - receive operations can be
    - *destructive / non-destructive*: a receive operation copies the data transferred with the message into the receive buffer and either destroys the message (destructive) or keeps it for later usage (non-destructive)
    - *synchronous / asynchronous*: a process performing a receive is either blocked until a message has arrived (synchronous) or not (asynchronous), thus it can continue with its execution and check again at a later point in time



# Process Interaction for MesMS

- **addressing modes**
  - different addressing modes can be distinguished
    - *direct naming*: process identifiers are used for sender and receiver → identifiers have to be known during development
    - *mailbox*: global memory where processes can store (send) and remove (receive) messages (used in the Distributed Execution and Communication Kernel (DECK), e.g.)
    - *port*: a port is bound to one process and can be used in one direction only, i.e. either for sending or receiving messages
    - *connection / channel*: for the usage of ports the setup of connections or channels is required, i.e. the connection of a send port of one process with the receive port of another process → data written from the sender to its port can be read from the receiver on the other port

# Overview

- terms and definitions
- process interaction for MemMS
- process interaction for MesMS
- **example of a parallel program**
- load balancing
- state-of-the-art LB strategies

# Example of a Parallel Program

- **labyrinth**
  - given: map of some labyrinth that contains
    - one entrance
    - one exit
    - no cycles



- task: determine if there exists a way from the entrance through the labyrinth to the exit (not the way itself)



# Example of a Parallel Program

- **labyrinth (cont'd)**
  - sequential algorithm (1)

```
position ← entrance
while (true) do
  switch (position) do
    case 'crossroads': position ← TURN_RIGHT()
    case 'dead end':   position ← TURN_BACK()
    case 'exit':       halt(OKAY)
    case 'entrance':   halt(ERROR)
  od
od
```

# Example of a Parallel Program

- labyrinth (cont'd)
  - sequential algorithm (2)

```

0#00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0#00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0#####00000000#####000.0000000#####00#000#####00000000
000000#00000#####00#000.0000000#00000000#00#0000+000#####00
000000#####00000#000.000#####00000000#####0000+000000000#00
000000+000000000000#000.000#000000+++0000+0000000+000000000#00
000000+000000000000#000.000###0000+00++++0000+++++000000000#00
000000+000000000000#000.00000#0000+00+0000000000000000000000#00
000000+000#####000.00000#0000+++++0000000++++++#####00
000000+000#000000000000.00000#0000000+0000000000000#00000000
000000+000#####000000000000000000000000000000+000000+000000#00000000
00+++++0000000000000000000000000000000000000000+000000+000000#00000000
000000+0000000000000000000000000000000000000000++++++000000#00000000
000000++++++0000000000000000000000000000000000000000000+000000+000000#00000000
0000000000000000000000000000000000000000000000000++++000000+000000#00000000
000000000000000000000000000000000000000000000000000000000#00000000

```

# Example of a Parallel Program

- **labyrinth (cont'd)**
  - parallelisation
    - basic questions
      - which parts to be parallelised
      - which structure: function or data parallelism
      - which model: shared or distributed memory

# Example of a Parallel Program

- **labyrinth (cont'd)**
  - variant A: *function parallelism*
    - start new processes at crossroads
    - terminate processes at dead ends
    - halt in case
      - one process has reached the exit
      - all processes have been terminated
  - questions
    - shared or distributed memory
    - total number of processes → queue of tasks
    - drawbacks

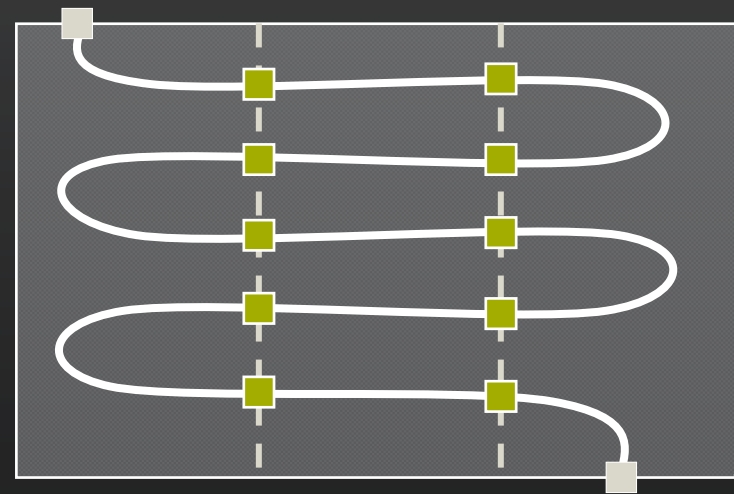
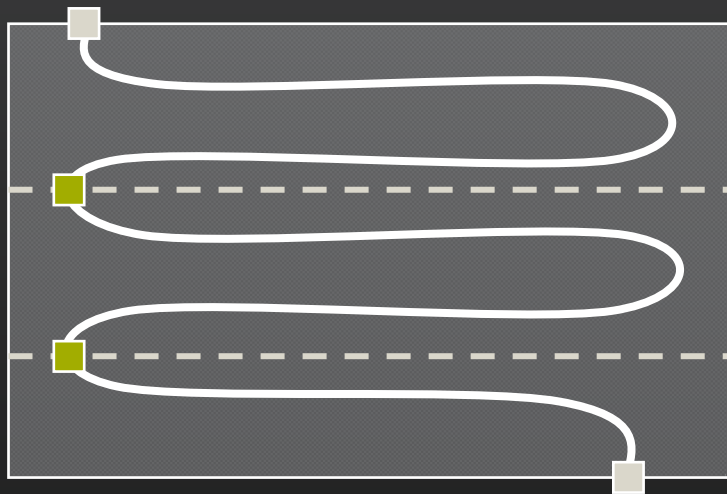


# Example of a Parallel Program

- **labyrinth (cont'd)**
  - variant B: *competitive parallelism*
    - start  $N$  processes following  $N$  different algorithms
    - first process reaching exit or entrance tells others to stop
    - possible algorithms
      - always go left instead of going right
      - start from the exit and try to reach the entrance
      - randomly walk around and remember all paths that have already been examined
      - ...
  - questions
    - shared or distributed memory
    - drawbacks

# Example of a Parallel Program

- labyrinth (cont'd)
  - variant C: *data parallelism* (1)
    - naïve approach
      - cut matrix into  $N$  parts
      - solve corresponding subproblems and check if there exist matching entrance-exit pairs → communication overhead



# Example of a Parallel Program

- **labyrinth (cont'd)**
  - variant C: *data parallelism* (2)
    - “better” approach
      - cut matrix  $A$  into parts  $A_i$  and distribute among processes
      - flood fill each part  $A_i$  with different “colors”
      - collect boundaries of all  $A_i$
      - check if (color) transition from entrance to exit is possible
    - questions
      - shared or distributed memory
      - drawbacks

# Overview

- terms and definitions
- process interaction for MemMS
- process interaction for MesMS
- example of a parallel program
- **load balancing**
- state-of-the-art LB strategies

# Load Balancing

- **motivation**
  - central issue: fairly distribution of computations across all processors / nodes in order to optimise
    - run time (user's point of view)
    - system load (computing centre's point of view)
- **problem**
  - amount of work is often not known prior to execution
  - load situation changes permanently (adaptive mesh refinement within numerical simulations, I/O, searches, ...)
  - different processor speeds (heterogeneous systems, e.g.)
  - different latencies for communication (grid computing, e.g.)
- **objective: load distribution or load balancing strategies**

# Load Balancing

- **static approaches**
  - to be applied before the execution of any process (in contrast to dynamic load balancing to be applied during the execution)
  - usually referred to as mapping problem or scheduling problem
  - potential static load-balancing techniques
    - *round robin*: assigning tasks (more general formulation than work to cover both data and function parallelism) in sequential order to processes, coming back to the first when all processes have been given a task
    - *randomised*: selecting processes at random to assign tasks
    - *recursive bisection*: recursive division into smaller subproblems of equal computational effort with less communication costs

# Load Balancing

- **dynamic approaches**
  - division of tasks dependent upon the execution of parts of the program as they are being executed → entails additional overhead (to be kept as small as possible, else bureaucracy wins)
  - assignment of tasks to processes can be classified as
    - *centralised*
      - tasks are handed out from a centralised location
      - within a master-slave structure one dedicated master process directly controls each of a set of slave processes
    - *decentralised*
      - tasks are passed between arbitrary processes
      - worker processes interact among themselves → a worker process may receive / send tasks from / to others

# Load Balancing

- **diffusion model (a. k. a first order scheme)**
  - analogy to physical processes in nature (salt or ink in water, e.g.)
  - original algorithm introduced by CYBENKO (1989) for static network topologies, meanwhile it has been often studied and derived (second order scheme, dynamic network topologies, e.g.)
  - *idea*: a process  $P_i$  balances its load simultaneously with all its neighbours  $N(i)$   $\rightarrow$  ratio  $\alpha_{ij}$  of the load difference between process  $P_i$  and  $P_j$  is swapped between them according to

$$w_i^{(t+1)} = w_i^{(t)} - \sum_{j \in N(i)} \alpha_{ij} \cdot (w_i^{(t)} - w_j^{(t)}), \quad 1 \leq i \leq p, \quad -1 < \alpha_{ij} < 1$$

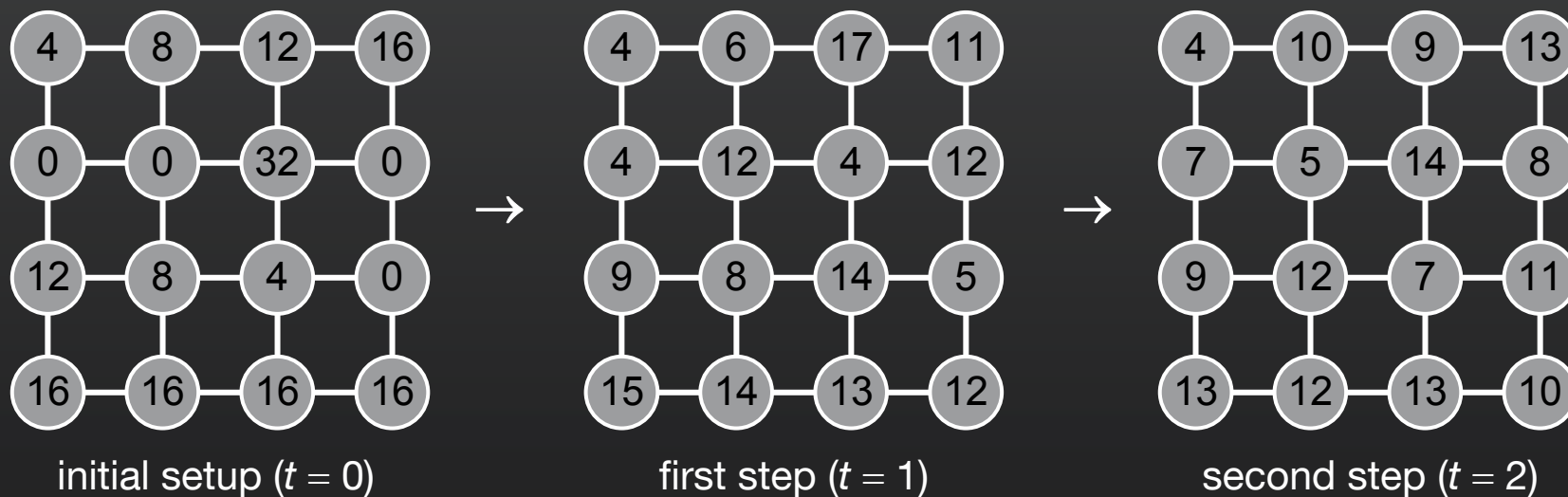
where  $w_i^{(t)}$  is the workload done by process  $P_i$  at time  $t$

- various methods to be found that determine parameter  $\alpha_{ij}$  such as
  - optimal choice: needs global knowledge of the network
  - BOILLAT choice: needs only local knowledge of the neighbours



# Load Balancing

- **diffusion model (cont'd)**
  - update of workload can be done
    - a) after all balancing factors have been computed (JACOBI-like)
    - b) during computation of balancing factors (GAUSS-SEIDEL-like)
  - example: first two iteration steps with  $\alpha = 0.25$  for JACOBI method



# Load Balancing

- **bidding (economic model)**
  - analogy to mechanisms of price fixing in markets
  - idea
    - process (with high workload) advertises tasks to its neighbours
    - neighbours submit their free resources as bid
    - process with highest bid (i.e. largest free resources) wins
  - remarks
    - maybe several rounds of bidding necessary → successively extending the range of bidders
    - in case of sudden workload peaks, a process might reject the purchased tasks
    - processes with free resources are still allowed to ask for tasks
  - drawback: quite complex analysis of this model

# Load Balancing

- **precalculation of the load**
  - all strategies so far are based on local information only
  - hence, load balancing is often quite expensive since (from a global point of view) balancing steps not always lead to a better load distribution among the processors
  - idea
    - global determination of the workload at program start or at certain points in time
    - global determination of an appropriate load distribution
    - workload transfer with less communication
  - developed and used for hierarchical network topologies → workload recording and load balancing between child and parent nodes

# Overview

- terms and definitions
- process interaction for MemMS
- process interaction for MesMS
- example of a parallel program
- load balancing
- **state-of-the-art LB strategies**

# State-of-the-art LB Strategies

- **space-filling curves**
  - origin of the idea: analysis and topology (“topological monsters”)
  - nice example of a construct from pure mathematics that gets practical relevance only decades later
  - definition of a space filling curve (SFC)
    - curve: image of a continuous mapping  $f: [0,1] \rightarrow [0,1]^D$
    - SFC: continuous, surjective mapping  $f: [0,1] \rightarrow [0,1]^D$  that covers an area (with a JORDAN content) greater than zero
  - prominent representatives
    - HILBERT’s SFC (1891): most famous SFC
    - PEANO’s SFC (1890): oldest SFC
    - LEBESGUE’s SFC: most important SFC for computer science
  - further reading: M. Bader, *Space-Filling Curves*, Springer (2012)

# State-of-the-art LB Strategies

- **HILBERT's space filling curve**

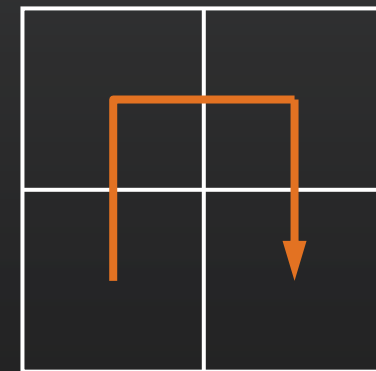
- for reasons of simplicity only in 2D  $\rightarrow f: I = [0,1] \rightarrow [0,1]^2 = Q$
- construction of SFC follows the geometric conception

*If  $I$  can be mapped onto  $Q$  in the space filling sense, then each of the four congruent subintervals of  $I$  can be mapped to one of the four quadrants of  $Q$  in the space filling sense, too.*

- recursive application of above preserves
  - *neighbourhood relations*: neighbouring subintervals in  $I$  are mapped onto neighbouring subsquares of  $Q$
  - *subset relations (inclusion)*: from  $I_1 \subseteq I_2$  follows  $f(I_1) \subseteq f(I_2)$
- border case: HILBERT's SFC

# State-of-the-art LB Strategies

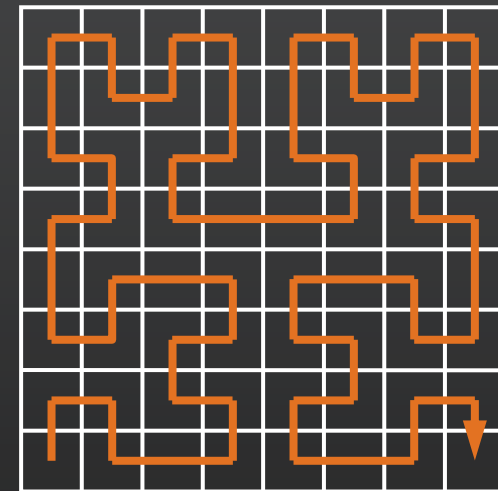
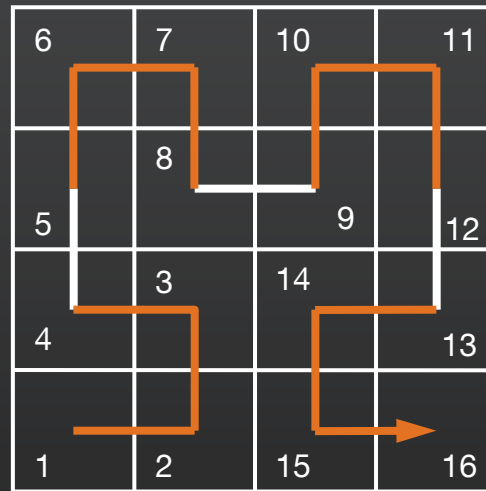
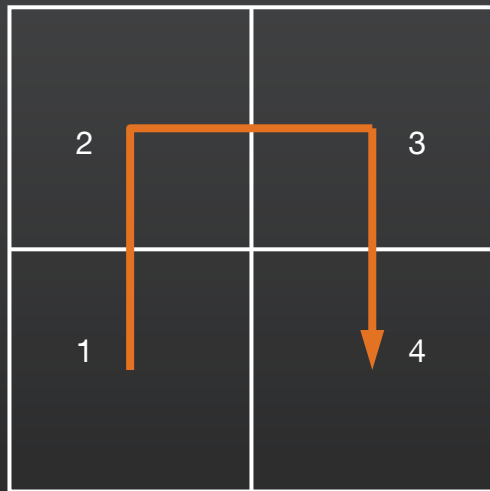
- **HILBERT's space filling curve (cont'd)**
  - correspondence of nested intervals in  $I$  and nested squares in  $Q$  provides pairs of points in  $I$  with corresponding image points in  $Q$
  - of course, the iterative steps in this generation process are of practical relevance, not the border case itself
    - 1) starting with a generator or "Leitmotiv" that defines the order in which the subsquares are visited
    - 2) applying generator in each subsquare (with appropriate similarity transformations if necessary)
    - 3) connecting the open ends



generator for HILBERT's SFC

# State-of-the-art LB Strategies

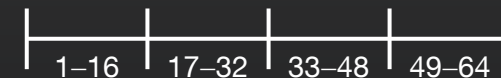
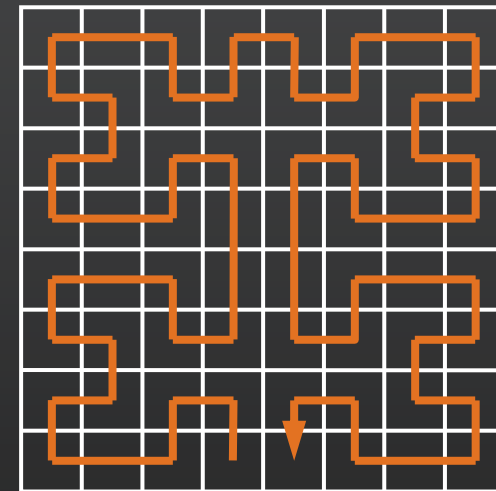
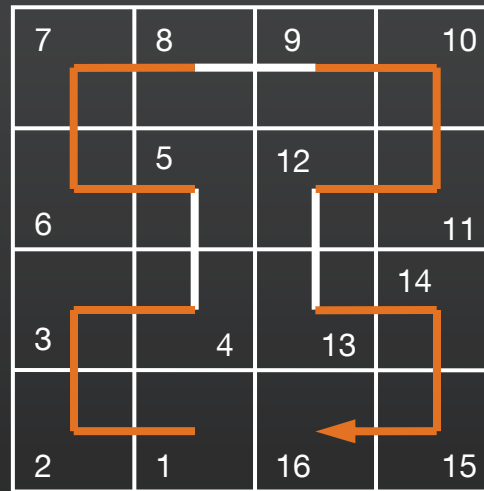
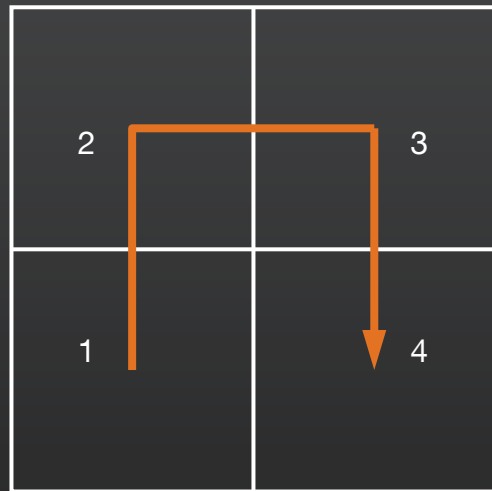
- HILBERT's space filling curve (cont'd)
  - classical version of HILBERT





# State-of-the-art LB Strategies

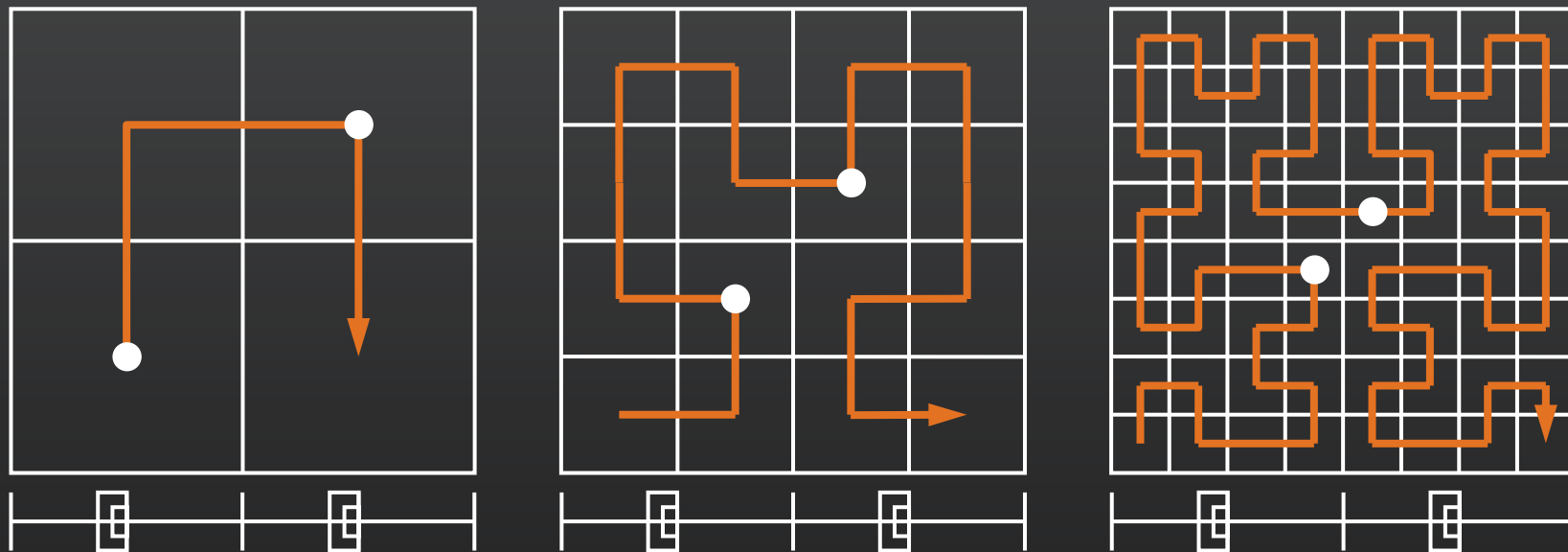
- HILBERT's space filling curve (cont'd)
  - variant of MOORE



- modulo symmetry, these are the only two possibilities

# State-of-the-art LB Strategies

- HILBERT's space filling curve (cont'd)
  - all iterations are injective, but HILBERT's SFC itself is not injective (there are image points with more than one original point in  $I$ )

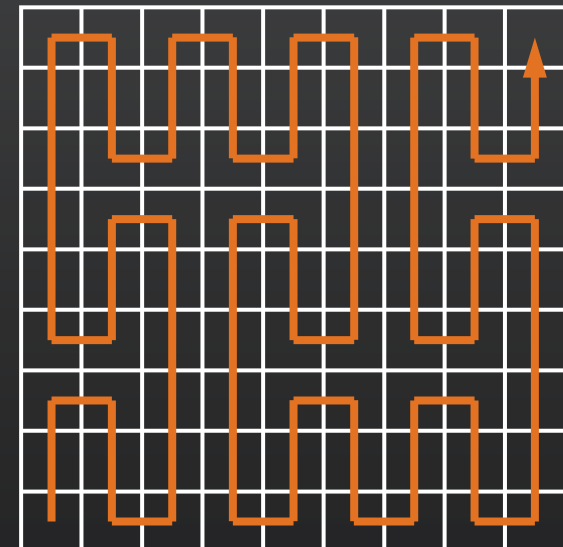
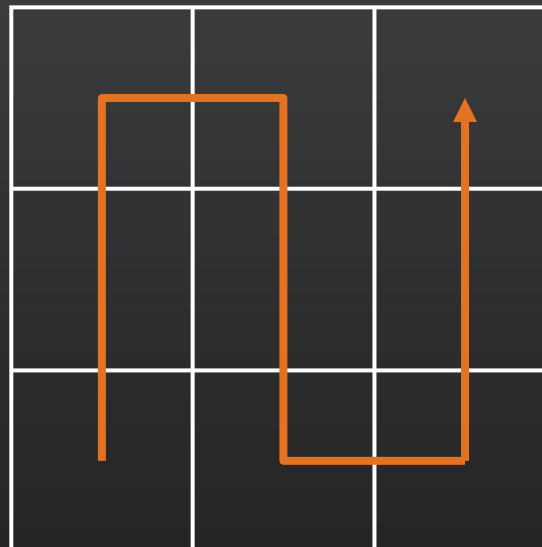


note: there exists a bijective mapping between two finite-dimensional smooth manifolds (CANTOR, 1878), but it cannot be both bijective and continuous (NETTO, 1879)

# State-of-the-art LB Strategies

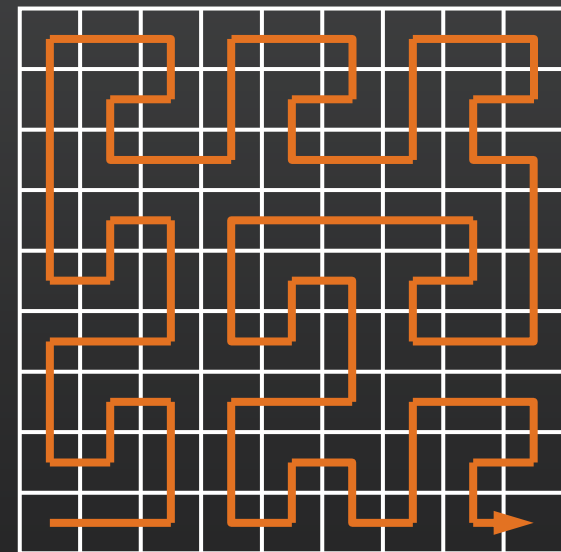
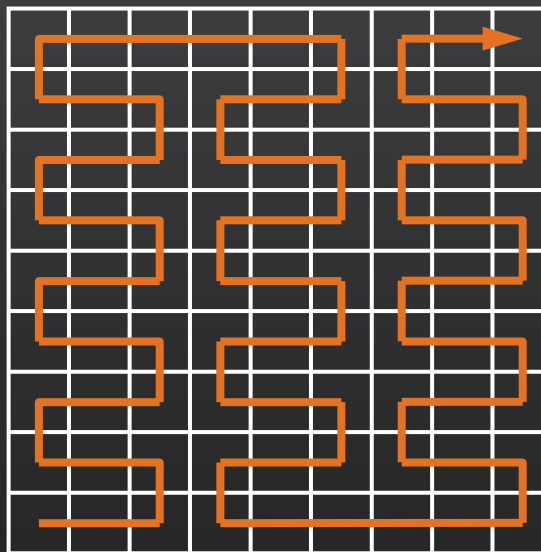
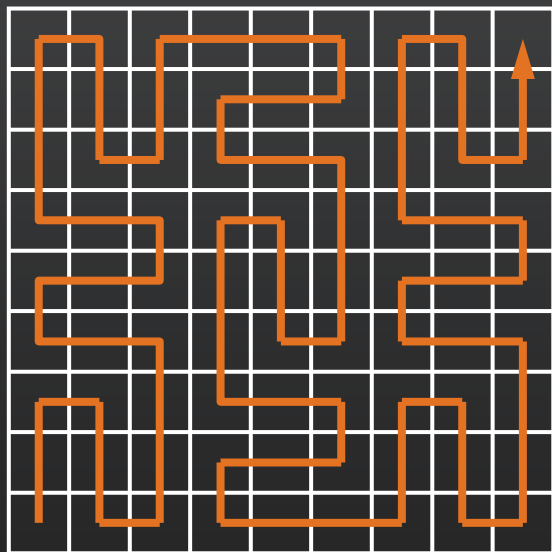
- **PEANO's space filling curve**
  - ancestor of all SFCs
  - subdivision of  $I$  and  $Q$  into nine congruent subdomains
  - definition of a generator, again, defines the order of visit

3	4	9
2	5	8
1	6	7



# State-of-the-art LB Strategies

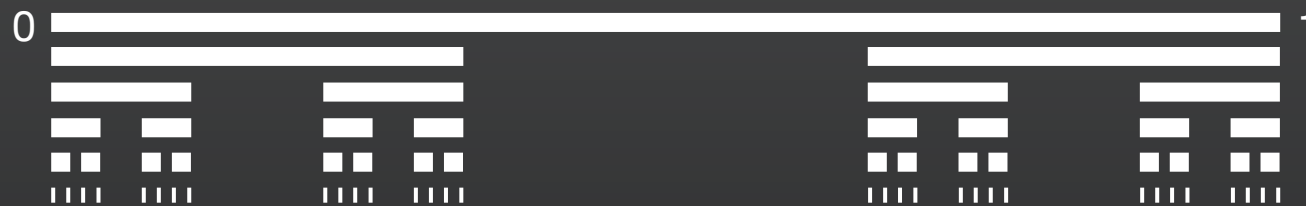
- PEANO's space filling curve (cont'd)
  - now, there are (modulo symmetry) 273 different possibilities to recursively apply the generator preserving neighbourhood and inclusion



serpentine type (left and centre) and meander type (right)

# State-of-the-art LB Strategies

- **LEBESGUE's space filling curve**
  - definition of LEBESGUES's SFC by the CANTOR set
  - CANTOR set  $C$ : repeatedly deleting the open middle thirds of  $[0,1]$



- $C$  is defined as set of points not excluded, hence the remaining interval can be computed by the total length removed

$$\sum_{N=0}^{\infty} \frac{2^N}{3^{N+1}} = \frac{1}{3} + \frac{2}{9} + \frac{4}{27} + \frac{8}{81} + \dots = \frac{1}{3} \cdot \left( \frac{1}{1 - \frac{2}{3}} \right) = 1$$

- the proportion of the remaining interval seems to be  $1 - 1 = 0$ , but in fact  $C$  has the same cardinality as the unit interval  $[0,1]$  (!)

# State-of-the-art LB Strategies

- **LEBESGUE's space filling curve (cont'd)**
  - nested intervals of  $C$  to be represented by ternary numbers of the form  $0_3.w_1w_2w_3\dots$  with  $w_i \in \{0, 1, 2\}$



- example: parameter  $T = 2/9$  can be written as  $[0, 1]$ ,  $[0_3.0, 0_3.1]$ ,  $[0_3.02, 0_3.10]$ ,  $[0_3.020, 0_3.021]$ ,  $[0_3.0200, 0_3.0201]$ , ...



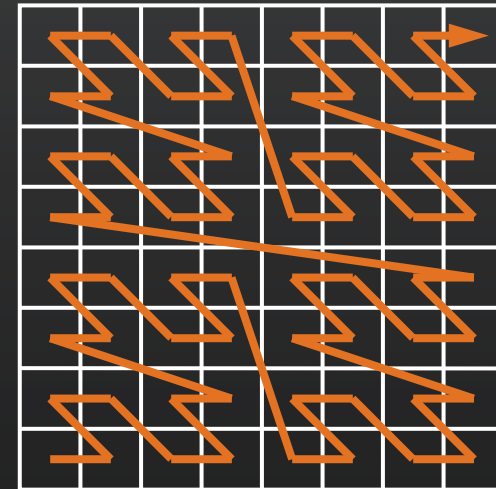
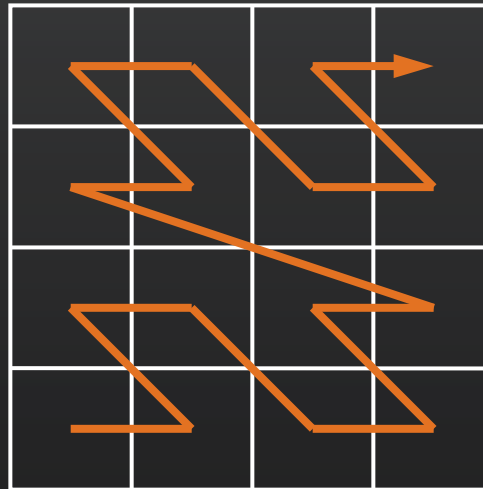
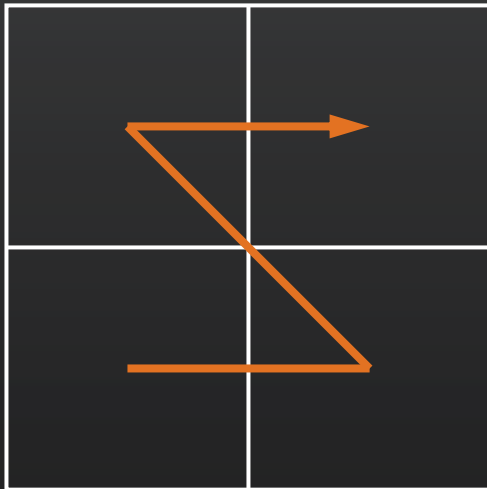
- since all interval borders can be written in two different ways ( $1_3.0$  or  $0_3.222\dots$ , e.g.) and the middle third ( $[0_3.1, 0_3.2]$ , e.g.) is repeatedly deleted, the CANTOR set only contains ternary numbers that consist of "0" and "2"

# State-of-the-art LB Strategies

- LEBESGUE's space filling curve (cont'd)
  - when mapping  $C$  to  $[0,1]^2$  according to

$$f : \left( \frac{0_3 \cdot w_1 w_2 w_3 w_4 \dots}{2} \right) = \begin{pmatrix} 0_2 \cdot x_1 x_3 \dots \\ 0_2 \cdot y_2 y_4 \dots \end{pmatrix}$$

and connecting the image points via linear interpolation, this results to LEBESGUE's SFC also referred to as "Z-order"



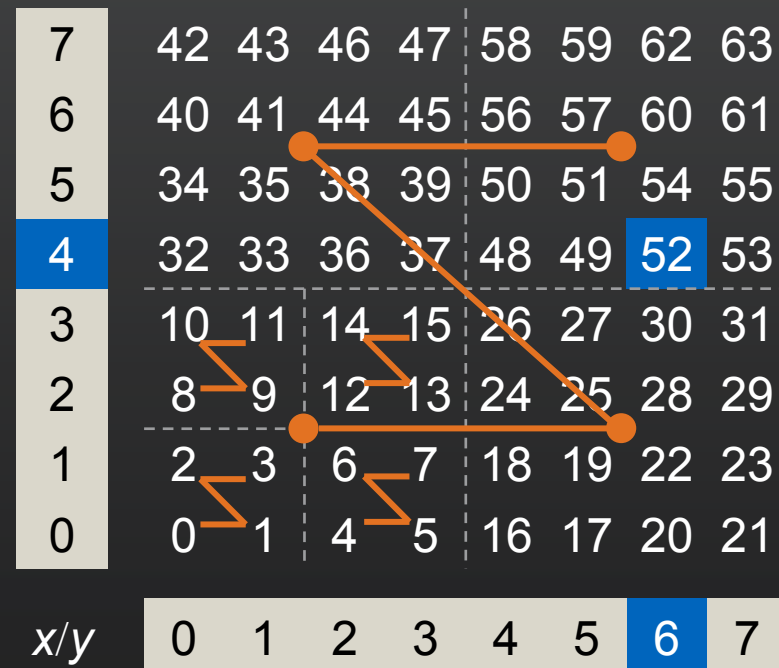
# State-of-the-art LB Strategies

- **LEBESGUE's space filling curve (cont'd)**
  - Z-ordering is well-known from quadtrees and octrees when linearising a tree by a depth-first traversal (provides a common naming scheme for cells → lexicographic or MORTON index)
  - bitwise interleaving of coordinate values (x, y) leads to Z-value
  - useful for multidimensional range searches, e.g.

x: 02.100 → 4

y: 02.110 → 6

Z: 02.110100 → 52





# State-of-the-art LB Strategies

- **applications**
  - sequentialisation of multidimensional data to one dimension while preserving locality
    - data are stringed sequentially like pearls
    - neighbouring points in the unit interval  $[0,1]$  have neighbouring images in  $[0,1]^D$
  - important applications such as
    - efficient multidimensional range searches in databases (Oracle, e.g.)
    - multi-particle or  $N$ -body problems
    - adaptive grid refinement for partial differential equations
    - dynamic load balancing

# State-of-the-art LB Strategies

- **load balancing**

- idea

- 1) assign points  $q_i$  of some iteration of an SFC (i.e.  $q_i \in Q$ ) to points  $s_i$  in the  $D$ -dimensional space
- 2) linearly order points  $q_i$  according to SFC's original points  $p_i \in I$
- 3) simple partitioning based on this sequential order of points  $p_i$  (while preserving locality) to processors possible

