High Performance Computing – Programming Paradigms and Scalability

PD Dr. rer. nat. habil. Ralf-Peter Mundani
Computation in Engineering / BGU
Scientific Computing in Computer Science / INF

Summer Term 2018

Part 3: Foundations

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

- overview
  - terms and definitions
  - process interaction on UMA / NUMA architectures
  - process interaction on NORMA architectures
  - putting everything together: an example
  - load balancing
  - state-of-art: space-filling curves

Terms and Definitions

- algorithmic analysis
  - sequential algorithms are characterised that way
    - all instructions U are processed in a certain sequence
    - this sequence is given due to the causal ordering of U, i.e. the causal dependencies from another instructions' result
  - hence, for set U a partial order $\leq$ can be declared
    - $x \leq y$ for $x, y \in U$
    - $\leq$ representing a reflexive, antisymmetric, transitive relation
  - example (a, b of type integer)

    I1: $a \leftarrow a - b$

    I2: $b \leftarrow b + a$

    I3: $a \leftarrow b - a$

    partial order: I1 $\leq$ I2 $\leq$ I3

what does this program compute?

Terms and Definitions

- algorithmic analysis (cont'd)
  - often, for $(U, \leq)$ more than one sequence can be found so that all computations (on the monoprocessor) are executed correctly
  - example

I1: $x \leftarrow a + b$		I2: $y \leftarrow c * c$
I2: $y \leftarrow c * c$	partial order: I1, I2 $\leq$ I3	I1: $x \leftarrow a + b$
I3: $z \leftarrow x - y$		I3: $z \leftarrow x - y$

  - typical questions arise
    - which part of the program can be done in parallel
    - what kind of structure to be used for parallelisation
    - what kind of compiler to be used
    - what about load balancing strategies
    - …

Terms and Definitions

- dependence analysis
  - (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 i ← 0 to N do
   a[i] ← i + 1
od
```

```
for i ← 1 to N do
   x ← 2*i + 3
   a[i] ← a[x]
od
```

  - as dependencies are not always obvious, an algorithmic / automated way of recognising those (e.g. via the compiler) would preferable

Terms and Definitions

- dependence analysis (cont'd)
  - BERNSTEIN (1966) established a set of conditions, sufficient for determining whether two instructions 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

$$I_1 \cap O_2 = \varnothing \qquad I_2 \cap O_1 = \varnothing \qquad O_1 \cap O_2 = \varnothing$$

  - example

I1: $a \leftarrow x + y$ I2: $b \leftarrow x + z$

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

Terms and Definitions

- dependence analysis (cont'd)
  - further example

I1: $a \leftarrow x + y$ I2: $b \leftarrow a + b$

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

  - BERNSTEIN's conditions help to identify instruction-level parallelism or coarser parallelism (e.g. loops)
  - 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

$$a[2] \leftarrow a[0] + 4 \qquad a[3] \leftarrow a[1] + 4$$
$$a[4] \leftarrow a[2] + 4 \qquad a[5] \leftarrow a[3] + 4$$
$$a[6] \leftarrow a[4] + 4 \qquad a[7] \leftarrow a[5] + 4$$

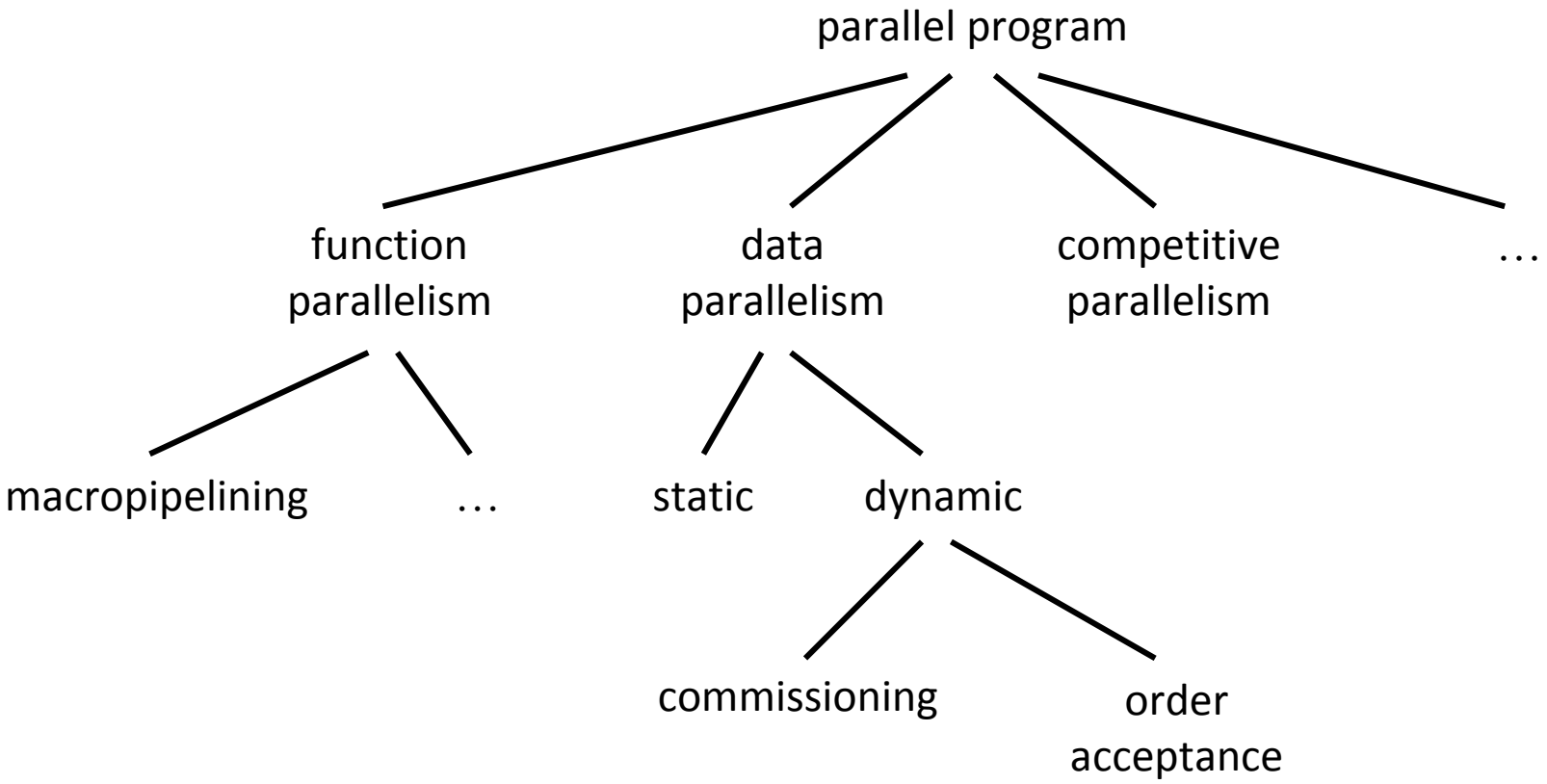
  - hence, $a[3]$ can only be computed after $a[1]$, $a[4]$ after $a[2]$, …
    ➔ computation can be split into two independent loops

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

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

Terms and Definitions

- structures of parallel programs
  - typical parallelisation approaches



Terms and Definitions

- function parallelism
  - parallel execution (on different processors) of components such as functions, procedures, or blocks of instructions (MIMD)
  - 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: often communication between processors necessary
  - 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 organisation / synchronisation overhead)

Terms and Definitions

- data parallelism (cont'd)
  - dynamic structuring
    - commissioning (*master-slave*)
      - one master process assigns data to slave processes
      - both master and slave program necessary
      - master becomes potential bottleneck in case of too much slaves (➔ hierarchical organisation)
    - order polling (*bag-of-tasks*)
      - processes pick next part of available data 'from a bag' as soon as they have finished their computations
      - mostly suitable for UMA / NUMA architectures as bag has to be accessible from all processes (➔ communication overhead for NORMA architectures)

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
  - examples
    - sorting algorithms
    - theorem proving within computational semantics

Terms and Definitions

- parallel programming languages
  - explicit parallelism
    - parallel programming interfaces
      - extension of sequential languages (e.g. C, Fortran) 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, profiler, …
      - most (machine dependent) environments come along with a parallel computer
      - example: MPICH

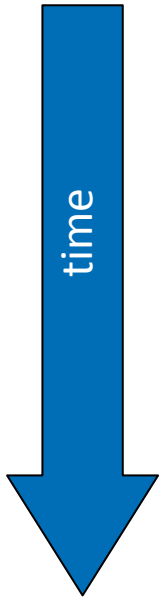
Terms and Definitions

- 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 on UMA / NUMA architectures
  - process interaction on NORMA architectures
  - putting everything together: an example
  - load balancing
  - state-of-art: space-filling curves

Process Interaction on UMA / NUMA Architectures

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



thread 1	thread 2	balance
(withdraw $50)	(withdraw $50)	
read balance: $125		$ 125
	read balance: $125	$ 125
	set balance: $(125–50)	$ 75
set balance: $(125–50)		$ 75
give out cash: $50		$ 75
	give out cash: $50	$ 75

Process Interaction on UMA / NUMA Architectures

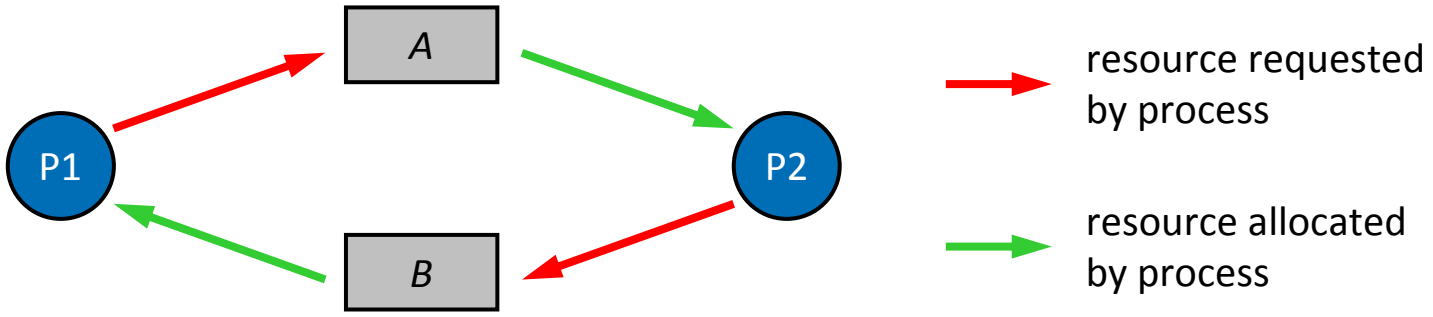
- principles
  - processes depend from each other if they have to be executed in a certain order; this can have two reasons
    - *cooperation*: processes execute parts of a common task
      - producer / consumer: one process generates data to be processed by another one
      - client / server: same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

Process Interaction on UMA / NUMA Architectures

- 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 (e.g. due to write / write or read / write conflicts)
  - 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 delays

Process Interaction on UMA / NUMA Architectures

- 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 on UMA / NUMA Architectures

- synchronisation (cont'd)
  - possibilities to handle deadlocks
    - *deadlock detection*
      - techniques to detect deadlocks (e.g. identification of cycles in waiting graphs) and measures to eliminate them (e.g. rollback)
    - *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 (e.g. HABERMANN's / banker's algorithm well known from OS)

Process Interaction on UMA / NUMA Architectures

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

Process Interaction on UMA / NUMA Architectures

- 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
    - 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 on UMA / NUMA Architectures

- 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 V-operation any suspended process is reactivated (busy waiting); alternatives: always next process in queue
  - *binary semaphore*: has only values '0' and '1' (similar to lock variable, but P and V can be executed by different processes)
  - *general semaphore*: has any nonnegative number

Process Interaction on UMA / NUMA Architectures

- semaphore (cont'd)
  - example: mutual exclusion

(binary) semaphore $s \leftarrow 1$

```
begin procedure proc1( )
   while (true) do
      P(s)
         enter_crit_section( )
      V(s)
   od
end
```

```
begin procedure proc2( )
   while (true) do
      P(s)
         enter_crit_section( )
      V(s)
   od
end
```

procedures *proc1*( ) and *proc2*( ) to be executed in parallel

Process Interaction on UMA / NUMA Architectures

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

(general) semaphore $s \leftarrow 0$

```
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
```

procedures *producer*( ) and *consumer*( ) to be executed in parallel

Process Interaction on UMA / NUMA Architectures

- monitor
  - semaphores solve synchronisation on a very low level ➔ already one wrong semaphore operation might cause 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)

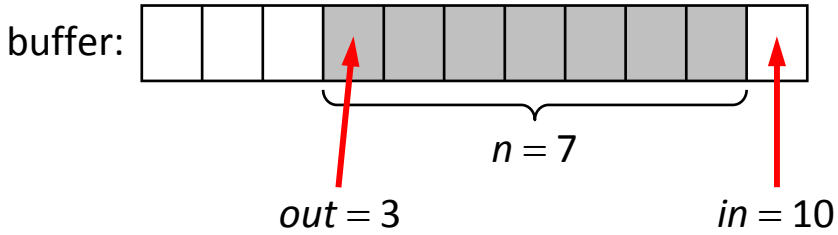
Process Interaction on UMA / NUMA Architectures

- 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 only accessible via operations wait and signal (➔ no manipulation from outside)

Process Interaction on UMA / NUMA Architectures

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

define monitor
  integer: *n*, *in*, *out*, *buffer*[*size*]
  condition: *notempty*, *notfull*
end



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

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

Process Interaction on UMA / NUMA Architectures

- monitor (cont'd)
  - consumer-producer-problem with limited (circular) buffer
  - once *remove*( ) and *store*( ) have been implemented to be used w/o risk

```
begin procedure monitor_init( )
   n ← 0; in ← 0; out ← 0
end
```

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

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

procedures *producer*( ) and *consumer*( ) to be executed in parallel

Process Interaction on UMA / NUMA Architectures

- 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 number 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 on UMA / NUMA Architectures

- 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 on UMA / NUMA Architectures

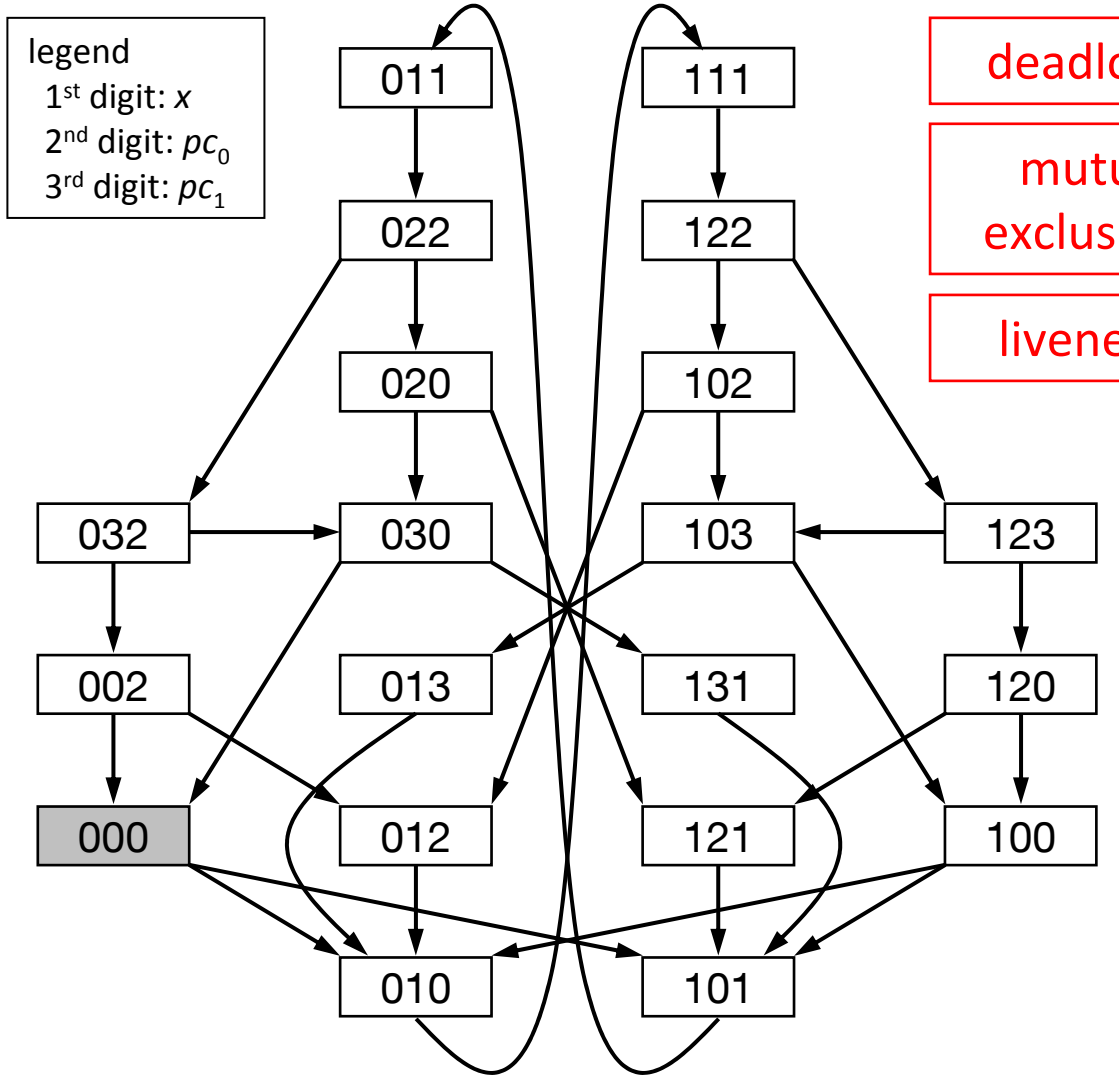
- simple case study (cont'd)
  - test case: reader-writer-problem (according to S. Siegel, UD, USA)
  - to be examined
    - *deadlock*: program will never deadlock
    - *mutual exclusion*: resource is never used by both processes at same time
    - *liveness*: resource will eventually be used by any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

```
                    boolean x ← 0

                    begin procedure rw0( )
                      while (true) do
pc0 { 0:              x ← 0
    { 1:              sync( )
    { 2:              if x = 0 then
    { 3:                 use_resource
                      fi
                    od
                    end

                    begin procedure rw1( )
                      while (true) do
pc1 { 0:              x ← 1
    { 1:              sync( )
    { 2:              if x = 1 then
    { 3:                 use_resource
                      fi
                    od
                    end
```

Process Interaction on UMA / NUMA Architectures

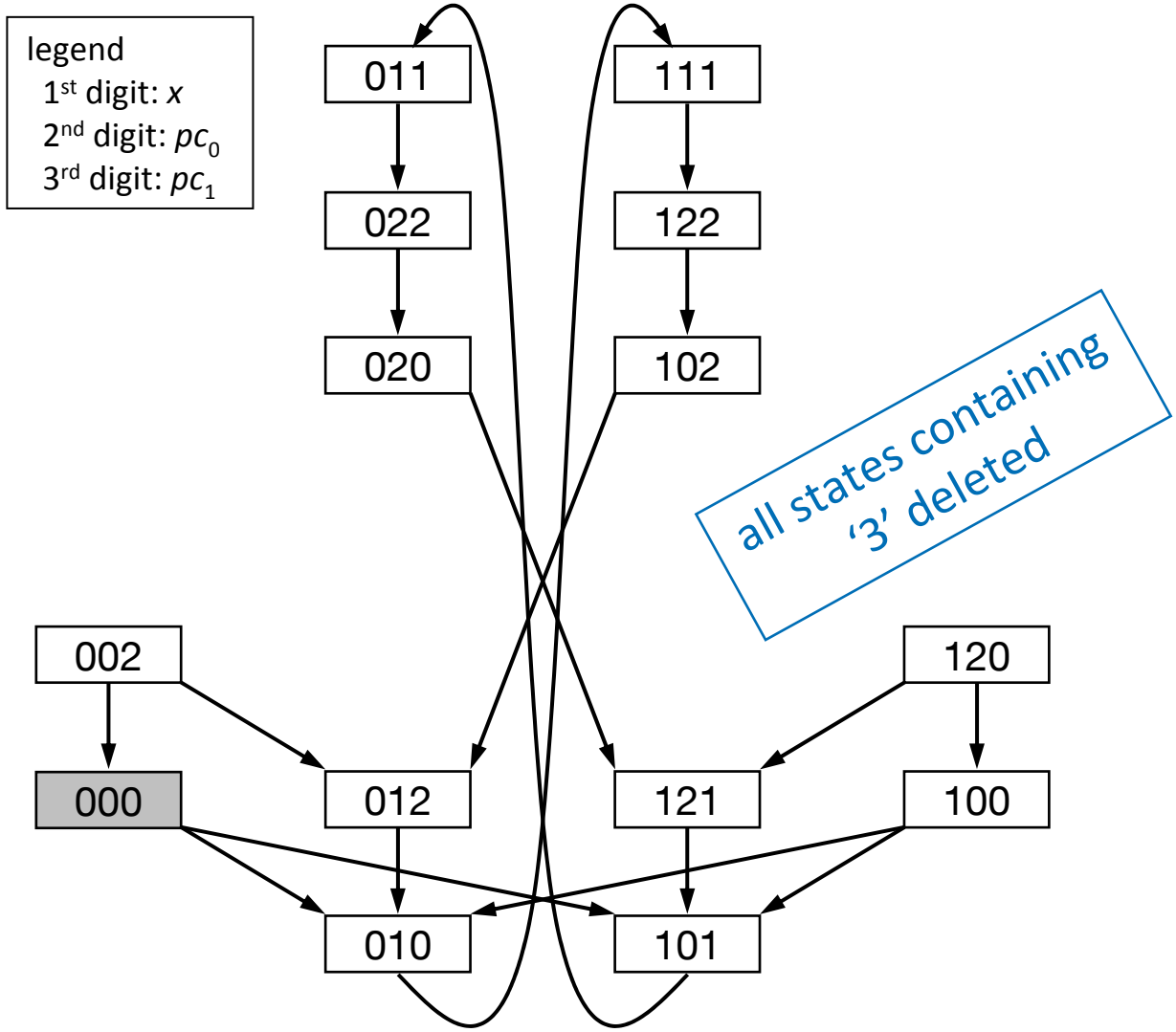



```
    boolean x ← 0

    begin procedure rw0( )
      while (true) do
0:      x ← 0
1:      sync( )
2:      if x = 0 then
3:        use_resource
        fi
      od
    end

    begin procedure rw1( )
      while (true) do
0:      x ← 1
1:      sync( )
2:      if x = 1 then
3:        use_resource
        fi
      od
    end
```

Process Interaction on UMA / NUMA Architectures

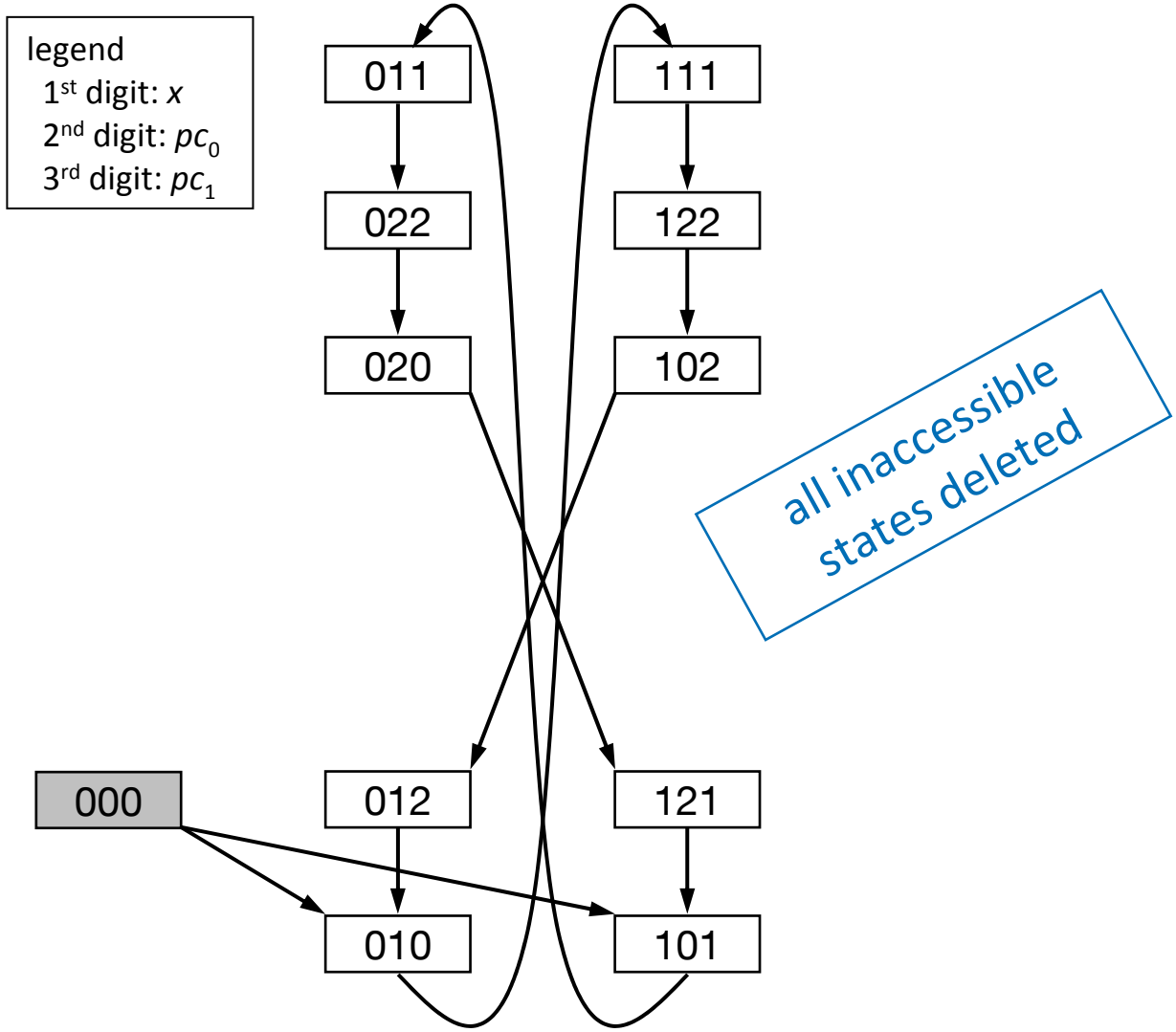



```
        boolean x ← 0

        begin procedure rw0( )
          while (true) do
0:          x ← 0
1:          sync( )
2:          if x = 0 then
3:            use_resource
            fi
          od
        end

        begin procedure rw1( )
          while (true) do
0:          x ← 1
1:          sync( )
2:          if x = 1 then
3:            use_resource
            fi
          od
        end
```

Process Interaction on UMA / NUMA Architectures

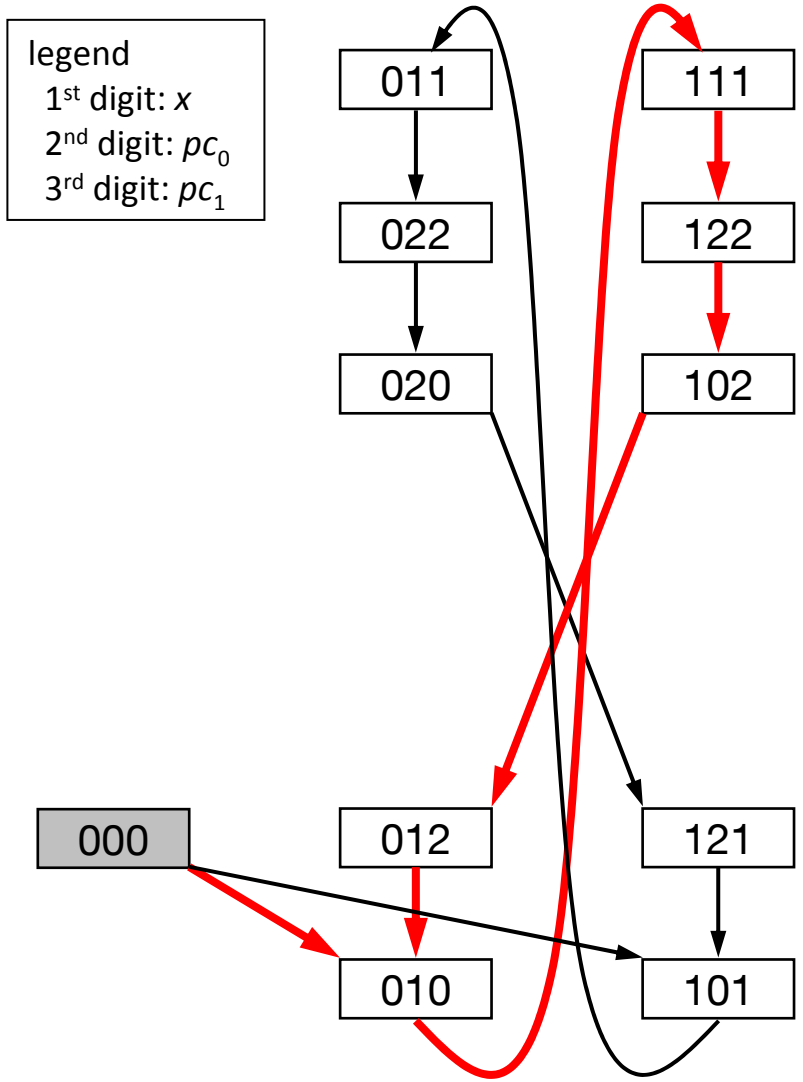



```
        boolean x ← 0

        begin procedure rw0( )
          while (true) do
0:          x ← 0
1:          sync( )
2:          if x = 0 then
3:            use_resource
            fi
          od
        end

        begin procedure rw1( )
          while (true) do
0:          x ← 1
1:          sync( )
2:          if x = 1 then
3:            use_resource
            fi
          od
        end
```

Process Interaction on UMA / NUMA Architectures

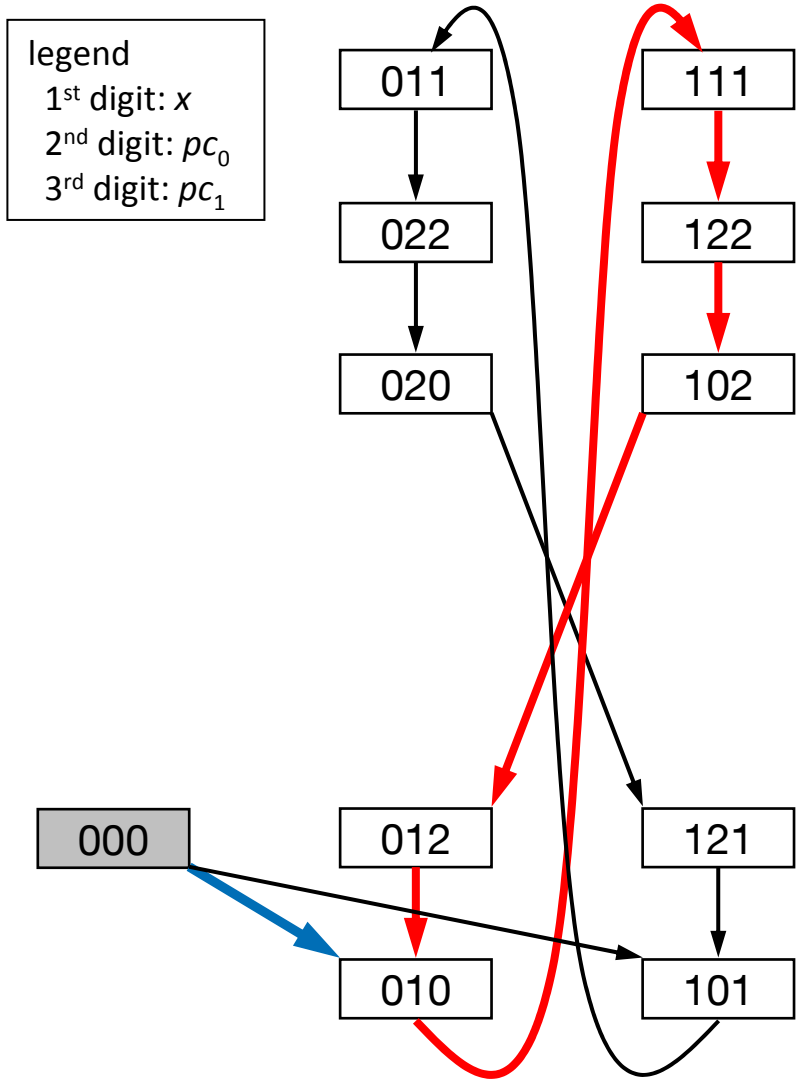



```
     boolean x ← 0

     begin procedure rw0( )
        while (true) do
0:         x ← 0
1:         sync( )
2:         if x = 0 then
3:            use_resource
           fi
        od
     end

     begin procedure rw1( )
        while (true) do
0:         x ← 1
1:         sync( )
2:         if x = 1 then
3:            use_resource
           fi
        od
     end
```

Process Interaction on UMA / NUMA Architectures

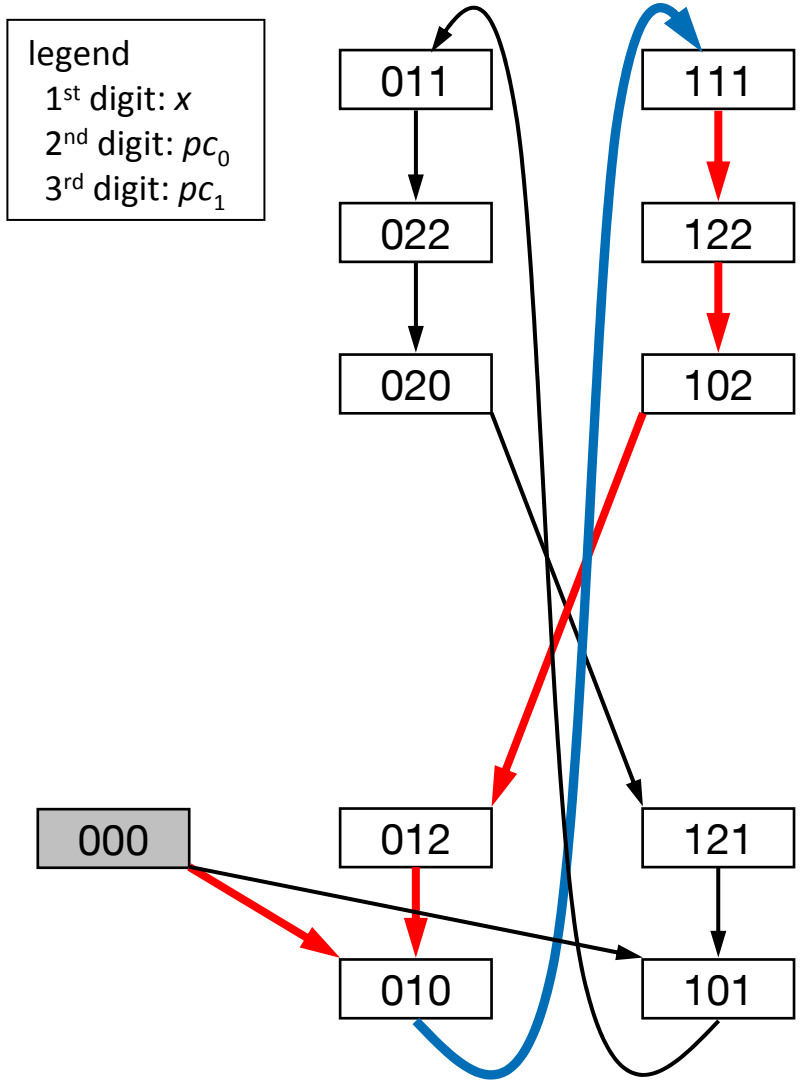



```
        boolean x ← 0

        begin procedure rw0( )
          while (true) do
0:          x ← 0
1:          sync( )
2:          if x = 0 then
3:            use_resource
            fi
          od
        end

        begin procedure rw1( )
          while (true) do
0:          x ← 1
1:          sync( )
2:          if x = 1 then
3:            use_resource
            fi
          od
        end
```

Process Interaction on UMA / NUMA Architectures

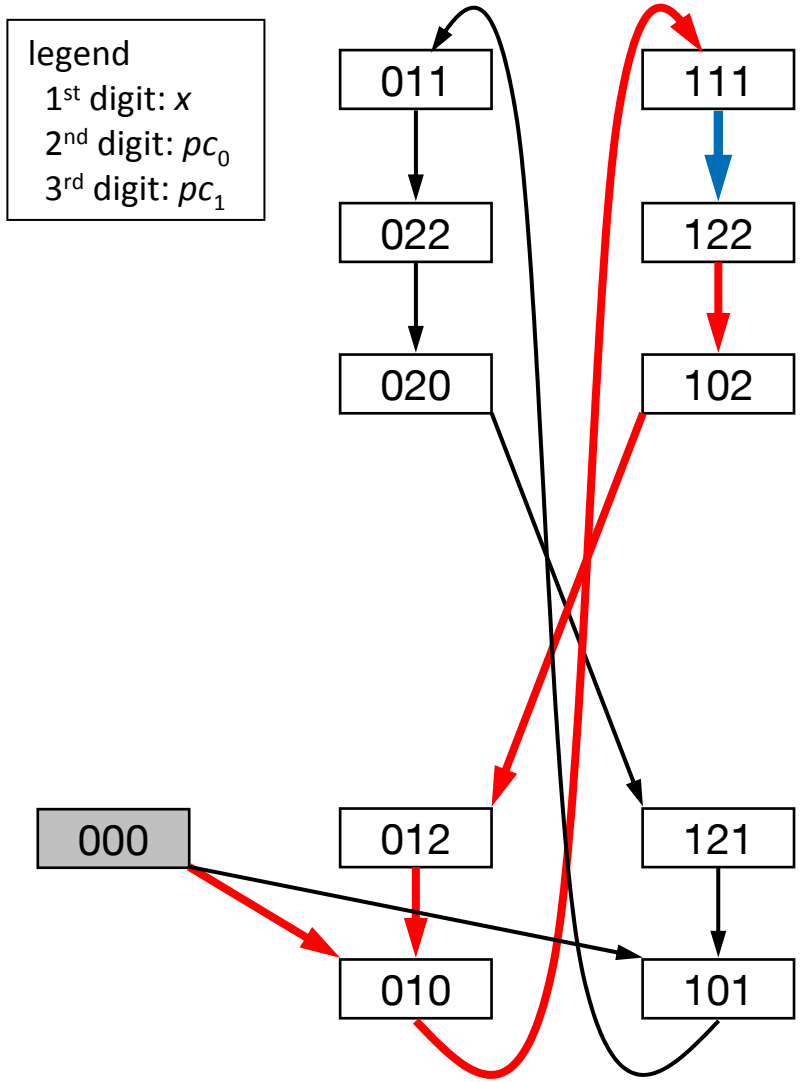



```
boolean x ← 0

begin procedure rw0( )
   while (true) do
0:    x ← 0
1:    sync( )
2:    if x = 0 then
3:       use_resource
      fi
   od
end

begin procedure rw1( )
   while (true) do
0:    x ← 1
1:    sync( )
2:    if x = 1 then
3:       use_resource
      fi
   od
end
```

Process Interaction on UMA / NUMA Architectures

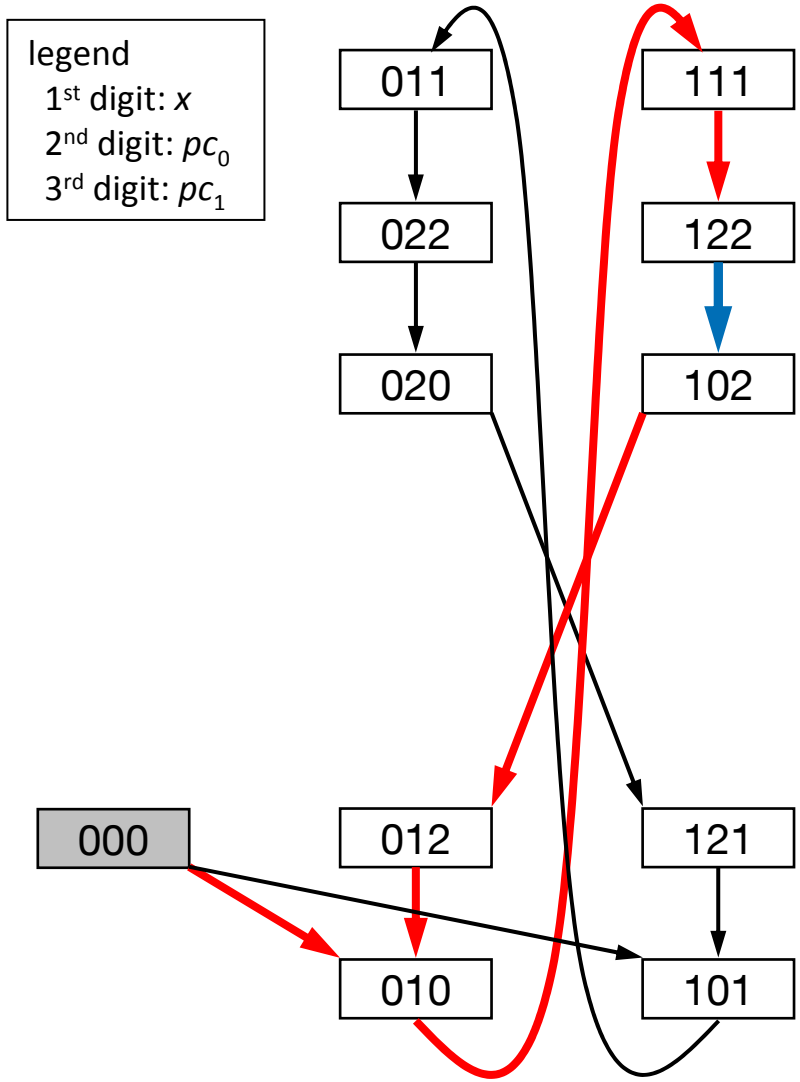



```
boolean x ← 0

begin procedure rw0( )
   while (true) do
0:    x ← 0
1:    sync( )
2:    if x = 0 then
3:       use_resource
      fi
   od
end

begin procedure rw1( )
   while (true) do
0:    x ← 1
1:    sync( )
2:    if x = 1 then
3:       use_resource
      fi
   od
end
```

Process Interaction on UMA / NUMA Architectures

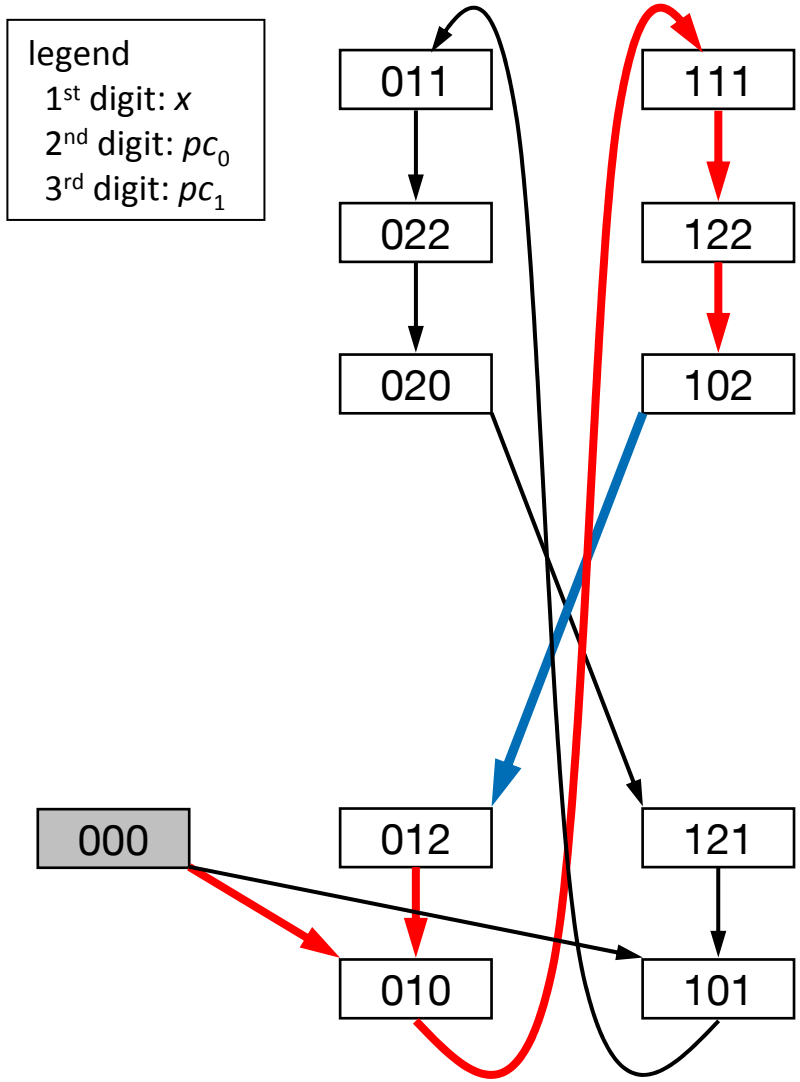



```
boolean x ← 0

begin procedure rw0( )
   while (true) do
0:    x ← 0
1:    sync( )
2:    if x = 0 then
3:       use_resource
      fi
   od
end

begin procedure rw1( )
   while (true) do
0:    x ← 1
1:    sync( )
2:    if x = 1 then
3:       use_resource
      fi
   od
end
```

Process Interaction on UMA / NUMA Architectures

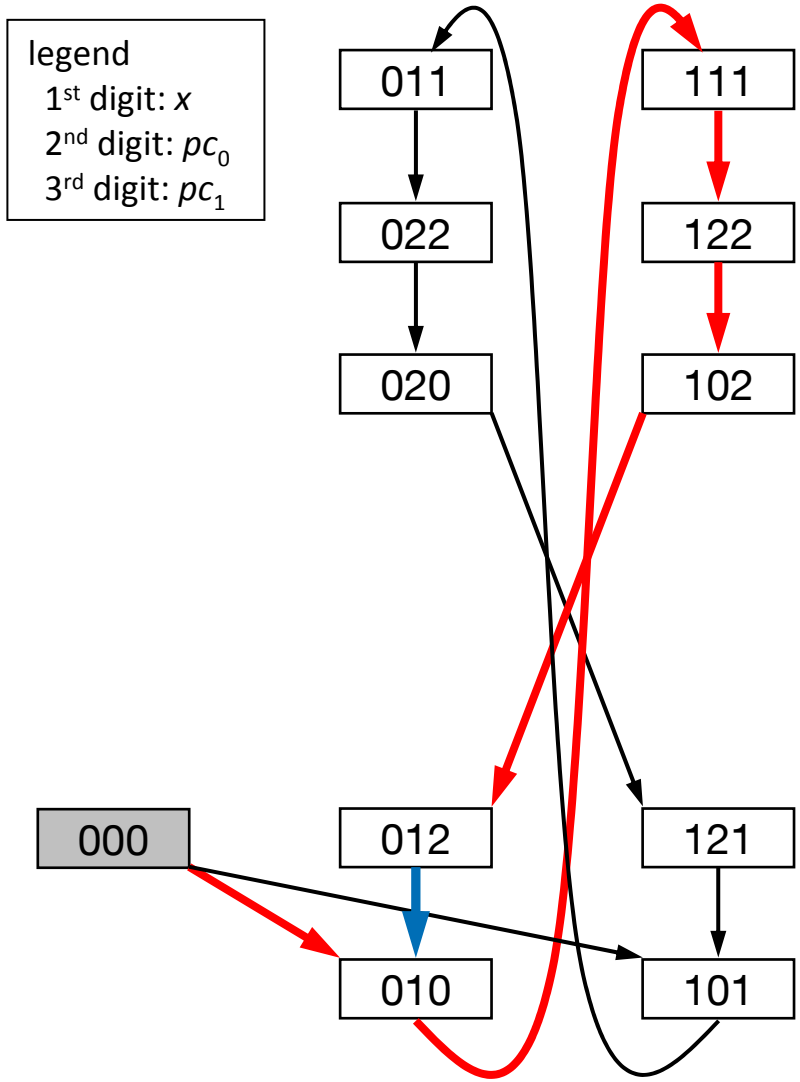



```
boolean x ← 0

begin procedure rw0( )
   while (true) do
0:       x ← 0
1:       sync( )
2:       if x = 0 then
3:          use_resource
         fi
   od
end

begin procedure rw1( )
   while (true) do
0:       x ← 1
1:       sync( )
2:       if x = 1 then
3:          use_resource
         fi
   od
end
```

Process Interaction on UMA / NUMA Architectures

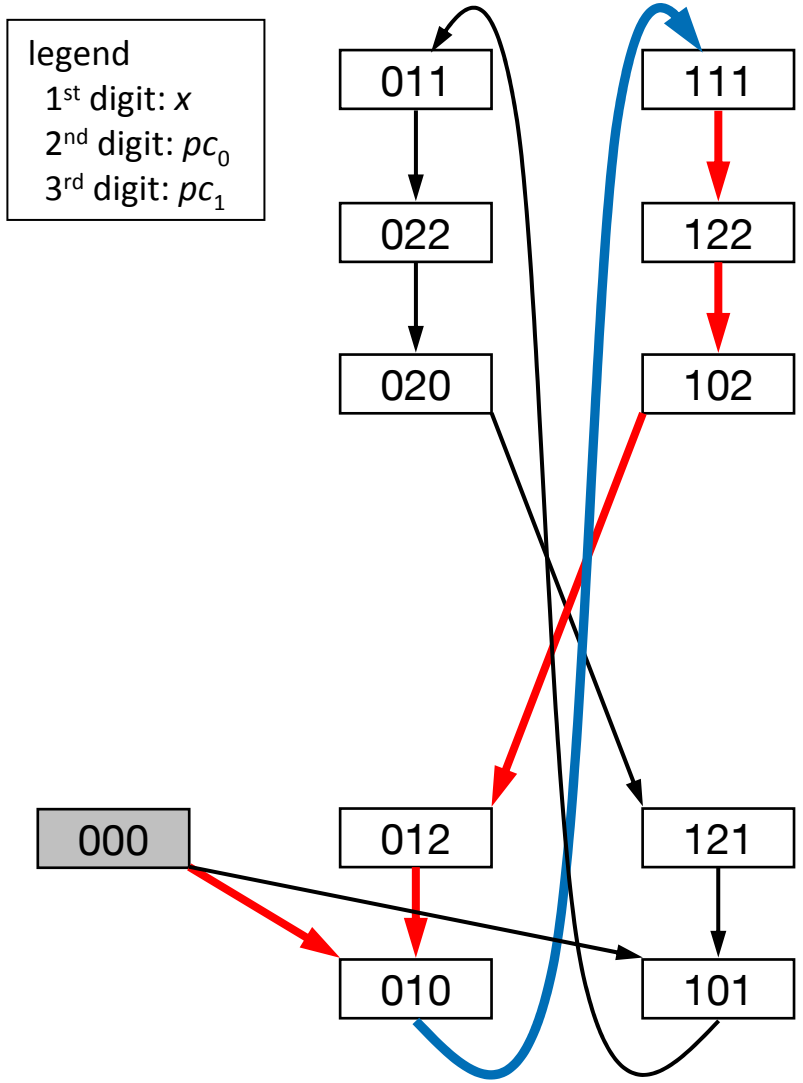



```
        boolean x ← 0

        begin procedure rw0( )
          while (true) do
0:          x ← 0
1:          sync( )
2:          if x = 0 then
3:            use_resource
            fi
          od
        end

        begin procedure rw1( )
          while (true) do
0:          x ← 1
1:          sync( )
2:          if x = 1 then
3:            use_resource
            fi
          od
        end
```

Process Interaction on UMA / NUMA Architectures




```
    boolean x ← 0

    begin procedure rw0( )
      while (true) do
0:      x ← 0
1:      sync( )
2:      if x = 0 then
3:        use_resource
        fi
      od
    end

    begin procedure rw1( )
      while (true) do
0:      x ← 1
1:      sync( )
2:      if x = 1 then
3:        use_resource
        fi
      od
    end
```

- overview
  - terms and definitions ✓
  - process interaction on UMA / NUMA architectures ✓
  - process interaction on NORMA architectures
  - putting everything together: an example
  - load balancing
  - state-of-art: space-filling curves

Process Interaction on NORMA Architectures

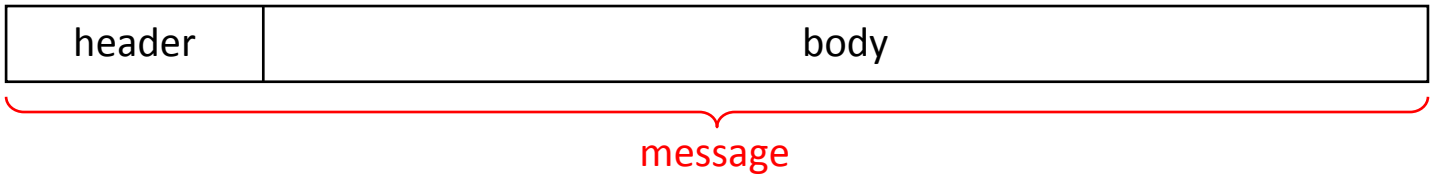
- message passing paradigm
  - no shared memory for synchronisation and communication
  - hence, transfer mechanism for information interchange necessary
  - message passing
    - messages: data units transferred between processes
    - send / receive operations instead of read / write operations
  - 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
    - messages are resources that don't exist before the send and in general also after the receive operation

Process Interaction on NORMA Architectures

- messages
  - created whenever a process performs a send
  - necessary information to be provided from the sender
    - destination (e.g. process, node, communication channel)
    - unique message identifier (e.g. number)
    - data type and number of elements to be transferred
    - memory (address) containing the data to be transferred

header

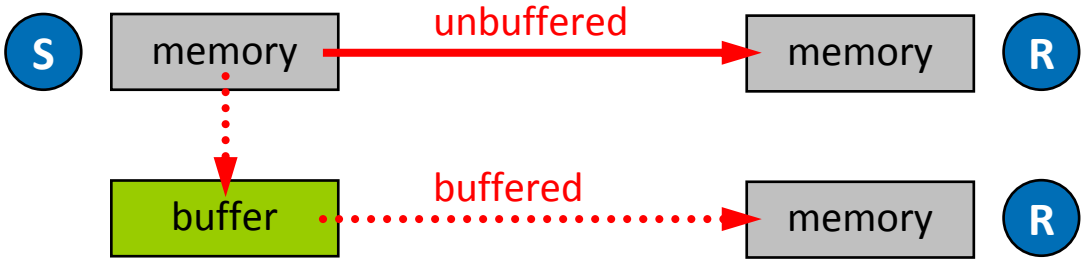
body



  - data type and number of elements must match for the receiver, otherwise a correct interpretation of data cannot be guaranteed

Process Interaction on NORMA Architectures

- sending messages
  - send operations can be
    - synchronous / asynchronous: sender is dependent on the availability of the receiver (synchronous) or not (asynchronous)
    - buffered / unbuffered: sender may first copy the data into so-called send buffer for later transfer (buffered) or directly perform the transfer from memory to memory (unbuffered)



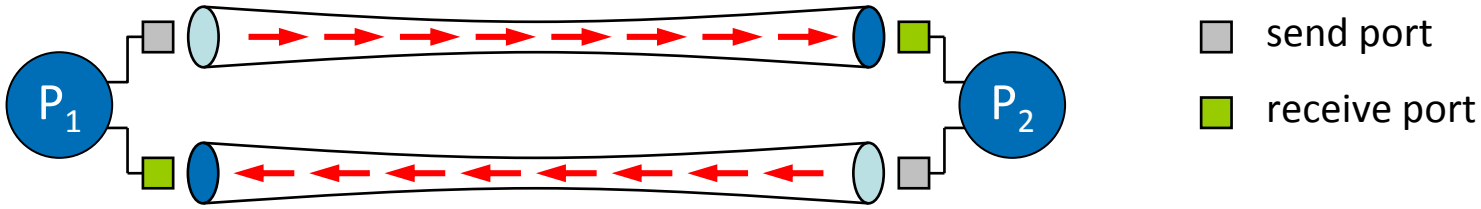
    - blocking / non-blocking: sender gets blocked until send operation finishes (blocking) or is given immediate control to continue with course of program (non-blocking)

Process Interaction on NORMA Architectures

- receiving messages
  - a process has to specify which message to receive (via message identifier or wildcard) and where to store the data (memory address)
  - receive operations can be
    - destructive / non-destructive: message is destroyed after receive (destructive) or kept for later usage (non-destructive)
    - synchronous / asynchronous: receiver is dependent on the availability of the sender (synchronous) or not (asynchronous)

Process Interaction on NORMA Architectures

- 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 (e.g. Distributed Execution and Communication Kernel (DECK))
    - port: a port is bound to one process and can be used in one direction only, i.e. either for sending or receiving messages (➔ sockets)
    - connection / channel: required for communication via ports, i.e. send / receive ports are connected (via virtual channels) for data transfer



- overview
  - terms and definitions ✓
  - process interaction on UMA / NUMA architectures ✓
  - process interaction on NORMA architectures ✓
  - putting everything together: an example
  - load balancing
  - state-of-art: space-filling curves

Putting Everything Together: An Example

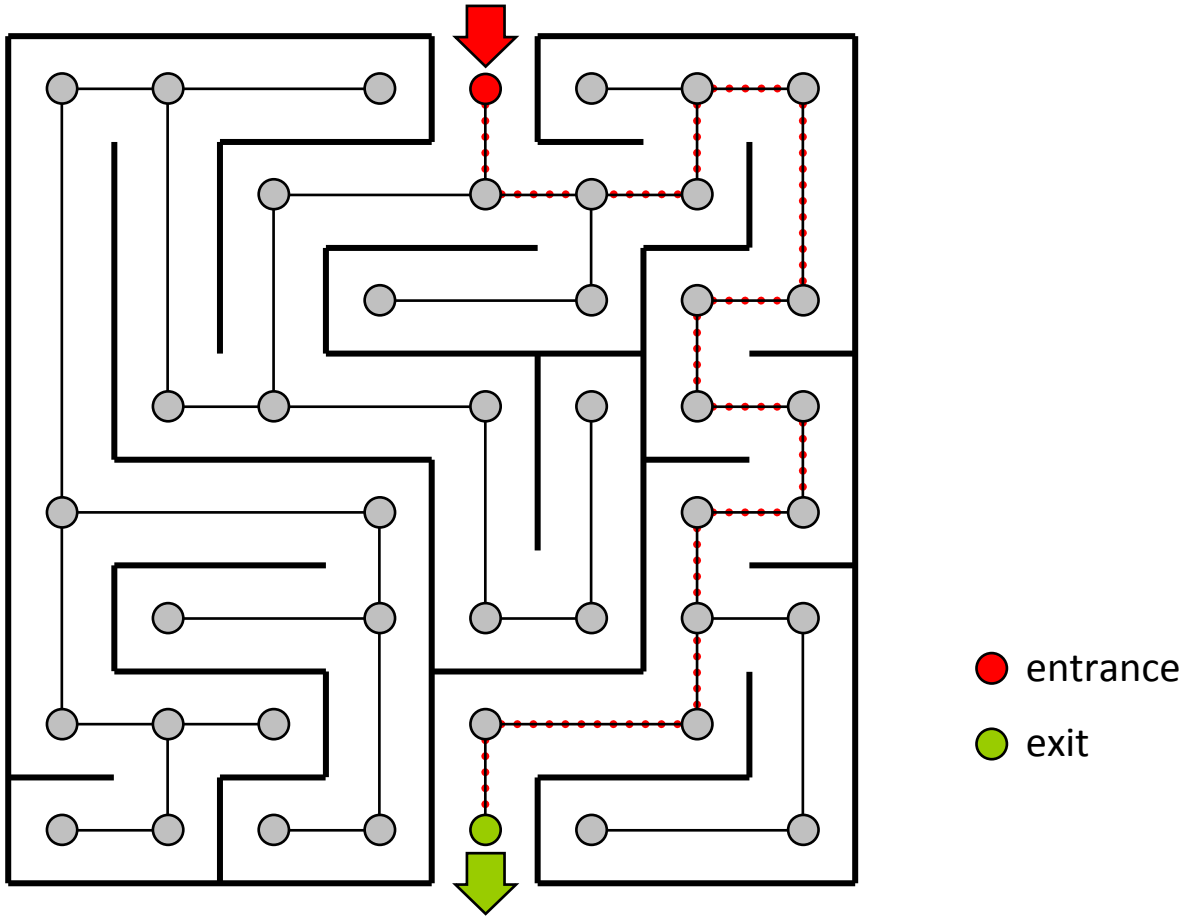
- problem setup
  - given: map of some labyrinth that contains
    - one entrance
    - one exit
    - no cycles

source: viralmonster.net

  - task: determine if there exists a way from the entrance through the labyrinth to the exit (not the way itself) ➔ binary answer: yes | no

Putting Everything Together: An Example

- problem definition
  - labyrinth stored as graph G = (V, E)



Putting Everything Together: An Example

- problem solution
  - sequential algorithm

```
position ← entrance

while (true) do
   position ← walk ( )

   switch (position) do
      case 'crossing':     position ← turn_right ( )
      case 'dead end':     position ← turn_around ( )
      case 'exit':         halt ('exit found')
      case 'entrance':     halt ('error')
   od

od
```

How does this work in parallel?

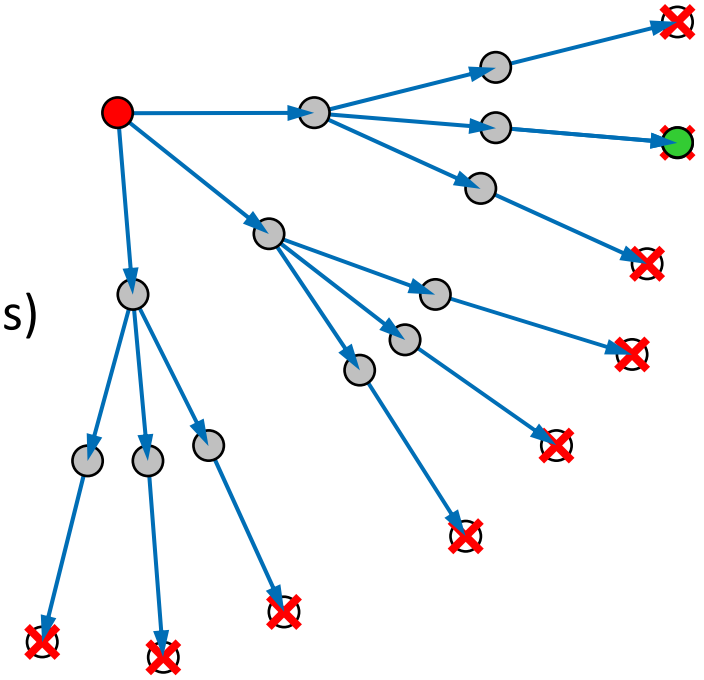
source: moviepilot.de

Putting Everything Together: An Example

- competitive parallelism
  - start *N* processes following *N* different algorithms
  - first process reaching exit or entrance tells other processes 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

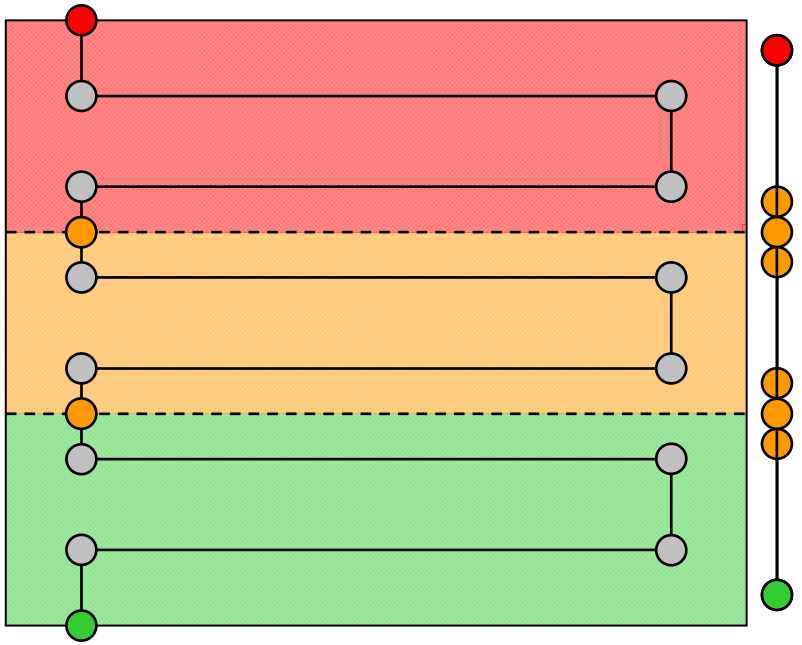
Putting Everything Together: An Example

- function parallelism
  - assumption: pool of processes [0, *N*–1]
  - start new processes at crossroads
  - terminate processes at dead ends
  - halt in case
    - one process reached the exit (➔ success)
    - all processes terminated (➔ error)
  - questions
    - shared or distributed memory
    - drawbacks



Putting Everything Together: An Example

- data parallelism
  - cut graph into *N* parts and distribute among processes
  - solve corresponding subproblems for all entrance-exit pairs
  - collect results, assemble ‘smaller’ problem, and repeat previous steps
  - simplified case
  - questions
    - shared or distributed memory
    - drawbacks
    - and what about MINSKY ☺



- overview
  - terms and definitions ✓
  - process interaction on UMA / NUMA architectures ✓
  - process interaction on NORMA architectures ✓
  - putting everything together: an example ✓
  - load balancing
  - state-of-art: space-filling curves

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 (e.g. heterogeneous systems)
    - different latencies for communication (e.g. grid / cloud computing)
  - objective: simple, but efficient load balancing strategies

Load Balancing

- static load balancing
  - to be applied before execution of any process (in contrast to dynamic load balancing to be applied during execution)
  - usually referred to as mapping problem or scheduling problem
  - potential techniques
    - round robin: assigning tasks in sequential order to processes, coming back to the first when all processes have been served
    - randomised: selecting processes at random to assign tasks
    - recursive bisection: recursive division into smaller tasks of equal computational effort with less communication costs
    - genetic algorithm: finding an optimal distribution of tasks according to a given objective function

Load Balancing

- dynamic load balancing
  - division of tasks dependent upon execution of the program ➔ entails additional overhead (to be kept small, otherwise 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 is responsible for assignment of tasks to slaves
    - decentralised
      - tasks are passed between arbitrary processes
      - worker processes operate upon the problem and interact among themselves ➔ a worker process may receive tasks from other or may send tasks to others

Load Balancing

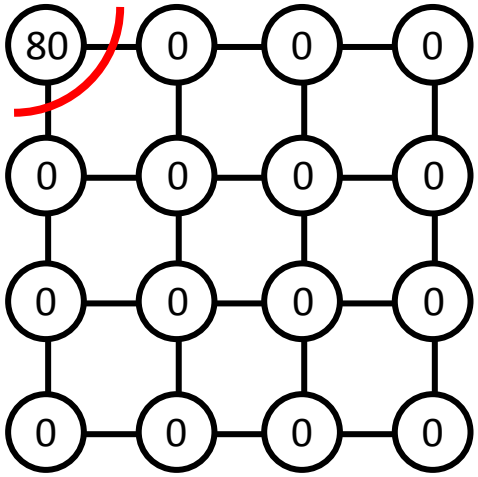
- diffusion model (a.k.a first order scheme)
  - analogy to physical processes in nature (e.g. salt or ink in water)
  - original algorithm introduced by CYBENKO (1989) for static network topologies, meanwhile it has been often studied and derived (e.g. second order scheme, dynamic network topologies)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

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

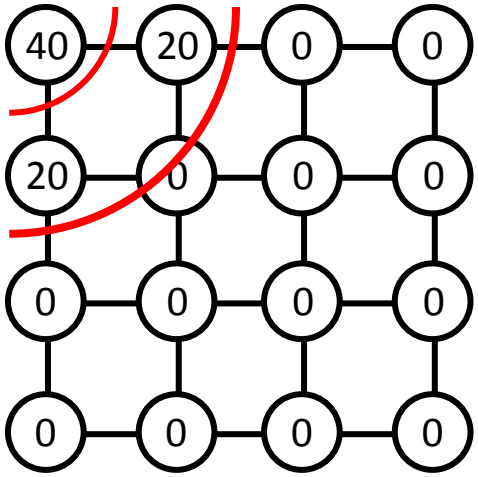
    where $w_i^{(t)}$ defines the workload done by process P_i at time t
  - various methods to be found that determine parameter α_{ij} such as
    - optimal choice: needs global knowledge of the network
    - BOILLAT choice: needs only local knowledge of the neighbours

Load Balancing

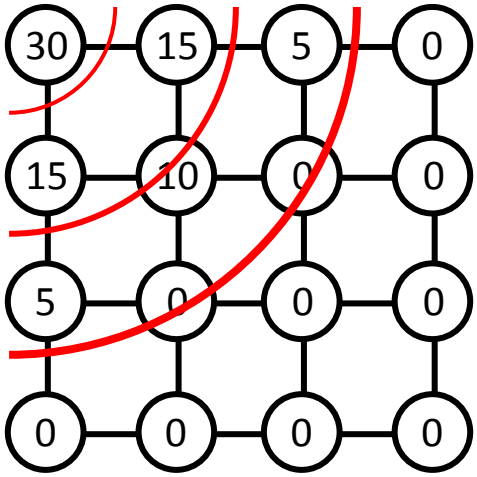
- diffusion model (cont'd)
  - update of workload can be done
    a) after all balancing factors have been computed (JACOBI-like)
    b) during computation of balancing factors (GAUSS-SEIDEL-like)
  - example: first two iteration steps according to method a) for a 2D grid with a ratio of $\alpha = 0.25$ for workload swapping



initial setup ($t = 0$)



initial setup ($t = 1$)



initial setup ($t = 2$)

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

- balanced allocation (balls into bins)
  - basic idea: placing *N* balls into *N* bins at random choice (extensively studied problem from probability and statistics)
  - variant of the above
    - each ball b_i comes with $D(b_i)$ possible destinations (to be placed), chosen independently and uniformly at random
    - ball b_i is placed in the least full bin among $D(b_i)$ possible destinations
  - applied to load balancing: a process p_i selects $D(p_i)$ processes at random and passes some of its workload to the least loaded one
  - for temporary tasks (i.e. tasks that are finished at unpredictable times) this strategy has a competitive ratio of $\mathrm{O}(\sqrt{N})$ compared to the optimal off-line strategy (that has global knowledge)

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 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 on UMA / NUMA architectures ✓
  - process interaction on NORMA architectures ✓
  - putting everything together: an example ✓
  - load balancing ✓
  - state-of-art: space-filling curves

State-of-art: Space-Filling Curves

- definition
  - 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-art: Space-Filling Curves

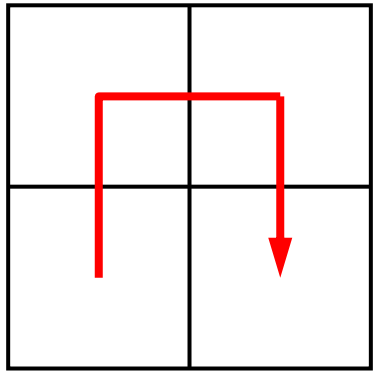
- HILBERT's space filling curve
  - for reasons of simplicity only in 2D ➔ $f : I = [0,1] \rightarrow [0,1]^2 = Q$
  - construction of SFC follows the geometric conception

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-art: Space-Filling Curves

- HILBERT's space filling curve (cont'd)
  - generation process
    1) starting with a generator or 'Leitmotiv' that defines the order in which the subsquares are visited
    2) recursively applying generator in each subsquare (with appropriate similarity transformations if necessary)
    3) connecting the open ends

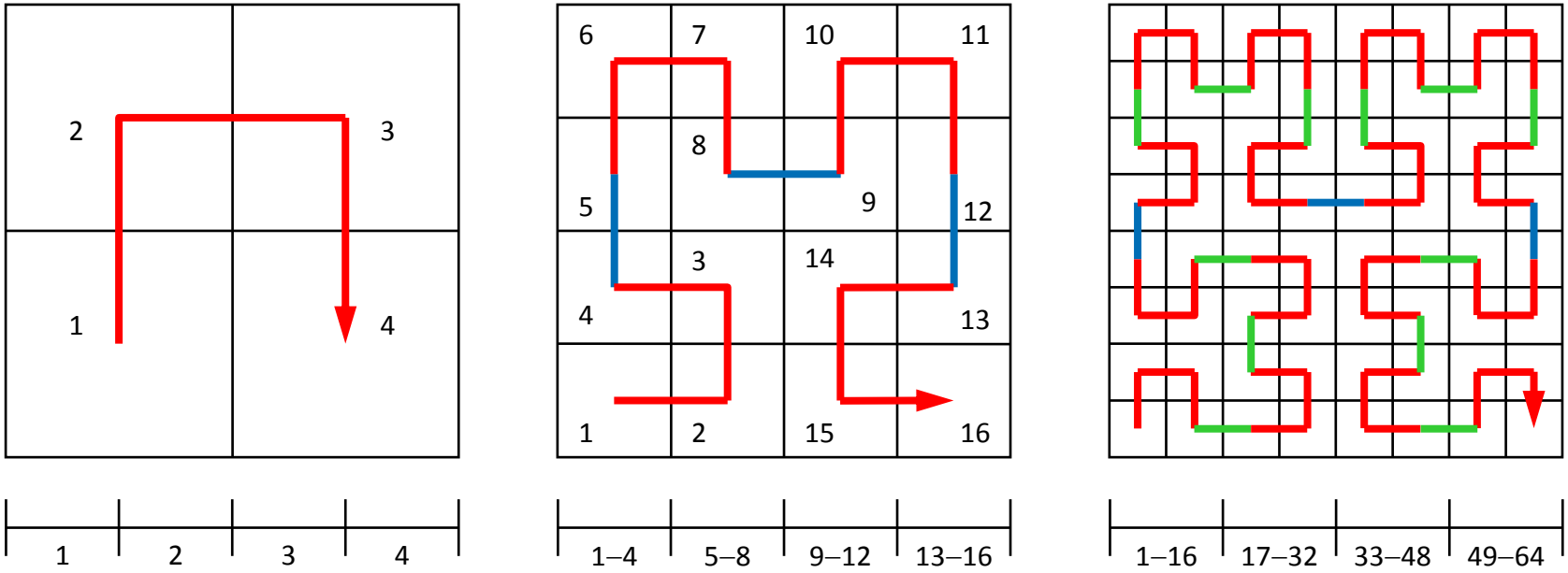


generator for HILBERT's SFC

  - of course, the iterative steps in this generation process are of practical relevance, not the border case itself

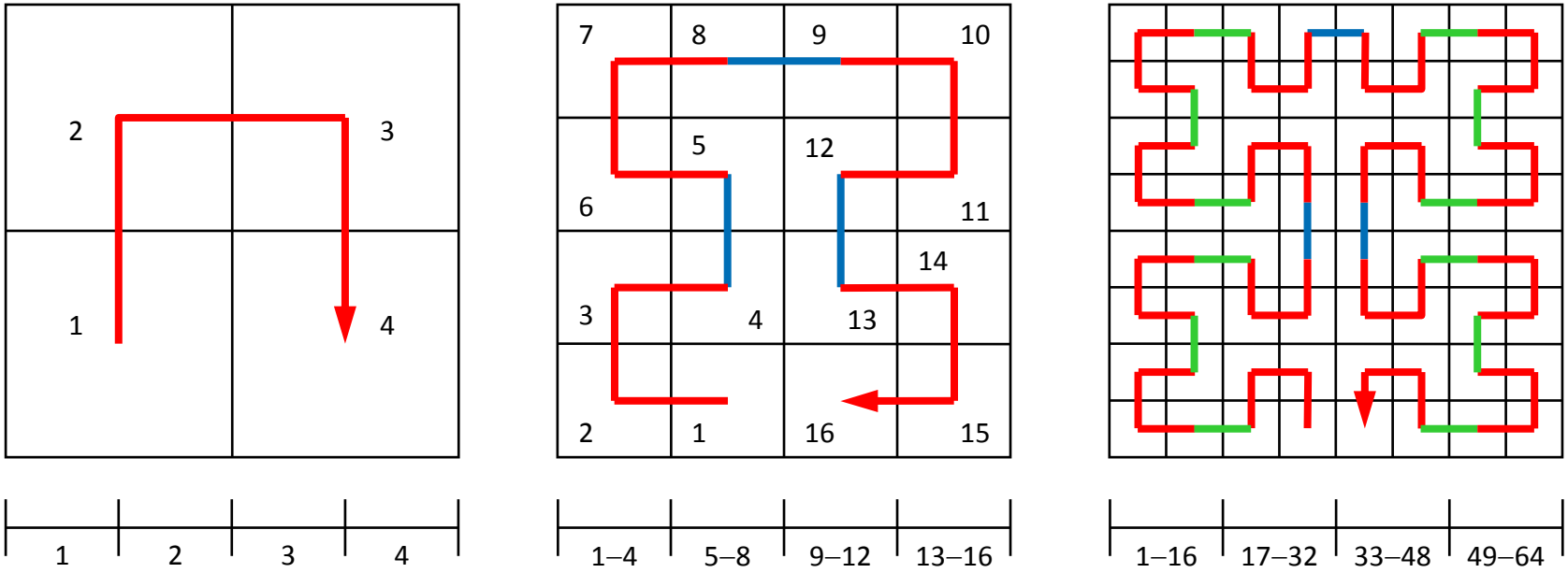
State-of-art: Space-Filling Curves

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



State-of-art: Space-Filling Curves

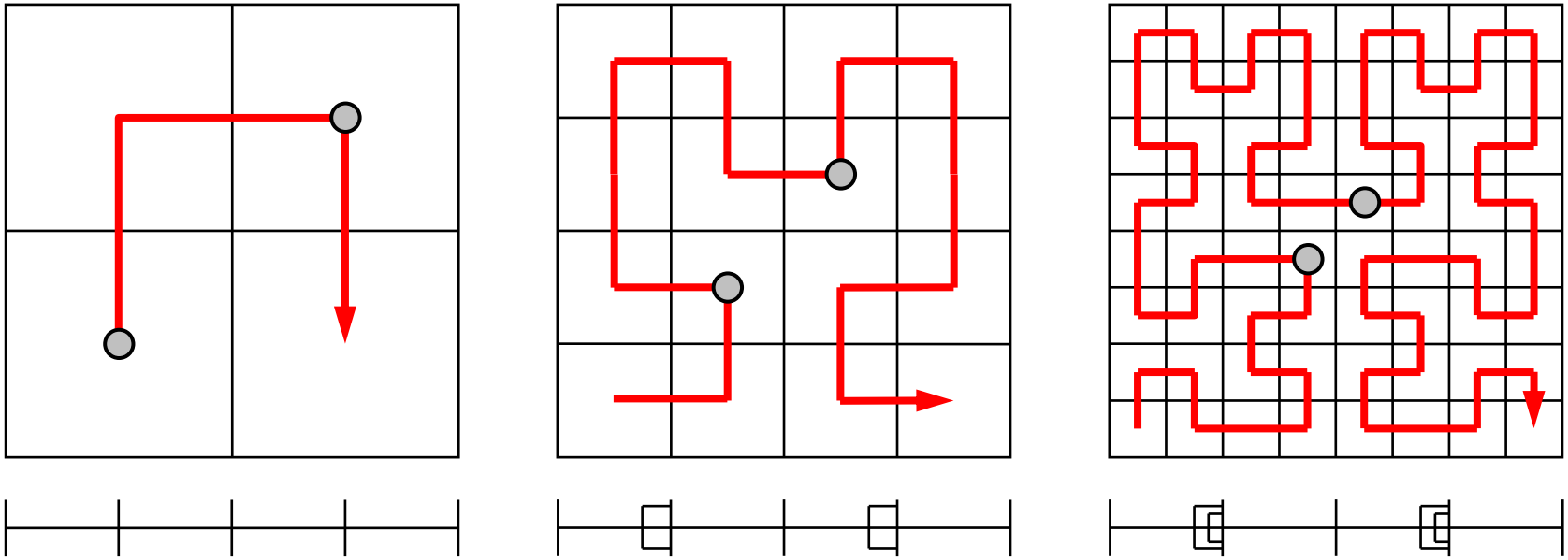
- HILBERT's space filling curve (cont'd)
  - variant of MOORE



  - modulo symmetry, these are the only two possibilities

State-of-art: Space-Filling Curves

- 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 source point)

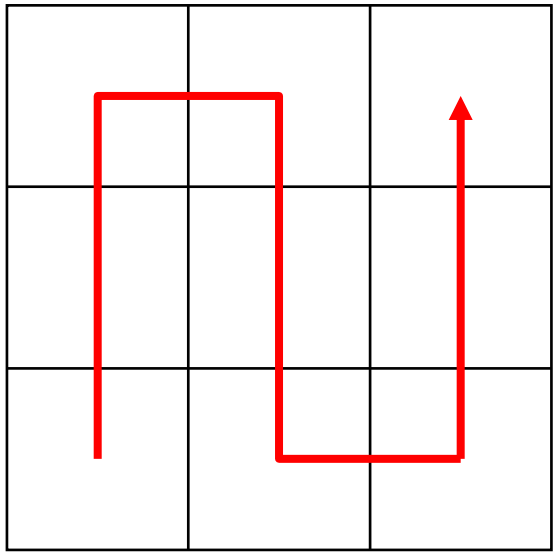


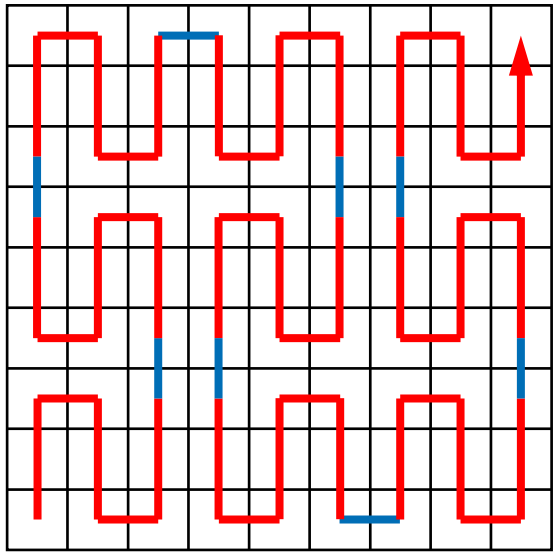
  - important precondition: there exists a bijective mapping between two finite-dimensional smooth manifolds (CANTOR, 1878), but it cannot be both bijective and continuous (NETTO, 1879)

State-of-art: Space-Filling Curves

- 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

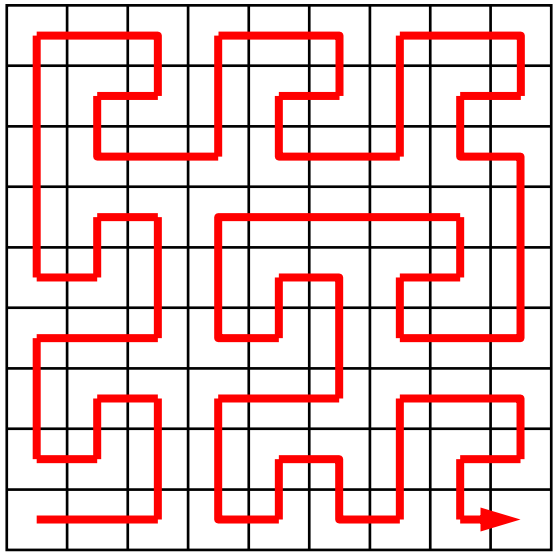
3	4	9
2	5	8
1	6	7



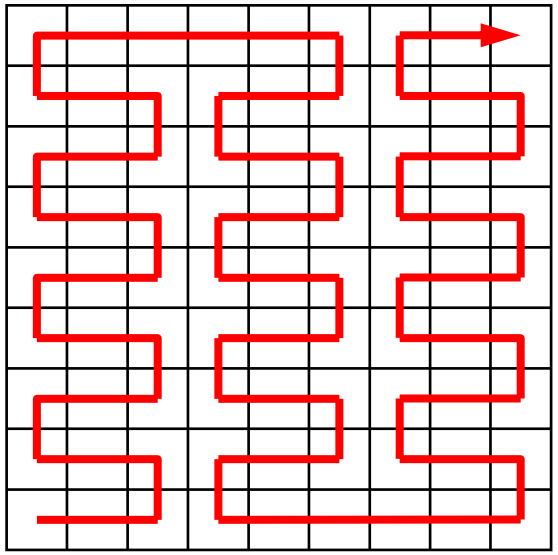


State-of-art: Space-Filling Curves

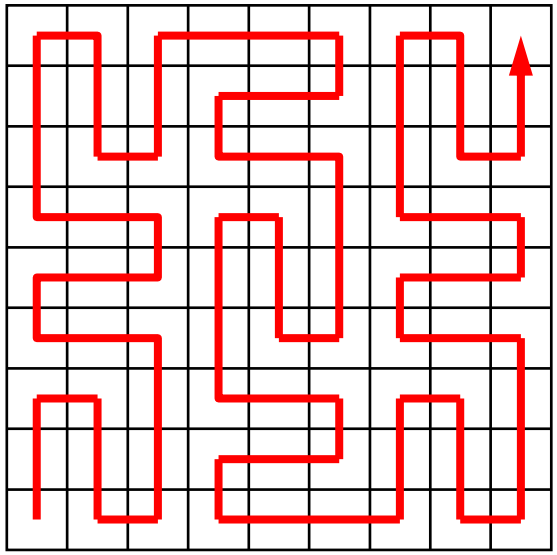
- PEANO's space filling curve (cont'd)
  - there are (modulo symmetry) 273 different possibilities to recursively apply the generator preserving neighbourhood and inclusion



meander type



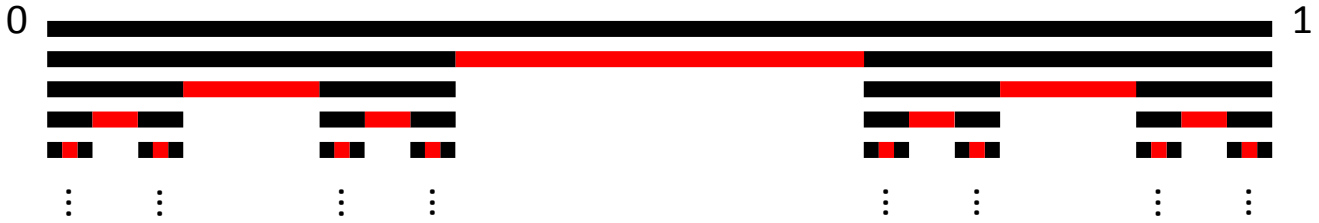
serpentine type



serpentine type

State-of-art: Space-Filling Curves

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



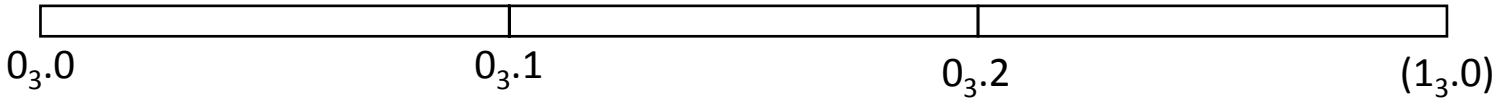
  - *C* is defined as set of points not excluded, hence the remaining interval can be computed by the total length removed

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

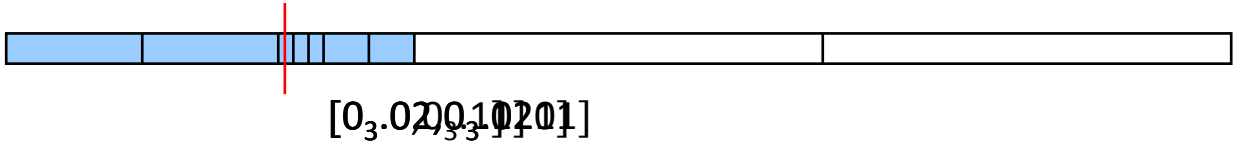
  - the proportion of the remaining interval seems to be $1 - 1 = 0$, but in fact *C* has the same cardinality as the unit interval [0,1] (!)

State-of-art: Space-Filling Curves

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



  - example: parameter $T = 2/9$



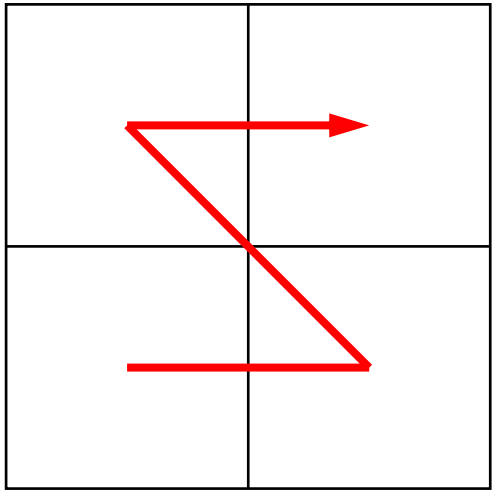
  - since the middle third (indicated by '1') is repeatedly deleted, the CANTOR set only contains ternary numbers that consist of '0' and '2'

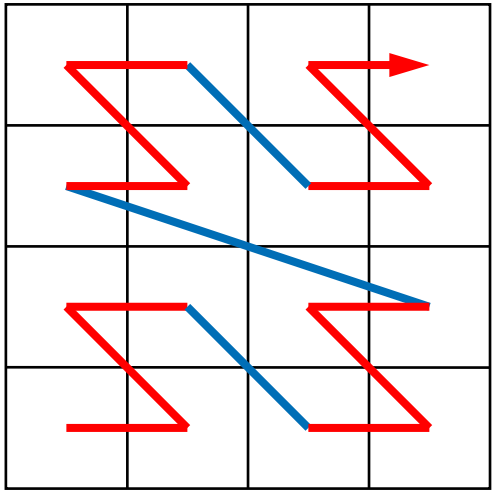
State-of-art: Space-Filling Curves

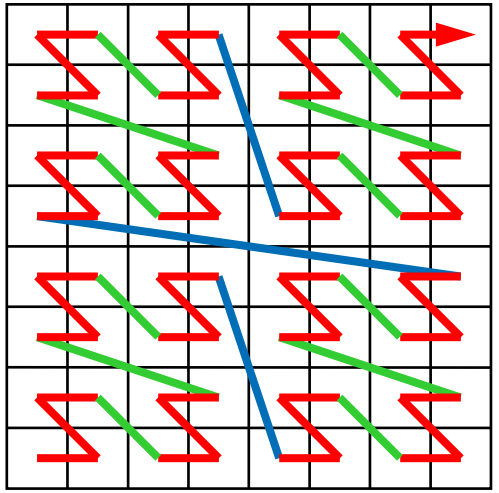
- LEBESGUE's space filling curve (cont'd)
  - when mapping C to $[0,1]^2$ according to

$$f: \begin{pmatrix} \frac{0_3.w_1w_2w_3w_4...}{2} \end{pmatrix} \rightarrow \begin{pmatrix} 0_2.x_2x_4... \\ 0_2.y_1y_3... \end{pmatrix}$$

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







State-of-art: Space-Filling Curves

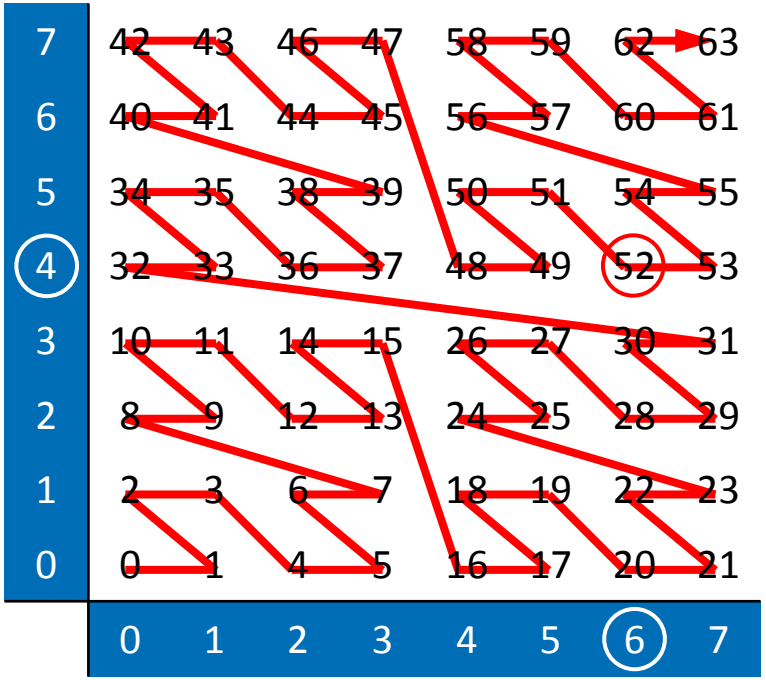
- LEBESGUE's space filling curve (cont'd)
  - Z-ordering is well-known from quadtrees / octrees when linearising a tree by a depth-first traversal (➔ lexicographic or MORTON index)
  - for load distribution inverse function $f^{-1} : [0,1]^D \rightarrow [0,1]$ necessary
  - bitwise interleaving of coordinate values (x, y) leads to Z-value

$x = 6 \rightarrow 110_2$

$y = 4 \rightarrow 100_2$

$110100_2 \rightarrow 52 = Z$

➔ simple conversion $(x,y) \leftrightarrow Z$

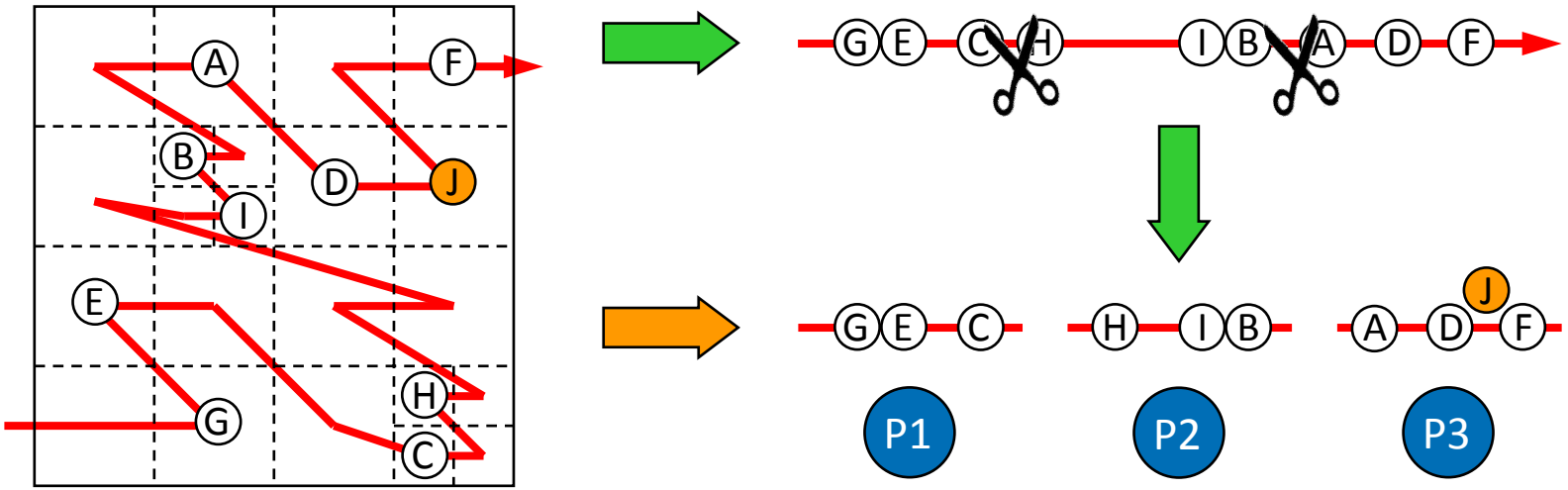


State-of-art: Space-Filling Curves

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

State-of-art: Space-Filling Curves

- load distribution / balancing
  - assign some iteration of SFC to points in *n*D-space
  - linearise data according to SFC
  - simple partitioning of data (preserving locality) to processors possible



- what to do in case of AMR or data (‘J’) newly inserted into image space?

- overview
  - terms and definitions ✓
  - process interaction on UMA / NUMA architectures ✓
  - process interaction on NORMA architectures ✓
  - putting everything together: an example ✓
  - load balancing ✓
  - state-of-art: space-filling curves ✓