Parallel Programming 2: MTP

9th SimLab Course on Parallel Numerical Simulation

October 3—9, 2010, Belgrade, Serbia

Ralf-Peter Mundani
Technische Universität München

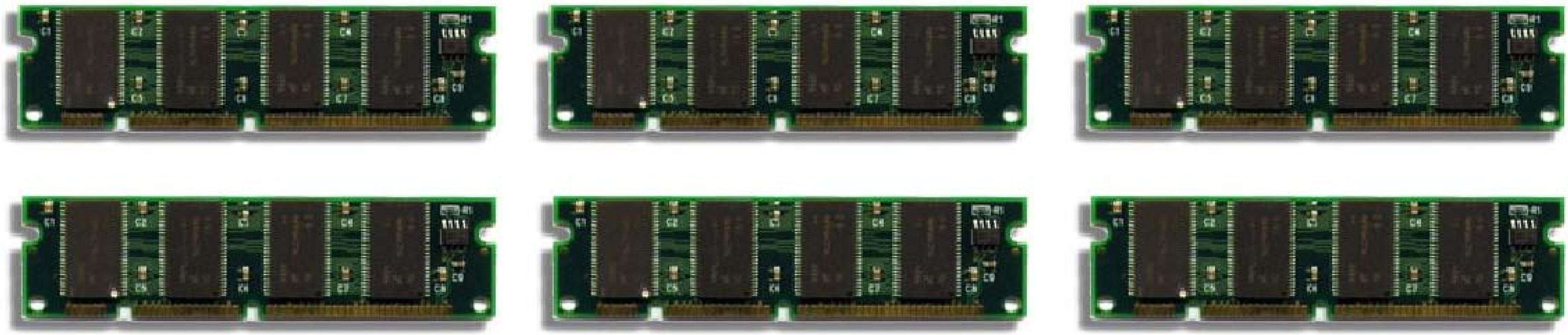
Stabilitätspakt für Südosteuropa
Gefördert durch Deutschland
Stability Pact for South Eastern Europe
Sponsored by Germany

IGSSE TUM
International Graduate School
of Science and Engineering

Wishes…

- what computer scientists want

- what engineers really need

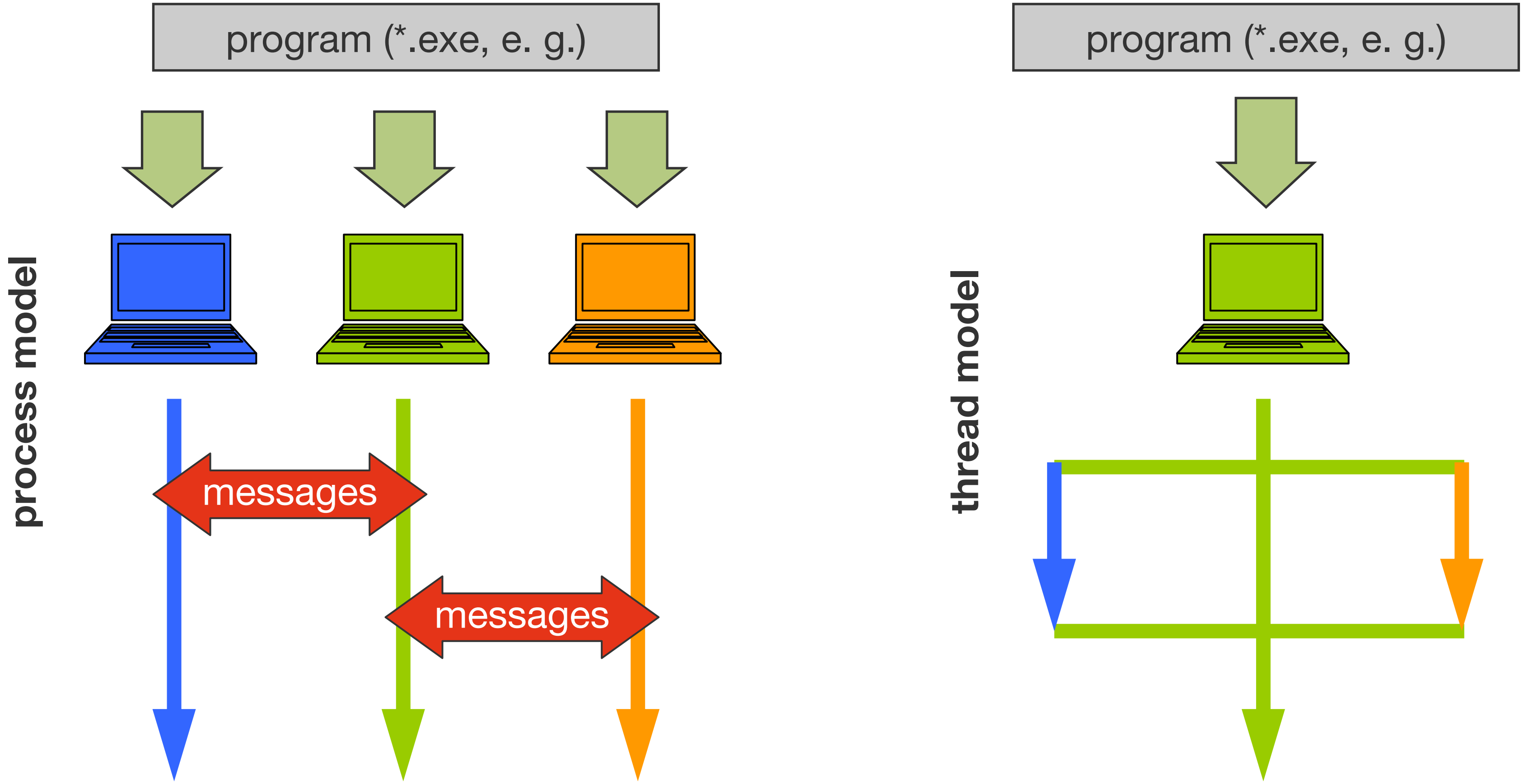


Overview

- process interaction
- synchronisation techniques
- program verification
- programming with OpenMP

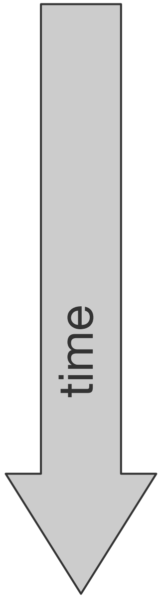
Process Interaction

- difference between processes and threads



Process Interaction

- 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

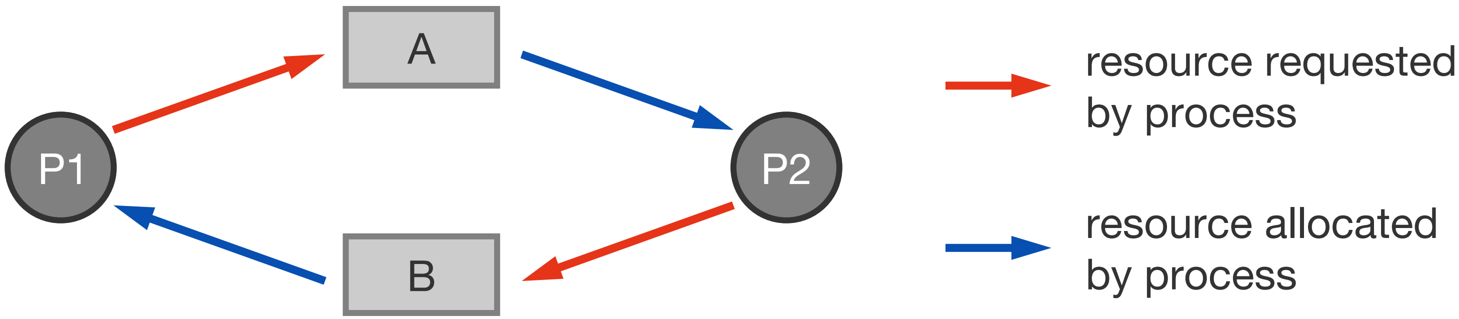
- principles
  - processes depend from each other if they have to be executed in a certain order; this can have two reasons
    - *cooperation*: processes execute parts of a common task
      - producer/consumer: one process generates data to be processed by another one
      - client/server: same as 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
  - realised via shared variables with read / write access

Process Interaction

- 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

- 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



Overview

- process interaction
- synchronisation techniques
- program verification
- programming with OpenMP

Synchronisation Techniques

- 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 $S = S - 1$
      else the processes executing P(S) will be suspended
    - V(S): $S = S + 1$
  - 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" (similar to lock variable, but P and V can be executed by different processes)
  - *general semaphore*: has any nonnegative number

Synchronisation Techniques

- 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 = 1
execute p1 and p2 in parallel

begin procedure p1
    while (true) do
        P(s)
        critical section 1
        V(s)
    od
end
```

```
begin procedure p2
    while (true) do
        P(s)
        critical section 2
        V(s)
    od
end
```

Synchronisation Techniques

- 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
```

Synchronisation Techniques

- 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

Overview

- process interaction
- synchronisation techniques
- program verification
- programming with OpenMP

Program Verification

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

E.W. Dijkstra

Program Verification

- test case
  - simple reader-writer-problem

```
      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
        }
      }
```

Program Verification

- questions
  - **deadlock**
    - program will *always continue*
  - **mutex**
    - resource is *never used by both* processes at any time
  - **liveness**
    - resource *will be used* by any process

```
    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
      }
    }
```

Program Verification

- questions
  - **deadlock**
    - program will always continue
  - **mutex**
    - resource is never used by both processes at any time
  - **liveness**
    - resource will be used by any process
- status variables

  (x, $pc0$, $pc1$)

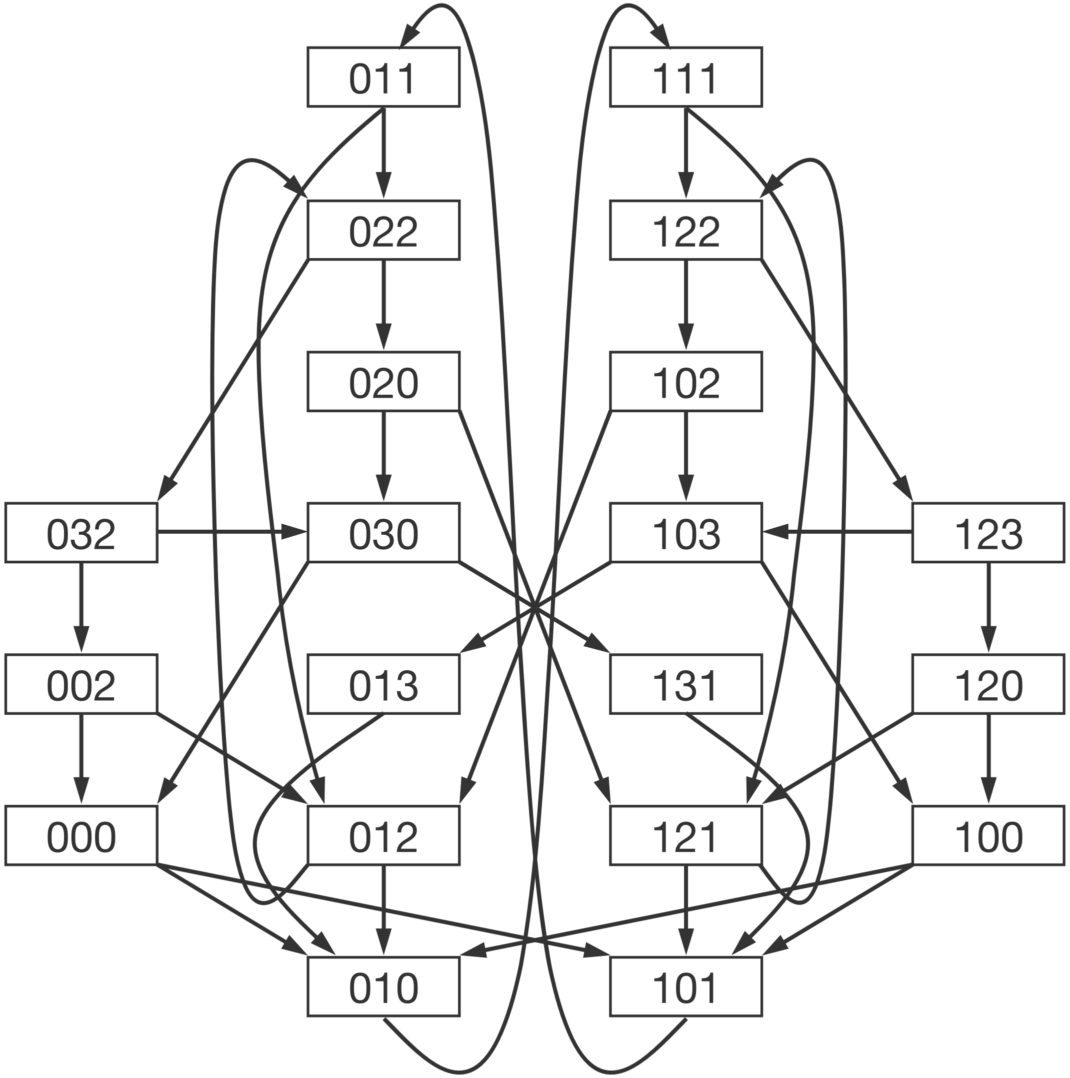
  ➔ hence, 32 states (10 not reachable)

```
        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
          }
        }
```

Program Verification

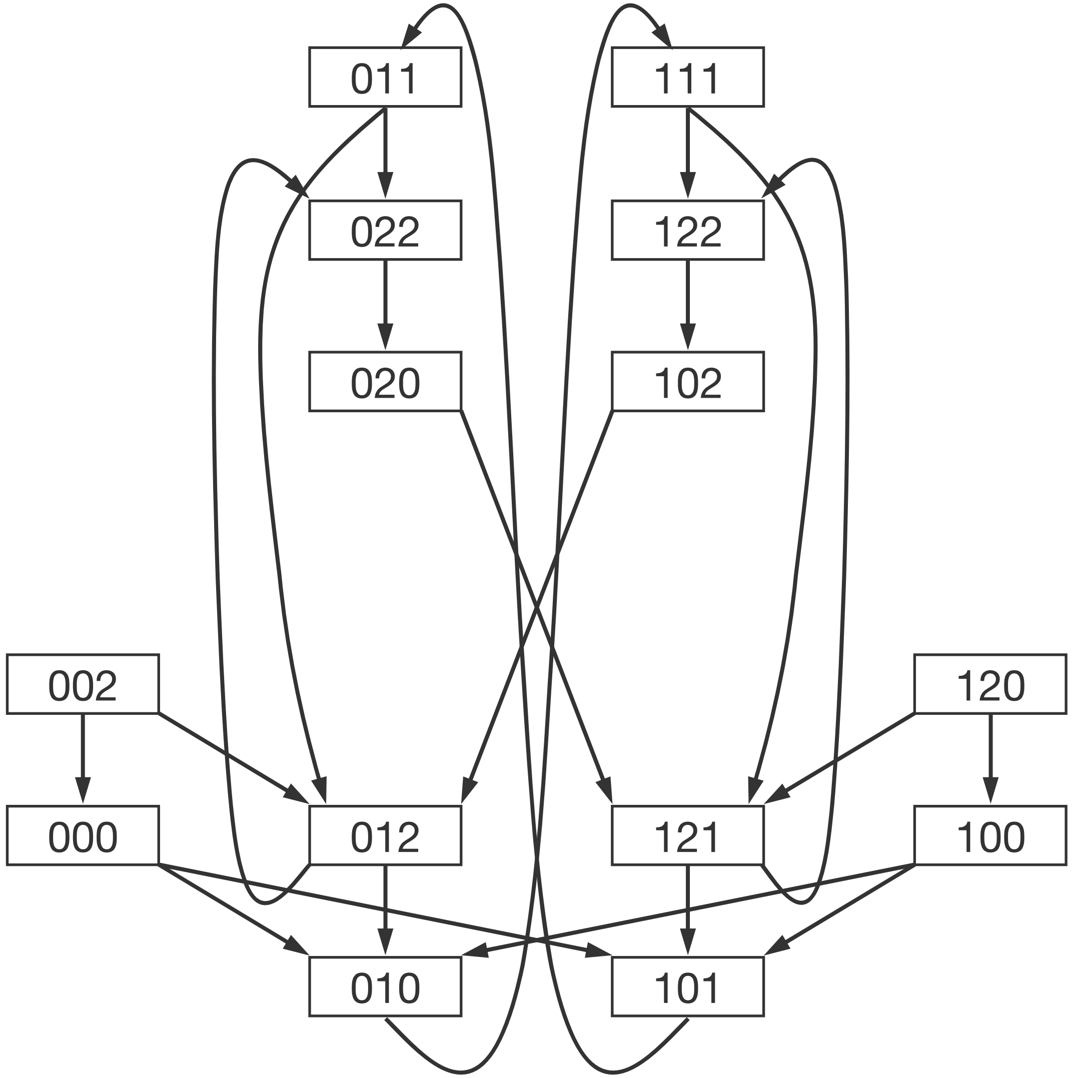



```
     boolean x ← 0

     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
        }
     }
```

Program Verification

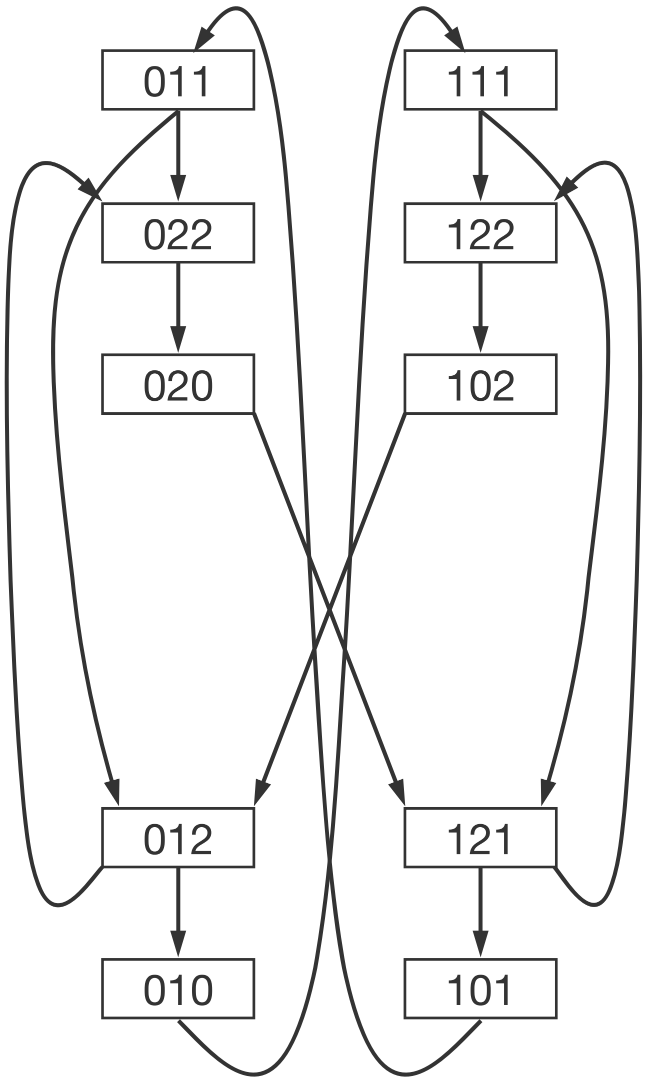



```
     boolean x ← 0

     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
        }
     }
```

Program Verification

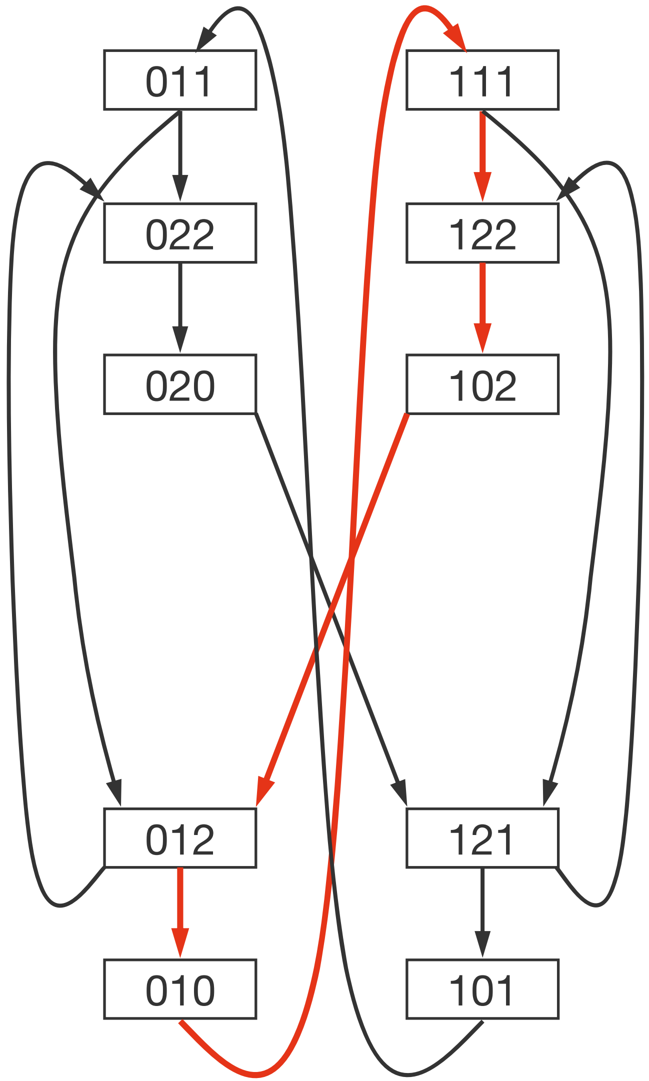



```
     boolean x ← 0

     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
        }
     }
```

Program Verification

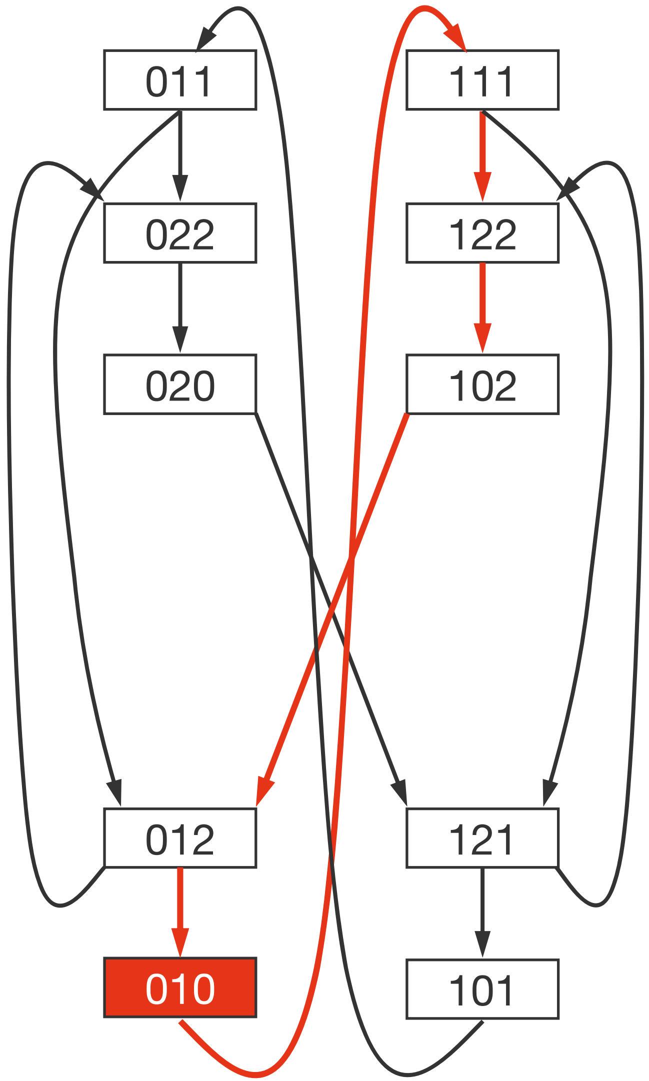



```
boolean x ← 0

proc rw0 {
   while (true) {
      x ← 0
      sync ( )
      if (x = 0)
         use_resource
   }
}

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

Program Verification

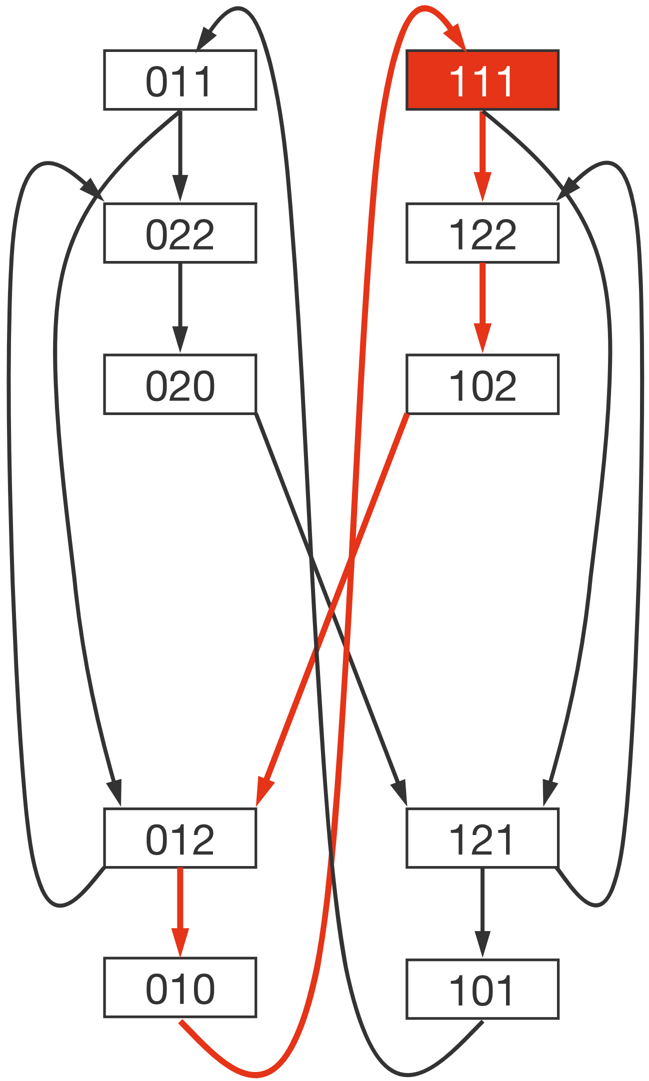



```
        boolean x ← 0

        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
          }
        }
```

Program Verification

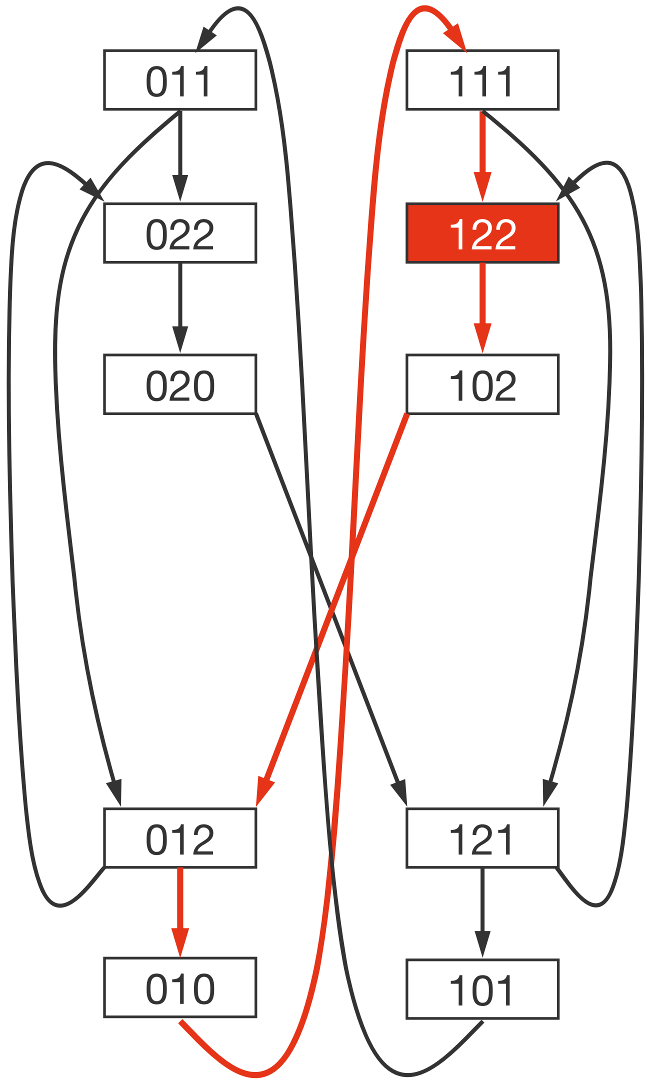



```
        boolean x ← 0

        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
           }
        }
```

Program Verification

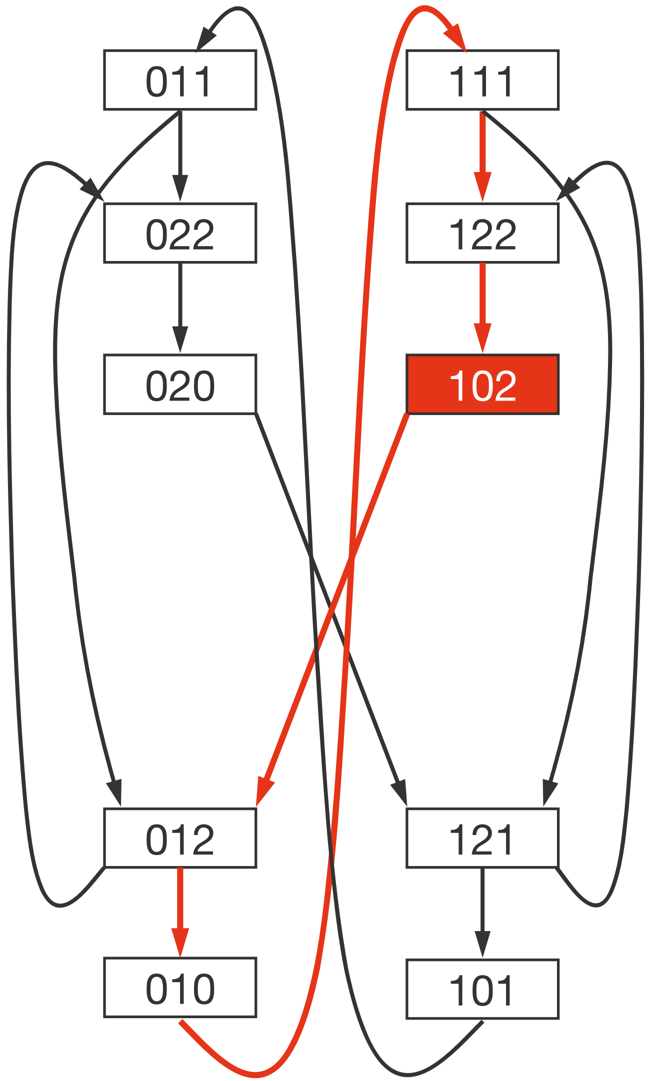



```
        boolean x ← 0

        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
          }
        }
```

Program Verification

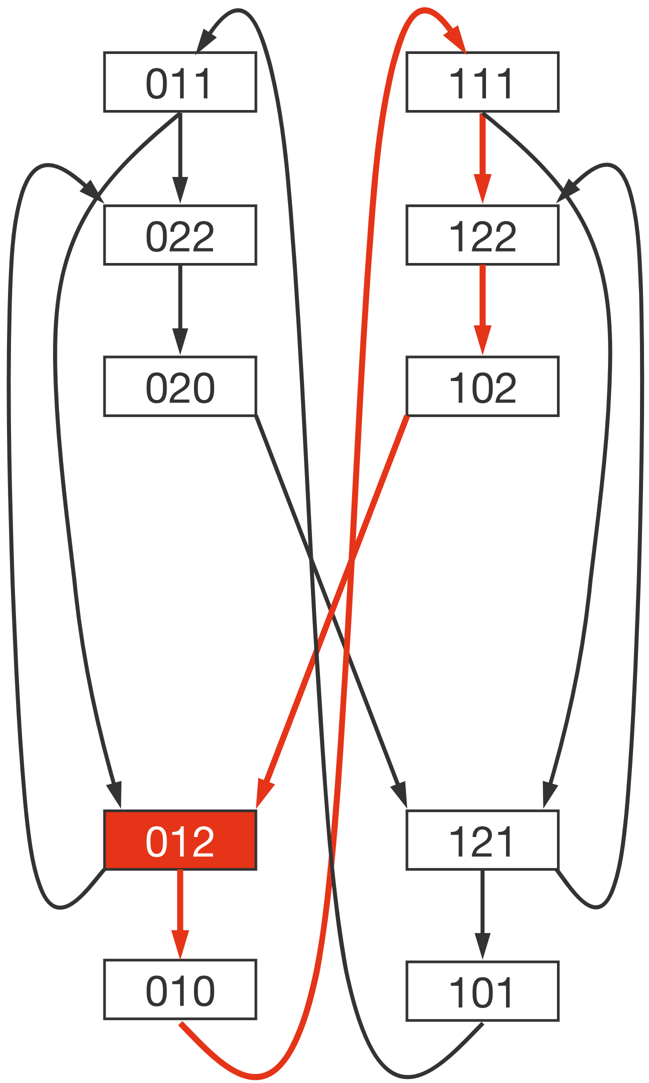



```
boolean x ← 0

proc rw0 {
  while (true) {
    x ← 0
    sync ( )
    if (x = 0)
      use_resource
  }
}

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

Program Verification



boolean $x \leftarrow 0$

proc *rw0* {
  while (**true**) {
0:    $x \leftarrow 0$
1:    **sync** ( )
2:    **if** ($x = 0$)
3:      *use_resource*
  }
}

proc *rw1* {
  while (**true**) {
0:    $x \leftarrow 1$
1:    **sync** ( )
2:    **if** ($x = 1$)
3:      *use_resource*
  }
}

Overview

- process interaction
- synchronisation techniques
- program verification
- programming with OpenMP

Programming with OpenMP

- brief overview
  - OpenMP is an application programming interface (API) for writing multithreaded programs, consisting of
    - a set of compiler directives
    - (runtime) library routines
    - environment variables
  - available for C, C++, and Fortran
  - suited for programming
    - UMA and SMP systems
    - DSM / VSM systems (i. e. NUMA, ccNUMA, and COMA)
    - hybrid systems (i. e. MesMS with shared-memory nodes) in combination with message passing (MPI, e. g.)
  - further information: http://www.openmp.org

Programming with OpenMP

- compiler directives
  - prototypical form of compiler directives (C and C++)

```
#pragma omp directive-name [clause, …]  newline
```

- *directive-name*: a valid OpenMP directive such as
  - parallel
  - for, sections, single
  - master, critical, barrier
  - …
- *clause*: optional statements such as
  - if
  - private, firstprivate, lastprivate, shared
  - reduction
  - …

Programming with OpenMP

- parallel region construct

  `#pragma omp parallel` ***`[clause, …]  newline`***

  - precedes a parallel region (i. e. structured block of code) that will be executed by multiple threads
  - when a thread reaches a "parallel" directive, it creates a team of threads and becomes the master of that team
  - code is duplicated and all threads will execute that code
  - implicit barrier at the end of parallel region
  - it is illegal to branch into or out of a parallel region
  - amount of threads set via **`omp_set_num_threads()`** library function or OMP_NUM_THREADS environment variable
  - threads numbered from 0 (master thread) to N–1

Programming with OpenMP

- parallel region construct (cont'd)
  - some clauses
    - *if (condition)*: must evaluate to TRUE in order for a team of threads to be created; only a single "if" clause is permitted
    - *private (list)*: listed variables are private to each thread; variables are uninitialised and not persistent (i. e. they do not longer exist when the parallel region is left)
    - *shared (list)*: listed variables are shared among all threads
    - *default (shared | none)*: default value for all variables in a parallel region
    - *firstprivate (list)*: like private, but listed variables are initialised according to the value of their original objects

Programming with OpenMP

- parallel region construct (cont'd)
  - example

```
#include <omp.h>

main () {
    int nthreads, tid;
    #pragma omp parallel private (tid)
        {
            tid = omp_get_thread_num ();
            if (tid == 0) {
                nthreads = omp_get_num_threads ();
                printf (“%d threads running\n”, nthreads);
            } else {
                printf (“thread %d: Hello World!\n”, tid);
            }
        }
}
```

Programming with OpenMP

- work-sharing constructs
  - divides the execution of the enclosed code region among the members of the team that encounter it
  - work-sharing constructs do not launch new threads
  - there is no implied barrier upon entry of a work-sharing constructs, only at the end
  - different types of work-sharing constructs
    - *for*: shares iterations of a loop (➔ data parallelism)
    - *sections*: work is broken down into separate sections, each to be executed by a thread (➔ function parallelism)
  - must be encountered by all members of a team or none at all

Programming with OpenMP

- work-sharing constructs (cont'd)

  ```
  #pragma omp for [clause, …]  newline
  ```

  - iterations of the loop immediately following the "for" directive to be executed in parallel (only in case a parallel region has already been initiated)
  - to branch out of a loop (break, return, exit, e. g.) associated with a "for" directive is illegal
  - program correctness must not depend upon which thread executes a particular iteration
  - some clauses
    - *lastprivate (list)*: like private, but values of listed variables are copied back at the end into their original variables
    - *nowait*: threads do not synchronise at the end of loop

Programming with OpenMP

- work-sharing constructs (cont'd)
  - example

```
main () {
    int i;
    float a[N], b[N], c[N];

    …

    #pragma omp parallel shared (a, b, c) private (i)
        {
        #pragma omp for nowait
            for (i = 0; i < N; ++i)
                c[i] = a[i] + b[i];
        }
    …
}
```

Programming with OpenMP

- synchronisation constructs

```
#pragma omp master  newline
```

  - specifies a region that is only to be executed by the master
  - there is no implied barrier associated with this directive
  - to branch into or out of a master block is illegal

```
#pragma omp critical [name]  newline
```

  - specifies a region of code that must be executed by only one thread at a time; threads trying to enter critical region are blocked until they get permission
  - optional name enables multiple critical regions to exist
  - to branch into or out of a critical region is illegal

Programming with OpenMP

- synchronisation constructs (cont'd)

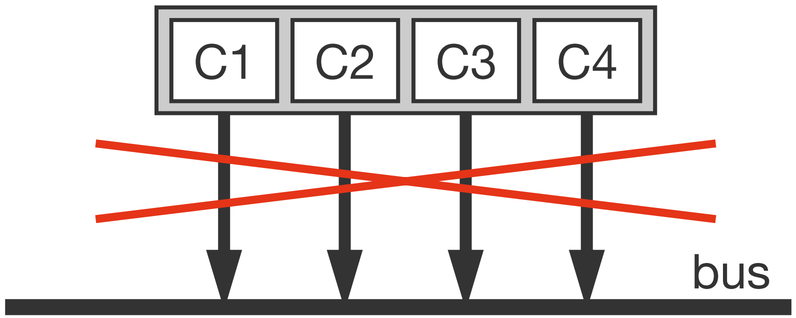
```
#pragma omp barrier  newline
```

  - synchronises all threads, i. e. before resuming execution a thread has to wait at that point until all other threads have reached that barrier, too

```
#pragma omp atomic  newline
```

  - specifies the atomic update of a specific memory location
  - applies only to a single, immediately following statement
  - example

```
#pragma omp atomic
    x = x + 1;
```



Programming with OpenMP

- runtime library

```
void omp_set_num_threads (int num_threads)
```

  - sets the number of threads that will be used in the next parallel region; it has precedence over the OMP_NUM_THREADS environment variable
  - can only be called from serial portions of the code

```
int omp_get_num_threads (void)
int omp_get_max_threads (void)
```

  - returns
    - the number of threads that are currently executing in the parallel region from which it is called
    - the maximum number of threads that can be active

Programming with OpenMP

- runtime library (cont'd)

  `int omp_get_thread_num (void)`

  - returns the number ($0 \leq TID \leq N-1$) of the thread making this call, the master thread has number "0"

  `int omp_in_parallel (void)`

  - may be called to determine if the section of code which is executing is parallel or not ➔ returns a non-zero integer if parallel, and zero otherwise

  - further runtime library routines available, see OpenMP specification (http://www.openmp.org) for details

謝謝