High Performance Computing – Programming Paradigms and Scalability

Part 4: Shared-Memory Programming

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

Summer Term 2015

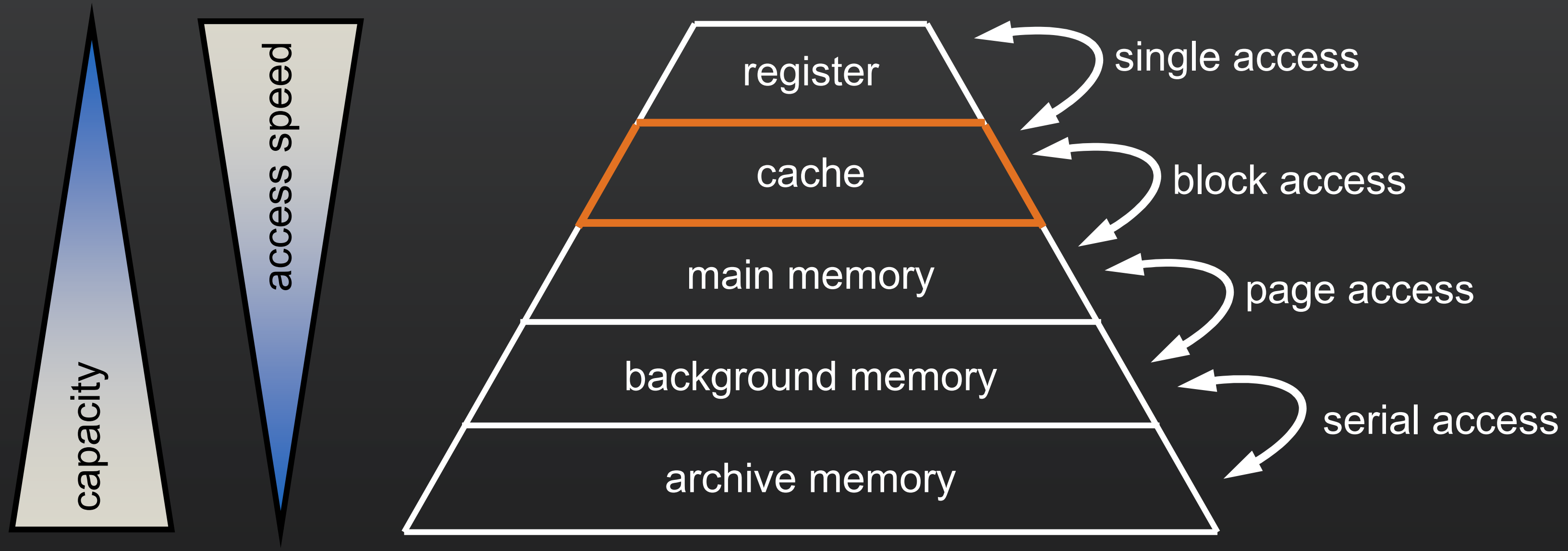
Overview

- cache coherence
- memory consistency
- dependence analysis
- programming with OpenMP

Technology is dominated by two types of people:
those who understand what they do not manage,
and those who manage what they do not understand.
—Archibald Putt

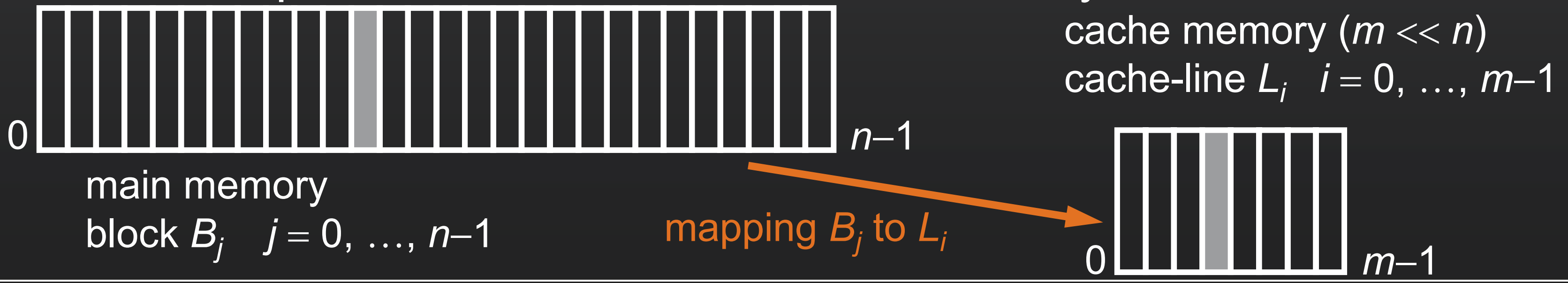
Cache Coherence

- reminder: cache
  - memory hierarchy
    - exploitation of program characteristics such as locality
    - compromise between costs and performance
    - components with different speeds and capacities



Cache Coherence

- reminder: cache (cont'd)
  - cache memory
    - fast access buffer between main memory and processor
    - provides copies of current (main) memory content for fast access during program execution
  - cache management
    - tries to provide always those data that processor needs for the next computation step
    - due to small capacity certain strategies for load and update operations of cache content necessary



Cache Coherence

- reminder: cache (cont'd)
  - for any memory access the cache controller checks if
    - the respective memory content has a copy stored in cache (1)
    - this cache entry is labelled as valid (2)
  - checkup leads to a
    - cache hit: (1) and (2) are fulfilled ➔ access served by cache
    - cache miss: (1) and / or (2) are not fulfilled
      - *read miss*
        - data is read from memory and a copy stored in cache
        - cache entry is labelled as valid
      - *write miss*: update strategy decides whether
        - the respective block is loaded (from memory) into cache and becomes updated due to write access
        - only memory is updated and cache stays unmodified

Cache Coherence

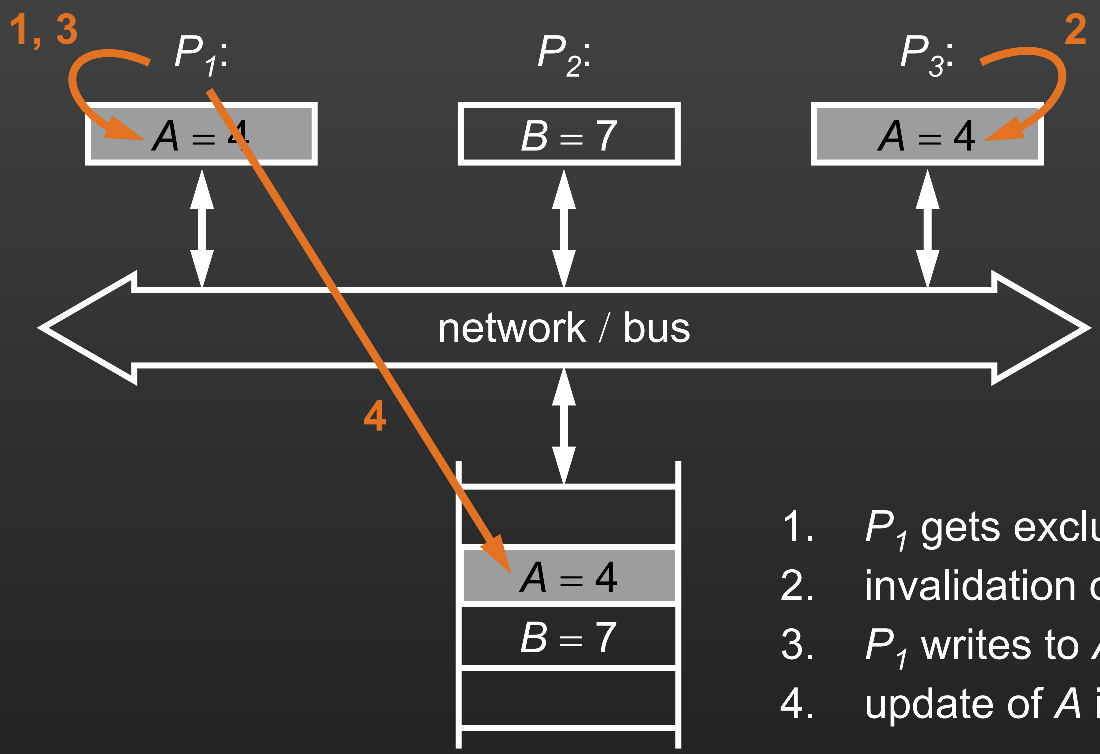
- definitions
  - processors with local cache that have independent access to a shared memory cause validity problems, i.e. several copies of the same memory block exist that contain different values
  - cache management is called
    - *coherent*: a read access always provides a memory block's value from its last write access
    - *consistent*: all copies of a memory block in main memory and local caches are identical (i.e. coherence implicitly given)
  - inconsistencies between cache and main memory occur when updates are only performed in cache but not in main memory (so called *copy-back* or *write-back cache policy*, in contrast to the *write-through cache policy*)
  - drawback: consistency is very expensive

Cache Coherence

- definitions (cont'd)
  - hence, inconsistencies (to some extent) can be acceptable if at least cache coherence is assured (temporary variables, e.g.)
    - *write-update protocol*
      - an update of a copy in one cache requires also the update of all other copies in other caches
      - update can be delayed, at the latest with next access
    - *write-invalidate protocol*
      - exclusive write access of a processor to shared data that should be updated has to be assured
      - before the update of a copy in one cache all other copies in other caches are labelled as invalid
  - in general, write-invalidate / copy-back used for SMP systems

Cache Coherence

- definitions (cont'd)
  - example: write-invalidate protocol / write-through cache policy



1. P_1 gets exclusive access for A
2. invalidation of other copies of A
3. P_1 writes to A
4. update of A in main memory

Cache Coherence

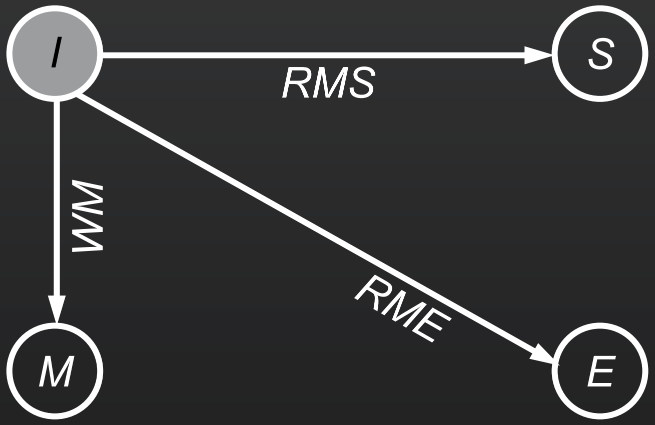
- bus snooping
  - processors with cache are attached to shared memory via a bus
  - each processor “listens” to all addresses sent over the bus by other processors and compares them to its own cache-lines
  - in case one cache-line matches this address, bus logic executes the following steps dependent from the cache-line’s state
    - *unmodified cache-line*: if a write access should be performed the cache-line becomes invalid
    - *modified cache-line*
      - bus logic interrupts the transaction and writes the modified cache-line to the main memory
      - afterwards, the initial transaction is executed again
  - MESI protocol frequently used with bus snooping

Cache Coherence

- MESI protocol
  - cache coherence protocol (write-invalidate) for bus snooping
  - each cache-line is assigned one of the following states
    - *exclusive modified (M)*: cache-line is the only copy in any of the caches and was modified due to a write access
    - *exclusive unmodified (E)*: cache-line is the only copy in any of the caches and was transferred for read access
    - *shared unmodified (S)*: copies of this cache-line reside in more than one cache and were transferred for read access
    - *invalid (I)*: cache-line is invalid
  - for write-through cache policy only the states shared unmodified and invalid are relevant

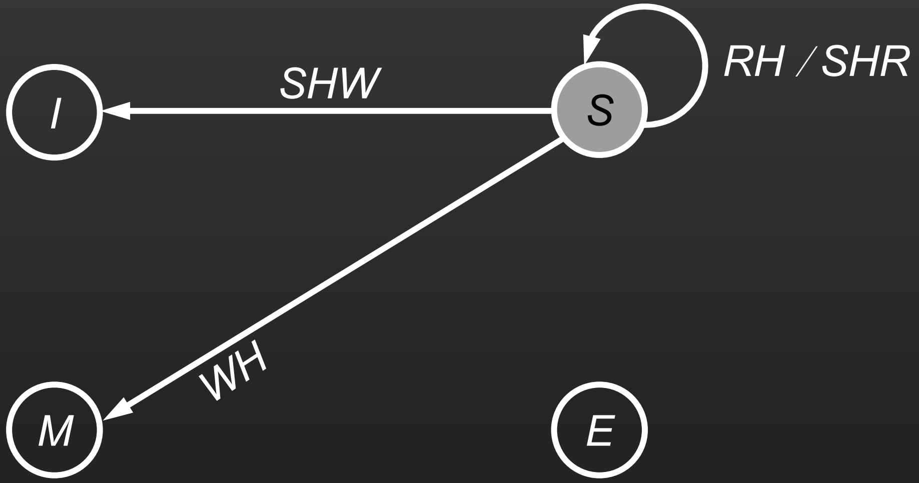
Cache Coherence

- MESI protocol (cont'd)
  - state: invalid
    - due to read / write access a valid copy is loaded into cache
    - other processes (snoop hit on a read) send signal SHARED if they have a valid copy
    - read miss: read miss shared *(RMS)* or read miss exclusive *(RME)* leads to state transition to *S* or *E*, resp.
    - write miss *(WM)*: transition to *M*


$$\begin{tikzcd} I \arrow[r, "RMS"] \arrow[d, "WM"] \arrow[rd, "RME"] & S \\ M & E \end{tikzcd}$$

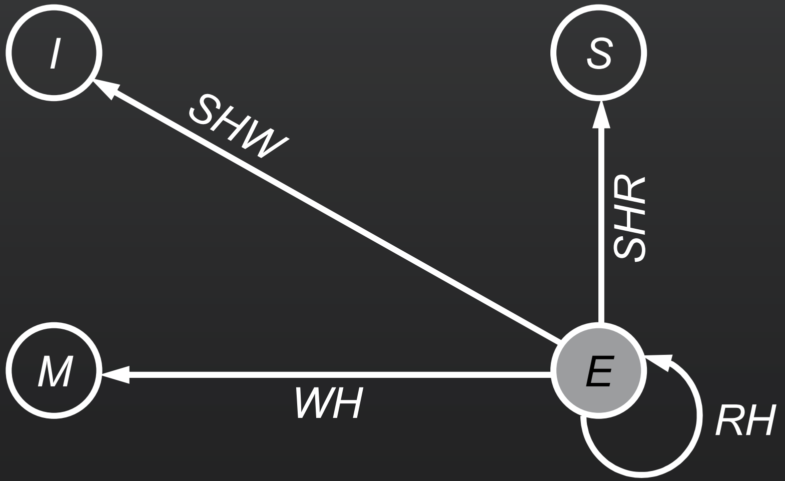
Cache Coherence

- MESI protocol (cont'd)
  - state: shared unmodified
    - read hit *(RH)* / snoop hit on a read *(SHR)*: state is unchanged; process sends signal SHARED in case of *SHR*
    - write hit *(WH)*: state transition to *M*
    - snoop hit on a write *(SHW)*: state transition to *I*



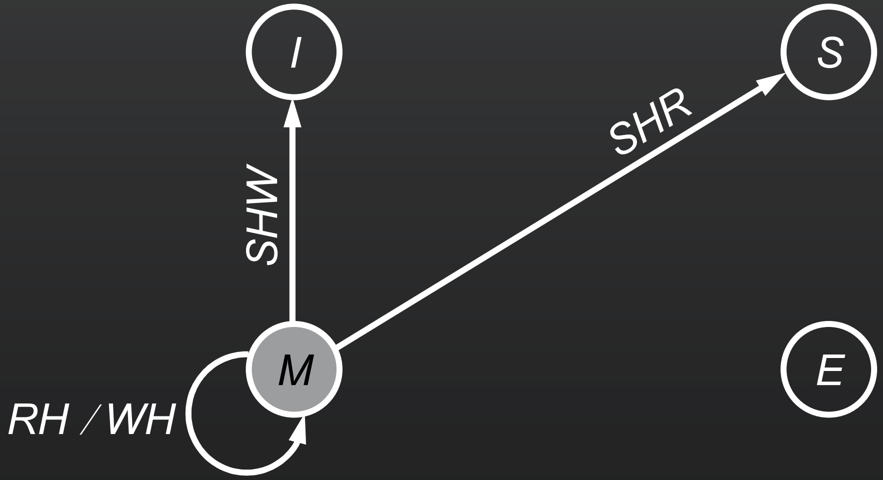
Cache Coherence

- MESI protocol (cont'd)
  - state: exclusive unmodified
    - *RH*: state is unchanged; no bus usage necessary
    - *SHR*: process sends signal SHARED; state transition to *S*
    - *SHW*: state transition to *I*
    - *WH*: state transition to *M*; no bus usage necessary



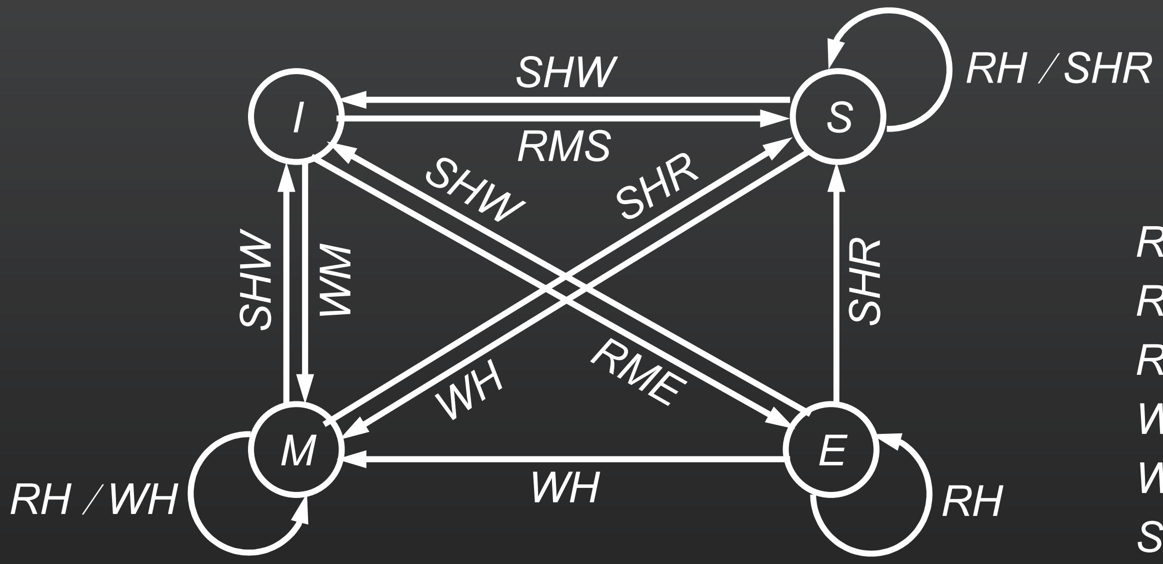
Cache Coherence

- MESI protocol (cont'd)
  - state: exclusive modified
    - *RH* / *WH*: state is unchanged; no bus usage necessary
    - *SHR* / *SHW*: other process is notified via signal RETRY that a copy-back of this cache-line to main memory is necessary; state transition to *I* or *S* in case of *SHW* or *SHR*, resp.



Cache Coherence

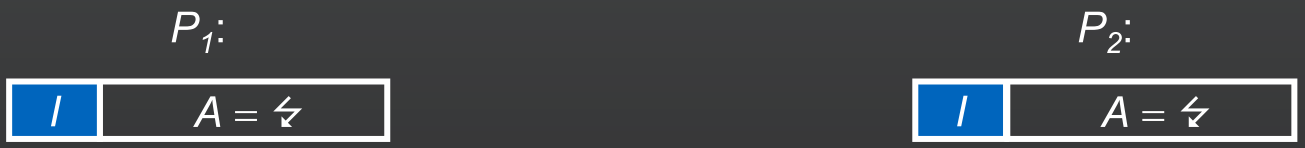
- MESI protocol (cont'd)
  - putting it all together



RH:	read hit
RMS:	read miss shared
RME:	read miss exclusive
WH:	write hit
WM:	write miss
SHR:	snoop hit on a read
SHW:	snoop hit on a write

Cache Coherence

- MESI protocol (cont'd)
  - example: SMP system with two processors (1)
    - subsequent read / write access to same cache-line



- read miss
- load valid copy from main memory
- state transition $I \rightarrow E$

$\longrightarrow$

- snoop hit on a read

Cache Coherence

- MESI protocol (cont'd)
  - example: SMP system with two processors (2)
    - subsequent read / write access to same cache-line

P_1:

E	$A = 4$

- snoop hit on a read
- send signal SHARED
- state transition $E \rightarrow S$

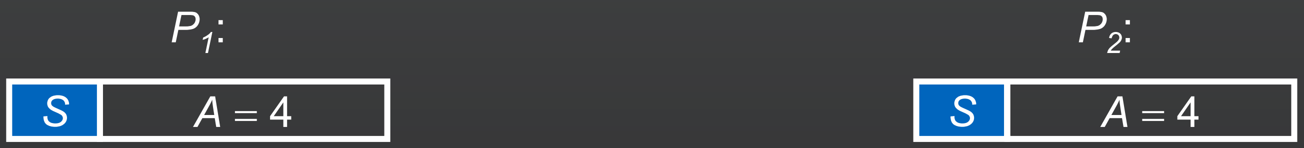
P_2:

I	$A = \lightning$

- read miss
- load valid copy from main memory
- state transition $I \rightarrow S$

Cache Coherence

- MESI protocol (cont'd)
  - example: SMP system with two processors (3)
    - subsequent read / write access to same cache-line



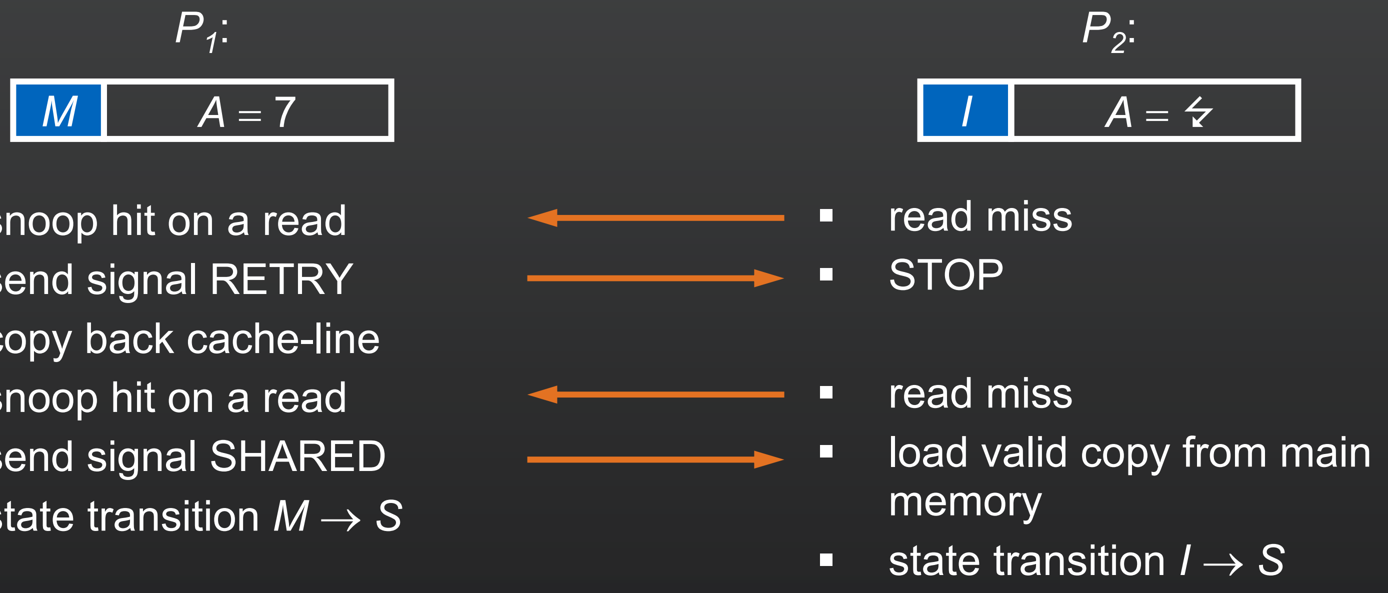
- write hit
- update cache-line
- state transition $S \rightarrow M$

$\longrightarrow$

- snoop hit on a write
- state transition $S \rightarrow I$

Cache Coherence

- MESI protocol (cont'd)
  - example: SMP system with two processors (4)
    - subsequent read / write access to same cache-line



P_1:
- snoop hit on a read
- send signal RETRY
- copy back cache-line
- snoop hit on a read
- send signal SHARED
- state transition $M \rightarrow S$

P_2:
- read miss
- STOP
- read miss
- load valid copy from main memory
- state transition $I \rightarrow S$

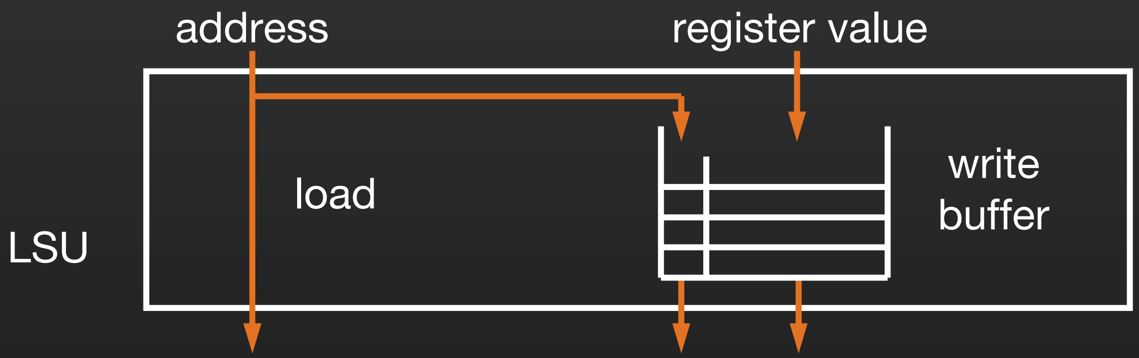
Overview

- cache coherence
- memory consistency [(*)]
- dependence analysis
- programming with OpenMP

[(*)] not to be covered within this lecture

Memory Consistency

- motivation
  - data transfer between cache and registers via load / store unit (LSU) of the processor ➔ cache coherence takes effect at the time when LSU performs read / write access to cache memory
  - in general, modern microprocessors reorder load and store operations for performance improvement, for instance
    - load operations are executed immediately
    - store operations are internally buffered (FIFO)



Memory Consistency

- motivation (cont'd)
  - hence, a subsequent load operation can "pass" a waiting store operation in case they have different addresses (to avoid that a load operation reads obsolete values from cache while current values (still to be written) reside in the write buffer)
  - further improvement: *non-blocking cache* / *lock-up free cache*
    - in case of a cache miss, execution can be continued with subsequent operations (accessing different cache-lines) without waiting for the blocked operation to be finished
  - consequence of both strategies: modified order of execution
  - nevertheless, due to local address comparison of affected values there is no impact on the computed result of a program on the monoprocessor system
  - what happens on the multiprocessor systems?

Memory Consistency

- motivation (cont’d)
  - even cache coherence is assured for SMP systems, reordering of operations and / or non-blocking caches might lead to unwanted results during program execution
  - example (DEKKER’s algorithm)

```
x ← 0; y ← 0                         x ← 0; y ← 0
process P1:                          process P2:
    x ← 1                                y ← 1
    if y = 0 then A1 fi                  if x = 0 then A2 fi
```

  - four different possibilities
    - *A1* will be executed, *A2* will not be executed
    - *A2* will be executed, *A1* will not be executed
    - *A1* and *A2* will not be executed
    - *A1* and *A2* will be executed (unexpected from programmer)

Memory Consistency

- motivation (cont'd)
  - further problems for DSM / VSM systems
  - intuitively we would expect that
    - updates of variables take effect everywhere at the same time
    - temporal order of memory accesses is retained
  - but in reality
    - we would need a global clock with very high precision
    - write operations are not atomic (i.e. new values don't take effect everywhere at the same time)
    - write accesses have different latencies due to network ➔ race between single memory accesses (a local / remote read subsequent to a write might get different values, e.g.)
  - hence, further thoughts about memory consistency are necessary

Memory Consistency

- notation
  - one line for each processor's memory accesses
  - time proceeds from left to right
  - memory / synchronisation operations
    - *R*(*X*)*val*: read variable *X*, obtain value "*val*"
    - *W*(*X*)*val*: write value "*val*" to variable *X*
    - ▶: synchronisation point
    - *AQ*(*L*): acquire lock *L* for entering critical section
    - *RL*(*L*): release lock *L* for leaving critical section
  - all variables are assumed to be initialised to 0
  - example ($\approx x \leftarrow x + 1$)

```
P1: R(x)0 W(x)1 R(x)1
----------------------->
```

Memory Consistency

- reminder: strict consistency
  - definition: any read on a data item *X* returns a value corresponding to the result of the most recent write on *X*
  - main aspect is precise serialisation on all memory accesses
  - example: C) is not valid under strict consistency

```
   P1: W(x)1                                P1:         W(x)1
A) ----------------------->             B)  ----------------------->
   P2:         R(x)1  R(x)1                 P2: R(x)0           R(x)1


   P1: W(x)1
C) ----------------------->
   P2:         R(x)0  R(x)1
```

Memory Consistency

- sequential consistency
  - slightly weaker model than strict consistency
  - definition by LAMPORT (1979)

    “The result of any execution is the same as if the operations of all the processors were executed [on a monoprocessor] in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.”

  - that means
    - order of operations to be retained on individual processor
    - any overlap of orders of operations is acceptable as long as the same overlap is visible on each processor
    - no global clock necessary

Memory Consistency

- sequential consistency (cont'd)
  - example: D) is not valid under sequential consistency

```
    P1: W(x)1 W(x)2                 P1:             W(x)1
A)  ----------------------->    B)  ----------------------->
    P2:       R(x)0 R(x)2           P2: R(x)0 R(x)1


    P1: W(x)1                       P1: W(x)1
    ----------------------->        ----------------------->
    P2:       R(x)1 R(x)2           P2:       R(x)2 R(x)1
C)  ----------------------->    D)  ----------------------->
    P3:       R(x)1 R(x)2           P3:       R(x)1 R(x)2
    ----------------------->        ----------------------->
```

Memory Consistency

- sequential consistency (cont'd)
  - consequences
    - sequential consistent memory is very easy to use but it also entails very high costs / drawbacks due to
      - only an overlapping execution of sequential operations instead of a complete parallel execution
      - strong limitations as reordering of operations / non-blocking caches are forbidden
      - very inefficient in case of frequent write accesses
    - semantic too strong for most problems ➔ weaker models necessary that are reasonable to use and easy to implement
    - furthermore, sequential consistency assures correct order of memory accesses but not correct access to shared data objects ➔ still synchronisation via programmer necessary

Memory Consistency

- sequential consistency (cont'd)
  - caution: cache coherence ≠ sequential consistency
  - cache coherence only requires a locally consistent view, i.e.
    - access to different memory locations might be seen in different orders
    - access to the same memory location is globally seen in the same order
  - sequential consistency requires a globally consistent view
  - example: valid under cache coherence only

```
P1: W(x)1 R(y)0
----------------->
P2: W(y)1 R(x)0
```

Memory Consistency

- causal consistency
  - weaker model than sequential consistency
  - definition by HUTTO and AHAMAD (1990)

    “Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.”

  - hence, write $w(t_2)$ at time t_2 is potentially dependent on write $w(t_1)$ at time t_1 (with $t_1 \leq t_2$), when there is a read between these two writes which may have influenced write $w(t_2)$ ➔ if $w(t_2)$ causally depends on $w(t_1)$ then only correct sequence is $w(t_1) \rightarrow w(t_2)$
  - implementing causal consistency requires keeping track of which processes have seen which writes ➔ construction and maintenance of a dependence graph

Memory Consistency

- causal consistency (cont'd)
  - example: B) is not valid under causal consistency

```
    P1:  W(x)1                W(x)3
    ----------------------------------------------->
    P2:       R(x)1  W(x)2
A)  ----------------------------------------------->
    P3:                             R(x)1  R(x)3  R(x)2
    ----------------------------------------------->
    P4:                             R(x)1  R(x)2  R(x)3


    P1:  W(x)1
    ----------------------------------->
    P2:       R(x)1  W(x)2
B)  ----------------------------------->
    P3:                       R(x)2  R(x)1
    ----------------------------------->
    P4:                       R(x)1  R(x)2
```

Memory Consistency

- processor consistency
  - also referred to as PRAM (pipelined RAM) consistency
  - definition by GOODMAN (1989)

    "A multiprocessor is said to be processor consistent if the result of any execution is the same as if the operations of each individual processor appear in the sequential order specified by its program."

  - difference to sequential consistency
    - order of operations for all processors *must not be uniform*, i.e. write accesses of two processors might be seen in a different sequence from a third processor than from the previous ones
    - however, write accesses of one processor are seen by all others in the order specified by its program
    - this better reflects the reality of networks due to latency

Memory Consistency

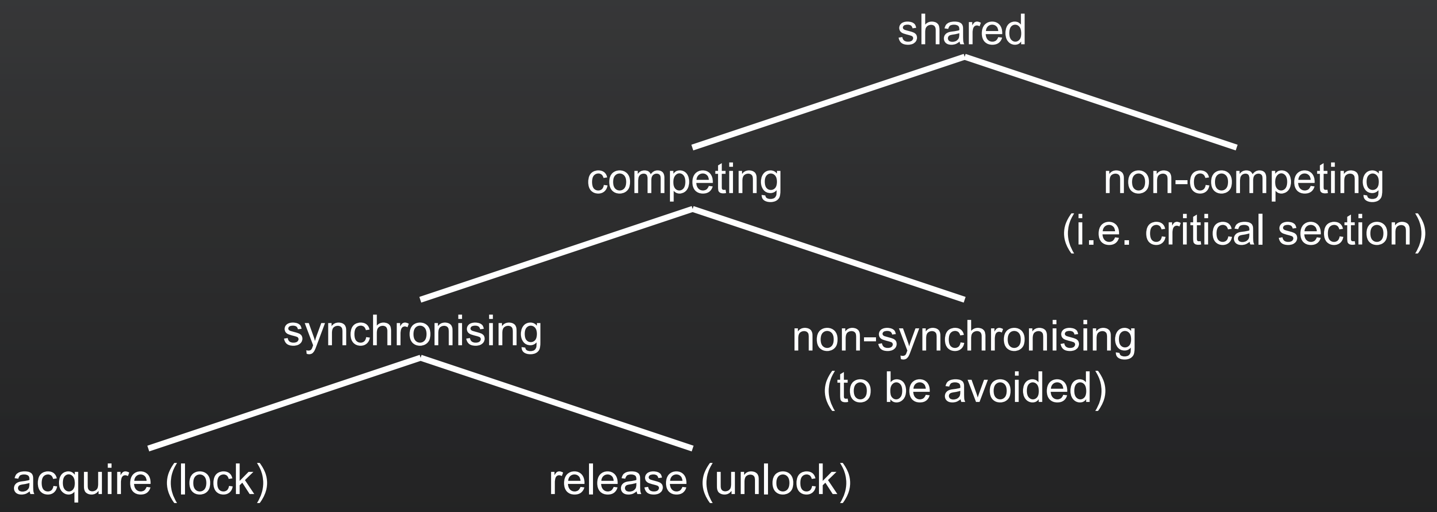
- processor consistency (cont'd)
  - example: B) is not valid under processor consistency

```
    P1:  W(x)1
    ----------------------------------------->
    P2:       R(x)1 W(x)2 W(x)3
A)  ----------------------------------------->
    P3:                       R(x)2 R(x)1 R(x)3
    ----------------------------------------->
    P4:                       R(x)1 R(x)2 R(x)3


    P1:  W(x)1 W(x)2
B)  ------------------------->
    P2:            R(x)2 R(x)1
```

Memory Consistency

- weak consistency
  - in general, access to shared data will be protected via mutual exclusion (i.e. obtain access, manipulate data, relinquish access); other processes don't need to see the intermediate values, they only need to see the final values
  - classification of shared memory accesses (GHARACHORLOO)



Memory Consistency

- weak consistency (cont'd)
  - conditions to be fulfilled for weak consistency
    1) accesses to synchronisation variables (associated with a write operation) are sequentially consistent
    2) no access to a synchronisation variable is allowed to be performed until all preceding write operations have completed everywhere
    3) no read / write operation is allowed to be performed until all preceding accesses to synchronisation variables have been performed
  - hence, accesses to synchronisation variables are visible for all processes in the same order (1)
  - all write operations have been completed everywhere (2)
  - all copies are up-to-date according to the synchronisation point (3)

Memory Consistency

- weak consistency (cont'd)
  - in weak consistent memory, modifications are not visible until a synchronisation has been performed
  - a program with properly set synchronisation behaves the same as a program without synchronisation on sequentially consistent memory ➔ de facto sequentially consistent groups of operations
  - example: B) is not valid under weak consistency

```
    P1: W(x)1 W(x)2        ▶
    ------------------------------------->
A)  P2:             R(x)2 R(x)1 ▶ R(x)2
    ------------------------------------->
    P3:             R(x)1 R(x)2 ▶ R(x)2


    P1: W(x)1 W(x)2 ▶
B)  ------------------------------->
    P2:             R(x)1 ▶ R(x)1
```

Memory Consistency

- release consistency
  - synchronisation does not tell if entering or leaving a critical section
  - hence, local changes need to be both propagated to all other processors (sharing a copy) and all other changes need to be consolidated ➔ too much communication
  - release consistency helps to weaken the communication problem
  - idea: consider locks and propagate locked memory only if needed

    1) before a R/W operation on shared data is performed, all preceding acquires done by the process must have completed successfully
    2) before a release is allowed to be performed, all preceding R/W ops done by the process must have been completed
    3) acquire / release accesses are processor consistent

Memory Consistency

- release consistency (cont'd)
  - *eager release consistency*: all changes are propagated via the release operation ➔ still huge communication overhead
  - *lazy release consistency*: all local copies are updated via the acquire operation ➔ complex implementation but avoidance of redundant communication
  - example: B) is not valid under release consistency

```
    P1: AQ(L) W(x)1 W(x)2 RL(L)
    ---------------------------------------------------->
A)  P2:                            AQ(L) R(x)2 RL(L)
    ---------------------------------------------------->
    P3:                                              R(x)1


    P1: AQ(L) W(x)1 W(x)2 RL(L)
B)  ---------------------------------------------->
    P2:                      R(x)1 AQ(L) R(x)1 RL(L)
```

Memory Consistency

- entry consistency
  - further weakening of release consistency
  - problem: all local updates propagated during release of shared data and an acquire has to determine which variables it needs
  - idea: each shared data element to be associated with a synchronisation variable (➔ elements of a shared array can be accessed independently in parallel, e.g.)
  - conditions to be fulfilled for entry consistency
    1) acquire access of a synchronisation variable is not allowed to perform w.r.t. a process until all updates of the guarded shared data have been performed w.r.t. that process
    2) before an exclusive access to a synchronisation variable by a process is allowed, no other process may hold the synchronisation variable, not even in non-exclusive mode

Memory Consistency

- entry consistency (cont'd)

  3) after an exclusive access to a synchronisation variable has been performed, any other process' next non-exclusive mode access to that synchronisation variable may not be performed until it has performed w.r.t. that variable's owner

  - an acquire blocks until all guarded (local) data are up-to-date (1)
  - each process intending to manipulate a guarded variable has to perform an acquire on the respective synchronisation variable (2)
  - an acquire operation for read access after a write access has to obtain a variable's current value from the last writing process (3)

```
P1: AQ(Lx) W(x)1 AQ(Ly) W(y)2 RL(Lx) RL(Ly)
------------------------------------------------------------>
P2:                                 AQ(Lx) R(x)1 R(y)0
------------------------------------------------------------>
P3:                                        AQ(Ly) R(y)2
```

Memory Consistency

- characteristics of different models

consistency		description
w/o synchron.	strict	absolute time ordering of all accesses matters
	sequential	all processes see all accesses in same order; accesses are not ordered in time
	causal	all processes see causally-related accesses in same order
	process	all processes see writes from one process in the order they were used; writes from different processes may not always be seen in that order
w/ synchron.	weak	only after a synchronisation is done data can be counted on to be consistent
	release	data is made consistent when a critical section is exited
	entry	data is made consistent when a critical section is entered

Overview

- cache coherence
- memory consistency
- dependence analysis
- programming with OpenMP

Dependence Analysis

- motivation
  - a program might have execution-order constraints between statements (i.e. instructions) due to dependencies
  - hence, dependence analysis should determine whether or not it is safe to reorder or parallelise these statements
  - topics to be addressed by dependence analysis
    - data dependencies
    - loop dependencies

Dependence Analysis

- data dependencies
  - arise due to competitive access to shared data
  - to be distinguished
    - *flow dependence*: read after write (RAW)
    - *antidependence*: write after read (WAR)
    - *output dependence*: write after write (WAW)
    - *input dependence*: read after read (RAR)
  - data dependencies might lead to inefficiencies, thus preventing optimisation such as out-of-order execution or parallelisation
  - modern tools use dependence graphs, for instance, to find potential problem areas (= cycles within graphs) and examine to see if they can be broken
  - example: KAP preprocessors for C, F77, and F90

Dependence Analysis

- data dependencies (cont'd)
  - flow dependence a.k.a. true dependence (RAW)
    - a statement S_2 is flow dependent on S_1 iff S_1 modifies a resource that S_2 reads and S_1 precedes S_2 in execution
    - example (sequence in a loop)

```
1:     a(i) ← x(i) – 3
2:     b(i) ← a(i) / c(i)
```

    - general problem: flow dependence cannot be avoided
    - here, *a*(*i*) has to be calculated first in line 1 before using it in line 2 ➔ lines 1 and 2 cannot be processed in parallel

Dependence Analysis

- data dependencies (cont'd)
  - antidependence (WAR)
    - a statement S_2 is antidependent on S_1 iff S_2 modifies a resource that S_1 reads and S_1 precedes S_2 in execution
    - example (sequence in a loop)

```
1:    a(i) ← x(i) – 3
2:    b(i) ← a(i+1) / c(i)
```

    - *a*(*i*+1) is first used with its former value in line 2 and only then computed at the next execution of the loop in line 1 ➔ several iterations of the loop cannot be processed in parallel
    - in general, antidependence can be avoided

Dependence Analysis

- data dependencies (cont'd)
  - output dependence (WAW)
    - a statement S_2 is output dependent on S_1 iff S_1 and S_2 modify the same resource and S_1 precedes S_2 in execution
    - example (sequence in a loop)

```
1:     c(i+4) ← b(i) + a(i+1)
2:     c(i+1) ← x(i)
```

    - some value is first assigned to $c(i+4)$ in line 1 and after three executions of the loop a new value is assigned to the same element (in the array) again in line 2 ➔ several iterations of the loop cannot be processed in parallel
    - nevertheless, output dependence can also be avoided

Dependence Analysis

- data dependencies (cont’d)
  - input dependence (RAR)
    - a statement S_2 is input “dependent” on S_1 <u>iff</u> S_1 and S_2 read the same resource and S_1 precedes S_2 in execution
    - example (sequence in a loop)

```
1:    d(i) ← a(i) + 3
2:    b(i) ← a(i+1) / c(i)
```

    - $a(i+1)$ is first used in line 2 and afterwards used again at the next execution of the loop in line 1 ➔ not a dependence in the same line as the others, hence it does not prohibit reordering instructions or parallel execution of lines 1 and 2

Dependence Analysis

- data dependencies (cont'd)
  - removing of *name dependencies*
    - antidependence and output dependence may be removed through renaming of variables
    - example:

```
1:    a ← 2 * x                          1:    c ← 2 * x
2:    b ← a / 3       renaming ➔         2:    b ← c / 3
3:    a ← 9 * y                          3:    a ← 9 * y
```

    - problem: line 3 (in variable *a*) is both antidependent on line 2 and output dependent on line 1
    - after renaming, both dependencies have been removed, but line 2 (in variable *c*) is still flow dependent on line 1

Dependence Analysis

- loop dependencies
  - statements (almost always w.r.t. array access and modification) within a loop body might form a dependence
  - problem: finding dependencies throughout different iterations
  - prototype of a “normalised” nested loop with *N* levels

```
for i1 ← 1 to n1 do                                   // loop #1
    for i2 ← 1 to n2 do                               // loop #2

        for iN ← 1 to nN do                           // loop #N
            … ← …                                     // statements
```

  - *nesting level K* $(1 \leq K \leq N)$: number of surrounding loops $+$ 1
  - *iteration number* I_K: value of iteration variable at nesting level *K*

Dependence Analysis

- loop dependencies (cont'd)
  - *iteration vector I*: vector of integers containing the iteration numbers I_K of a particular iteration for each of the loops in order of the nesting levels

    $I = (I_1, I_2, \ldots, I_N)^T$ with iteration numbers I_K, $1 \leq K \leq N$

  - *iteration space*: set of all possible iteration vectors (for a statement)
  - *precedence $I < J$*: iteration I precedes iteration J iff

    $\exists K$: $I_R = J_R$, $\forall R$: $1 \leq R < K$, and $I_K < J_K$

  - *statement $S(I)$*: statement S under iteration vector I

Dependence Analysis

- loop dependencies (cont'd)
  - a statement $S_2(J)$ is loop dependent on $S_1(I)$ iff

    1) $I < J$ or
       $I = J$ and there exists a path from S_1 to S_2 in the loop body
    2) a memory location is accessed by S_1 on iteration I and by S_2 on iteration J
    3) one of these accesses is a write

  - theorem of loop dependence

There exists a dependence graph from statement S_1 to statement S_2 in a common nested loop if and only if there exist two iteration vectors I and J (for the nested loop), such that $S_2(J)$ is loop dependent on $S_1(I)$.

Dependence Analysis

- loop dependencies (cont'd)
  - *distance vector* $D(I,J)$: if statement $S_2(J)$ is loop dependent on $S_1(I)$ then the dependence distance vector is computed as follows

    $$D(I,J)_K = J_K - I_K\,,\ 1 \leq K \leq N$$

  - *direction vector* $R(I,J)$: if statement $S_2(J)$ is loop dependent on $S_1(I)$ then the dependence direction vector is computed as follows

    $$R(I,J)_K = \begin{cases} \text{“}<\text{”} & \text{if } D(I,J)_K > 0 \\ \text{“}=\text{”} & \text{if } D(I,J)_K = 0 \\ \text{“}>\text{”} & \text{if } D(I,J)_K < 0 \end{cases} \quad ,\ 1 \leq K \leq N$$

Dependence Analysis

- loop dependencies (cont'd)
  - types of different loop dependencies
    - *loop-carried dependence*
      - dependence from statement S_1 in iteration I to statement S_2 in iteration J iff $R(I,J)$ contains a "$<$" as its leftmost component which is not equal to "$=$"
      - level of a loop-carried dependence conforms to the index of the leftmost component of $R(I,J)$ that is not equal to "$=$"
    - *loop-independent dependence*
      - dependence from statement S_1 in iteration I to statement S_2 in iteration J iff $I = J$

Dependence Analysis

- loop dependencies (cont'd)
  - example (1)

```
        for i ← 1 to N do
            for j ← 1 to M do
1:              a(i,j) ← b(i,j)
2:              c(i,j) ← 2*c(i,j) + a(i−1,j)
            od
        od
```

  - again, loop dependence iff
    - 1) $I \leq J$
    - 2) $S_1(I)$ and $S_2(J)$ access the same resource
    - 3) one of these accesses is a write

Dependence Analysis

- loop dependencies (cont'd)
  - example (2)
    - flow dependence (RAW) in variable *a*

```
1:     a(i,j) ← …
2:            … ← … + a(i−1,j)
```

      - $D(I,J) = (1, 0)^T$ and $R(I,J) = (“<”, “=”)^T$
      - hence, a loop-carried dependence of level 1
    - antidependence (WAR) in variable *c*

```
2:            … ← 2*c(i,j) + …
2:     c(i,j) ← …
```

      - $D(I,J) = (0, 0)^T$ and $R(I,J) = (“=”, “=”)^T$
      - hence, a loop-independent dependence

Overview

- cache coherence
- memory consistency
- dependence analysis
- 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

- compiler directives (cont'd)
  - parallel region construct (1)

```
#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 numbered from 0 (master thread) to *N*–1 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

Programming with OpenMP

- compiler directives (cont'd)
  - parallel region construct (2)
    - 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

- compiler directives (cont'd)
  - parallel region construct (3)
    - 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

- compiler directives (cont'd)
  - 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)
      - *single*: serialises a section of code
    - must be encountered by all members of a team or none at all

Programming with OpenMP

- compiler directives (cont’d)
  - work-sharing constructs: for (1)

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

- iterations of loop immediately following “for” directive to be executed in parallel (only if parallel region has 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

- compiler directives (cont'd)
  - work-sharing constructs: for (2)
    - clause *schedule (type [,chunk]):* describes how iterations of the loop are divided among the threads; the default schedule is implementation dependent
      - *static*: iterations are divided into pieces of size *chunk* and statically assigned to threads (if chunk is omitted, the iterations are evenly distributed)
      - *dynamic*: when a thread finishes one chunk, it is dynamically assigned another (default chunk size is 1)
      - *runtime*: the scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE

Programming with OpenMP

- compiler directives (cont'd)
  - work-sharing constructs: for (3)
    - example

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

    …

    #pragma omp parallel shared (a, b, c) private (i)
        {
        #pragma omp for schedule (dynamic, 10) nowait
            for (i = 0; i < N; ++i)
                c[i] = a[i] + b[i];
        }
    …
}
```

Programming with OpenMP

- compiler directives (cont'd)
  - work-sharing constructs: sections (1)

```
#pragma omp sections [clause, …]  newline
     {
     #pragma omp section  newline
          structured_block

     #pragma omp section  newline
          structured_block
     }
```

  - independent "section" directives are nested within a "sections" directive; each section is executed once by a thread, different sections may be executed by different threads
  - there is an implied barrier at the end of a "sections" directive
  - to branch into or out of section blocks is illegal

Programming with OpenMP

- compiler directives (cont'd)
  - work-sharing constructs: sections (2)
    - example

```
int i;
float a[N], b[N], c[N];

…

#pragma omp parallel shared (a, b, c) private (i)
    {
    #pragma omp sections nowait
        {
        #pragma omp section
            for (i = 0; i < N/2; ++i) c[i] = a[i] + b[i];

        #pragma omp section
            for (i = N/2; i < N; ++i) c[i] = a[i] + b[i];
        }
    }
```

Programming with OpenMP

- compiler directives (cont'd)
  - work-sharing constructs: single

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

  - the enclosed code block is to be executed by only one thread (the thread that reaches the code block first)
  - threads that do not execute the "single" directive wait at the end of the enclosed code block
  - might be useful when dealing with sections of code that are not thread safe (such as I/O)
  - to branch into or out of a single block is illegal

Programming with OpenMP

- compiler directives (cont'd)
  - combined parallel work-sharing constructs

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

- iterations will be distributed in equal sized blocks (i.e. schedule static) to each thread

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

- specifies a parallel region containing a single "sections" directive

Programming with OpenMP

- compiler directives (cont'd)
  - 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

- compiler directives (cont'd)
  - synchronisation constructs (cont'd)

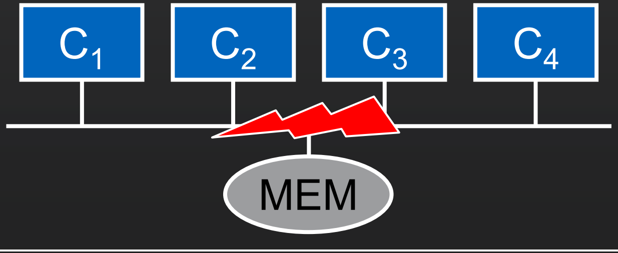
```
#pragma omp barrier  newline
```

- 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

- compiler directives (cont'd)
  - synchronisation constructs (cont'd)

```
#pragma omp flush (list)  newline
```

- identifies a synchronisation point at which the implementation must provide a consistent view of memory, i.e. thread-visible variables are written back to memory at this point
- optional list contains variables that will be flushed in order to avoid flushing all variables

```
#pragma omp ordered  newline
```

- specifies that iterations of the enclosed loop will be executed in same order as if they were executed on a monoprocessor
- further details see http://www.openmp.org

Programming with OpenMP

- data scope attribute clauses

  ```
  #pragma omp threadprivate (list)
  ```

  - used to make global file scope variables local and persistent to a thread through the execution of multiple parallel regions
  - directive must appear after declaration of listed variables
  - each thread gets its own copies of variables, hence data written by one thread is not visible to other threads
  - threadprivate variables differ from private variables because they are able to persist between different parallel sections of a code
  - on first entry to a parallel region, data in threadprivate variables should be assumed undefined, unless a “copyin” clause is specified in the “parallel” directive

Programming with OpenMP

- data scope attribute clauses (cont’d)
  - clauses (1)
    - *if (condition)*
    - *private (list)*, *firstprivate (list)*, *lastprivate (list)*, *shared (list)*
    - *default (shared | none)*
    - *nowait*
    - *schedule (static | dynamic | runtime [, chunk])*
    - *ordered*
    - *copyin (list)*
      - provides a means for assigning the same value to listed threadprivate variables for all threads
      - master thread variable is used as source, team threads are initialised with its value upon entry into the parallel construct

Programming with OpenMP

- data scope attribute clauses (cont'd)
  - clauses (2)
    - *reduction (operator: list)*
      - performs a reduction on the listed variables, i.e. several values are reduced to a single scalar value combined via the named operation *operator* (sum, product, e.g.)
      - listed variables must be of scalar type (no arrays and structs) and be declared shared in the enclosing context
      - the final result is written to the global shared variable
      - operator can be one of the following types
        - numerical: "+", "–", "*", "/"
        - logical: AND ("&&"), OR ("||")
        - bitwise: AND ("&"), OR ("|"), XOR ("^")

Programming with OpenMP

- data scope attribute clauses (cont'd)
  - example

```
main () {
    int i;
    int a[MAX], b[MAX];
    int res = 0;

    …

    #pragma omp parallel default (shared) private (i)
        {
        #pragma omp for reduction (+: res) nowait
            for (i = 0; i < MAX; ++i)
                res = res + a[i]*b[i];
        }

    printf (“result = %d\n”, res);
}
```

Programming with OpenMP

- data scope attribute clauses (cont'd)

clause	directive					
	parallel	for	sections	single	parallel for	parallel sections
if	✓				✓	✓
private	✓	✓	✓	✓	✓	✓
shared	✓	✓			✓	✓
default	✓				✓	✓
firstprivate	✓	✓	✓	✓	✓	✓
lastprivate		✓	✓		✓	✓
reduction	✓	✓	✓		✓	✓
copyin	✓				✓	✓
schedule		✓			✓	
ordered		✓			✓	
nowait		✓	✓	✓		

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