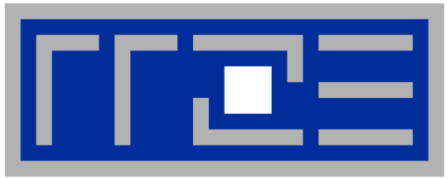
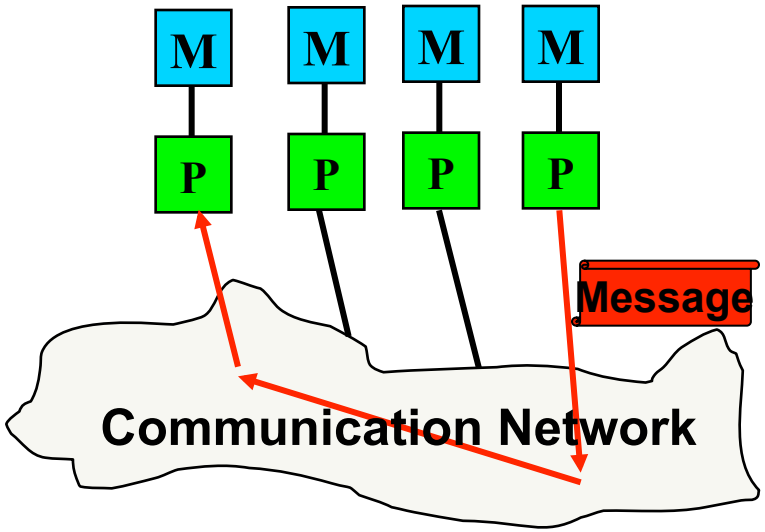


Elementary Parallel Programming

G. Hager (RRZE)
R. Bader (LRZ)

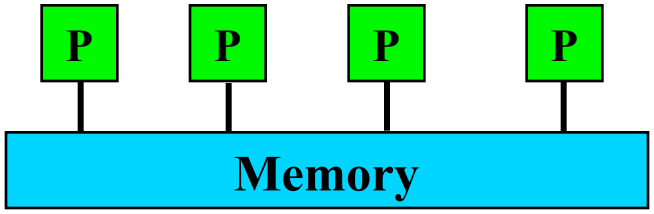
Two Paradigms for Parallel Programming
Hardware Designs

- **Distributed Memory**
  - message passing
  - explicit programming required



- **Special design:**
  - cache coherency protocol over interconnect
  - behaves like non-uniform shared memory

- **Shared Memory**
  - common address space for a number of CPUs
  - access efficiency may vary → SMP, (cc)NUMA
  - many programming models
  - potentially easier to handle
  - hardware and OS support required



Message Passing vs. Shared Memory: Programming Models

Distributed Memory

Same program on each processor/ machine (SPMD) or

Multiple programs with consistent communication structure (MPMD)

- **Program written in a sequential language**
  - all variables process-local
  - no implicit knowledge of data on other processors
- **Data exchange between processes:**
  - send/receive messages via appropriate library
  - most tedious, but also the most flexible way of parallelization
- **Parallel library discussed here:**
  - Message Passing Interface, **MPI**

Shared Memory

- **Single Program on single machine**
  - UNIX Process splits off **threads**, mapped to CPUs for work distribution
- **Data**
  - may be process-global or thread-local
  - exchange of data not needed, or via suitable synchronization mechanisms
- **Programming models**
  - explicit threading (hard)
  - directive-based threading via **OpenMP** (easier)
  - automatic parallelization (very easy, but mostly not efficient)

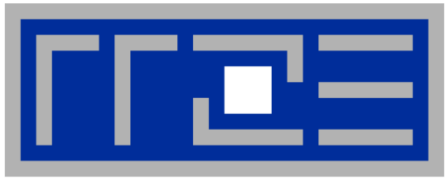
Standards-based Parallelism

- **MPI standard**
  - MPI forum released version 2.2 in September 2009
  - unified document („MPI1+2“)
- **Base languages**
  - Fortran (77, 95)
  - C
  - C++ binding obsolescent
    → use C bindings
- **Resources:**
  - http://www.mpi-forum.org

- **OpenMP standard**
  - architecture review board released version 3.1 in July 2011
  - feature update (tasking etc.)
- **Base languages**
  - Fortran (77, 95)
  - C, C++

  (Java is not a base language)
- **Resources:**
  - http://www.openmp.org
  - http://www.compunity.org

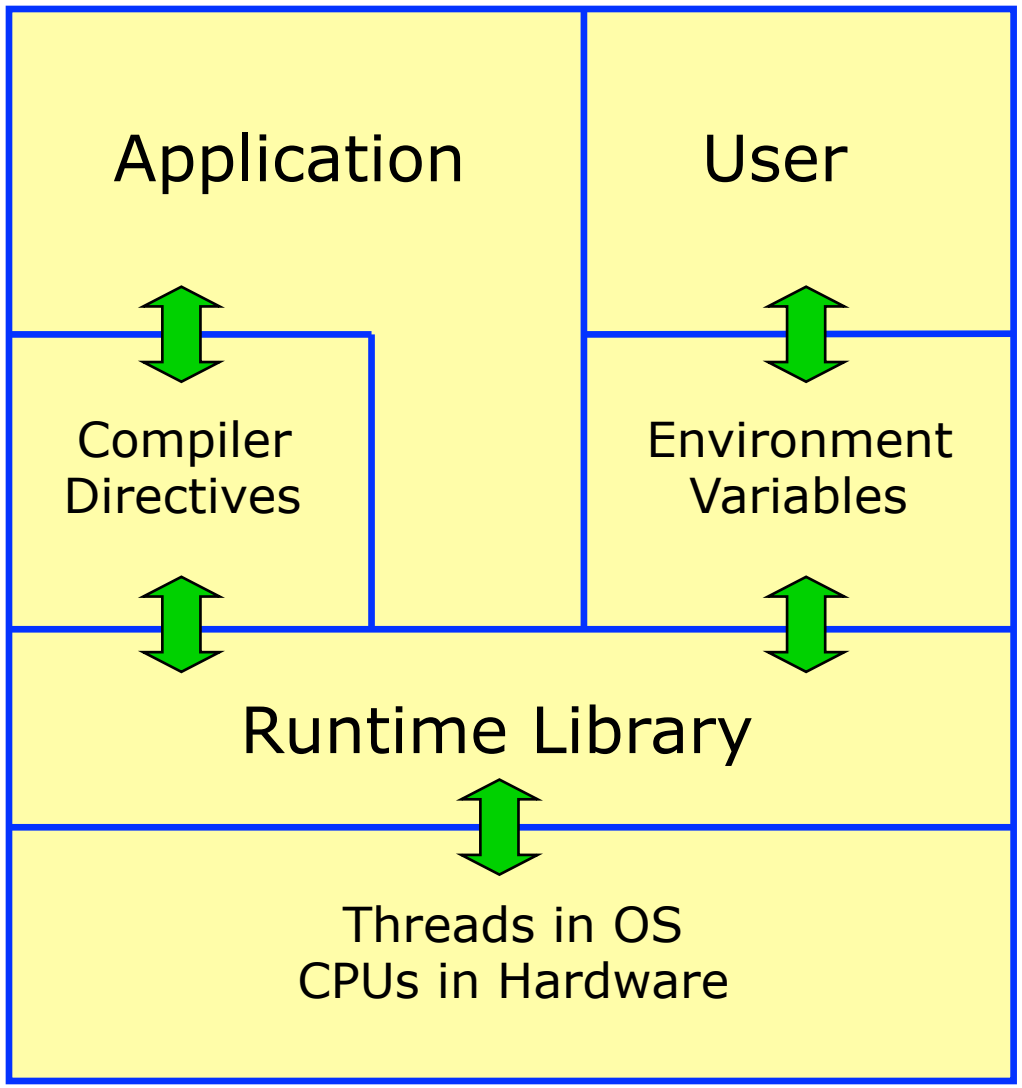
Portability:
- of semantics (well-defined, „safe“ interfaces)
- of performance (a difficult target)



OpenMP

Principles of
directive driven
shared memory parallelism

OpenMP Architecture



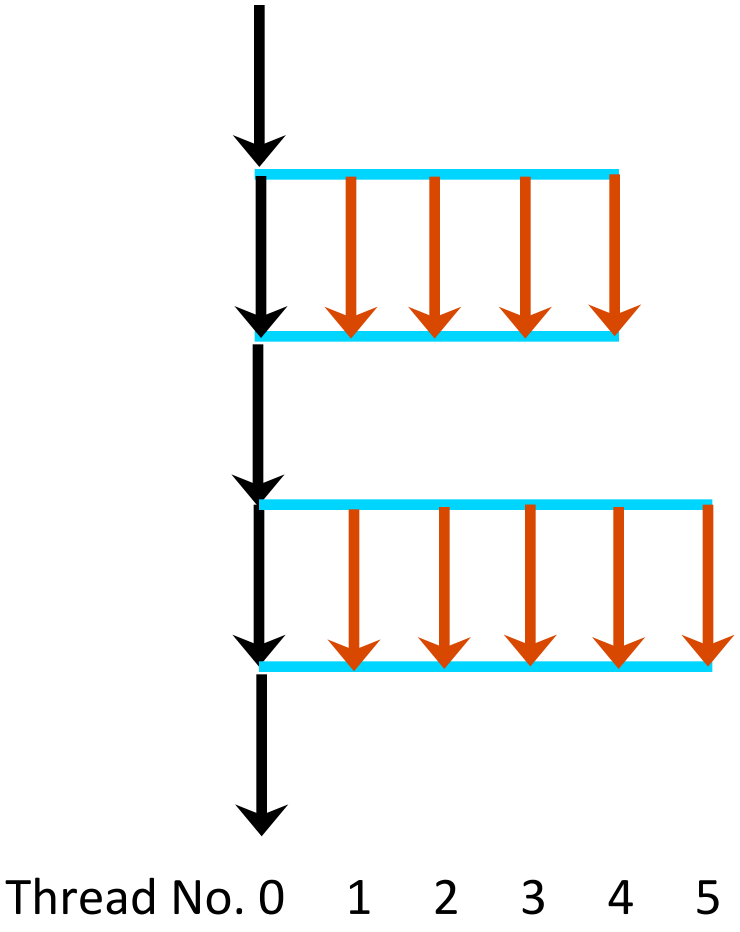
- **Operating system view:**
  - parallel work done by **threads**
- **Programmer's view:**
  - **directives**: comment lines in code
  - library routines
- **User's view:**
  - **environment variables** determine:

    resource allocation

    scheduling strategies

    and other (implementation-dependent) behaviour

OpenMP: Parallel execution



- **Program start:** only **master thread** runs
- **Parallel region:** team of worker threads is generated (“fork”)
- Threads **synchronize** when leaving parallel region (“join”)
- Only master executes sequential part (worker threads persist, but are inactive)
- **Task** and **data** **distribution possible via directives**
- **Nesting of parallel regions:**
  - allowed, but level of support implementation dependent
- Usually optimal:

  one thread per processor core

  other resource mappings are allowed/ possible

Parallel region:
Simplest program example

```
program hello
  use omp_lib
  implicit none
  integer :: nthr, myth

!$omp parallel private(myth)

!$omp single
  nthr = omp_get_num_threads()
!$omp end single

  myth = omp_get_thread_num()

  write(*,*) `Hello from `,myth, &
       & `of `, nthr

!$omp end parallel

end program hello
```

- **Parallel region directive:**
  - enclosed code executed by **all** threads („lexical construct“)
  - may include subprogram calls („dynamic region“)
- **Special function calls:**
  - module **omp_lib** provides interface
  - here: get number of threads and index of executing thread
- **Data scoping:**
  - uses a **clause** on the directive
  - **myth** thread-local: **private**
  - **nthr** process-global: **shared**

(will be discussed in more detail later)

Compiling and running an OpenMP program

Compile:

```
f90 -openmp -o hello.exe hello.f90
```

Run:

```
export OMP_NUM_THREADS=4
./hello.exe
 Hello from  0 of  4
 Hello from  2 of  4
 Hello from  3 of  4
 Hello from  1 of  4
```

ordering not reproducible

Compile for serial run:

```
f90 -o hello.exe hello.f90
```

- may require special switch for „stub library“ and module file

- **Special compiler switch**
  - activates OpenMP directives
  - generates threaded code
  - further suboptions may be available
  - each compiler has something **different** here
- **OpenMP environment**
  - defines runtime behaviour
  - here: number of threads used
- **Serial functionality of program**
  - (dis)order of output

OpenMP Fortran Syntax

- **Specifications:**
  - Fortran 77 style

```
include "omp_lib.h"
```

  - Fortran 90 module (**preferred**)

```
use omp_lib
```

- **Directives:**
  - fixed form source:

```
C$OMP <directive> [<clause [(<args>)]>, …]
```

    starting in column 1, also: `*$OMP`
  - free form source (preferred):

```
!$OMP <directive> [<clause [(<args>)]>, …]
```

- **Conditional compilation:**

```
   myid = 0
!$ myid = omp_get_thread_num()
```

  - beware implicit typing!
- **Continuation line:**

```
!$OMP <directive> &
!$OMP <clause>
```

OpenMP C/C++ Syntax

- **Include file:**

```
#include <omp.h>
```

- **Preprocessor directive: uses pragma feature**

```
#pragma omp <directive> [clause …]
```

- **Conditional compilation: OpenMP switch sets preprocessor macro**

```
#ifdef _OPENMP

… /* do something */

#endif
```

- **Continuation line:**

```
#pragma omp directive \
   clause
```

OpenMP Syntax: Remarks on clauses

- **Many (but not all) OpenMP directives support clauses**
  - more than one may appear on a given directive
- **Clauses specify additional information associated with the directive**
  - modification of directive's semantics
- **"Simplest example" from above:**
  - `private(…)` appears as clause to the **`parallel`** directive
- **The specific clause(s) that can be used depend on the directive**

OpenMP Syntax: Structured block

- **Defined by braces in C/C++**
- **If explicitly specified in Fortran:**
  - code between begin/end of an OpenMP construct must be a complete, valid Fortran block
- **Single point of entry:**
  - no **GOTO** into block (Fortran), no **setjmp()** to entry point (C)
- **Single point of exit:**
  - **RETURN, GOTO, EXIT** outside block are prohibited (Fortran)
  - **longjmp()** and **throw()** must not violate entry/exit rules (C, C++)
  - **exception:** termination via **STOP** or **exit()**

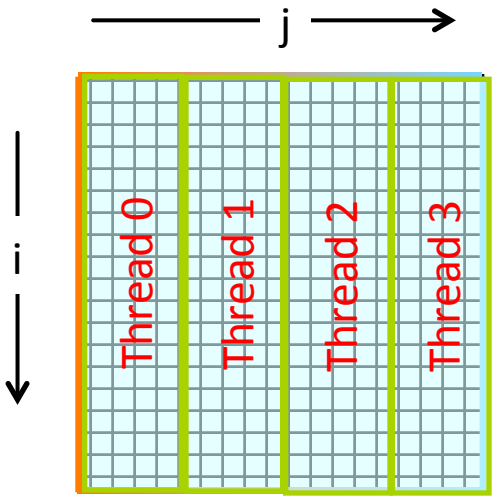
- **Block structure example:**
  - C version of simplest program

```
#include <omp.h>

int main() {
  int numth = 1;
#pragma omp parallel
  {
      int myth = 0; /* private */
#ifdef _OPENMP
#pragma omp single
      numth = omp_get_num_threads();
      /* block above: one statement */
      myth = omp_get_thread_num();
#endif
      printf("Hello from %i of %i\n",\
             myth,numth);
  }  /* end parallel */
}
```

Work sharing in OpenMP (1)

- **Making parallel regions useful …**
  - divide up work between threads
- **Example:**
  - working on an array processed by a nested loop structure



- iteration space of **directly nested loop** is sliced

```
real :: a(ndim, ndim)
…
!$omp parallel
!$omp do
do j=1, ndim
  do i=1, ndim
    …
    a(i, j) = …
  end do
end do
!$omp end do
  …
!$omp end parallel
```

j-loop is sliced

synchronization between threads

further parallel execution

Work sharing in OpenMP (2)

- **Synchronization behaviour:**
  - all threads (by default) **wait for completion** at the end of the work sharing region („barrier“)
  - following references and definitions to an array element by **other** threads are therefore OK.
- **Slicing of iteration space:**
  - „loop scheduling“
  - default behaviour is implementation dependent
  - usually as equal as possible chunks of largest possible size
- **Additional clauses on `!$OMP DO`**
  - will be discussed in another talk

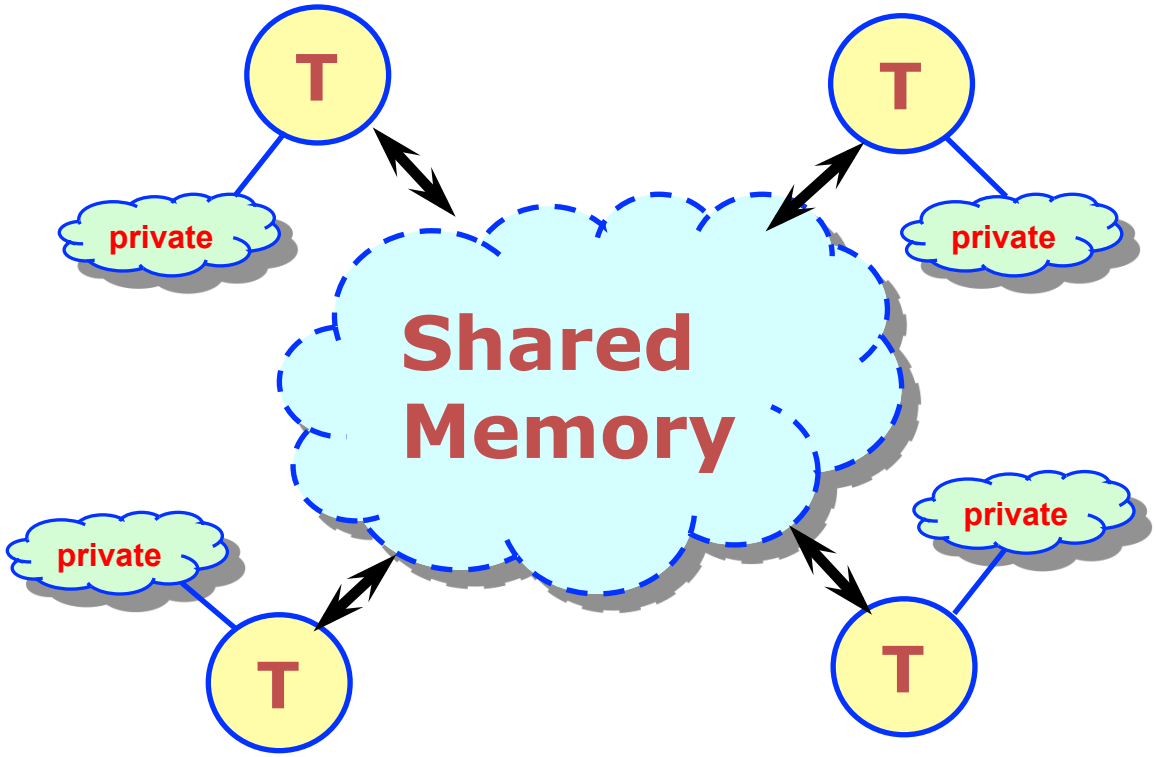
- **C/C++ syntax:**

```
#pragma omp for [clause]
for ( ... )   {
        ...    // loop body
    }
```

- **Restrictions on loop structure:**
  - trip count must be **computable** at entry to loop

    disallowed: Fortran **`do while`**, **`exit`**; C style loops modifying the loop variable, or otherwise violating the require-ment
  - loop body with single entry and single exit point

Two kinds of memory exist in OpenMP



- **Threads access globally shared memory**
- **Data can be shared or private**
  - shared data – one instance of an entity available to all threads (in principle)
  - private data – each per-thread copy only available to thread that owns it
- **Data transfer** transparent to programmer
- **Synchronization** takes place (is mostly implicit)

Defaults for data scoping

- **All variables defined or referenced in a parallel region are shared**
  - including global variables or variables in a COMMON block
- **Exceptions:**
  1. local variables of subroutines invoked (or blocks entered) inside a parallel region (unless they have the **SAVE** attribute)
  2. loop variables of workshared loops and loops nested inside a workshared loop

  which are (and always should be) **private**
  - exceptions immutable if default is changed

- **Notes:**

```
export OMP_STACKSIZE=100M
```

  - many local variables/automatic arrays → default setting for thread specific stack size may be too small
  - specifying **SAVE** and use of global variables:

    usually **not** a good idea

    code often not thread-safe
- **Changing defaults:**
  - **default** clause (for some directives)

```
!$OMP <directive> default(private)
!$OMP <directive> default(none)
!$OMP <directive> default(shared)
```

  - enforce explicit scoping for „none“ (**recommended practice**)
  - **default(private)** **only** in Fortran

Scoping: second-simplest example

- **Summation inside a loop**

```
real :: s, stot
stot = 0.0
!$omp parallel private(s)
s = 0.0
!$omp do
do i=1, ndim
    … ! workload ???
    s = s + …
end do
!$omp end do
!$omp critical
  stot = stot + s
!$omp end critical
!$omp end parallel
```

Note: large workload inside loop improves threaded performance

- require thread-individual variable for partial sum calculated on each thread
- **but:** private copies of variables are **undefined** at entry to, and become **undefined** at exit of the parallel region
- **therefore:** collect partial sums to a **shared** variable defined after the worksharing region
- **updates** to shared variable must be specially protected:

→ use a **critical region**

→ only one thread at a time may execute (mutual exclusion)

(performance impact due to explicit synchronization)

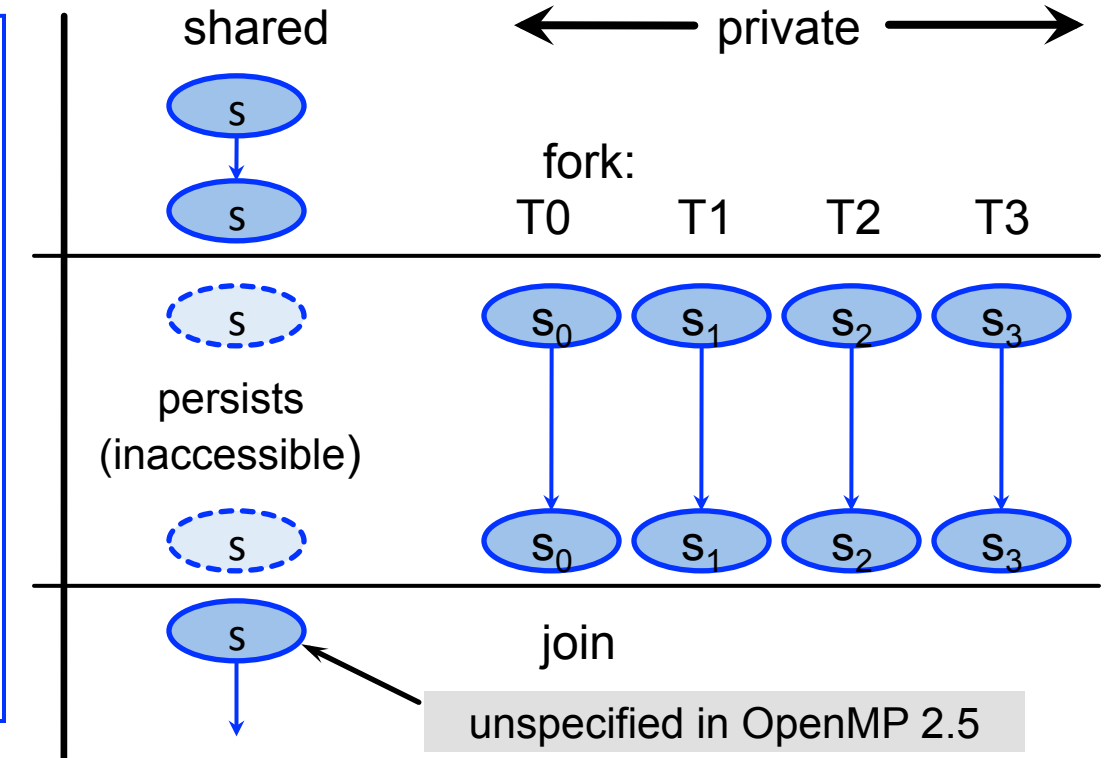
Private variables - Masking

```
real :: s

  s = …
!$omp parallel private(s)

  s = …
  … = … + s


!$omp end parallel
  … = … + s
```



- **Masking relevant for**
  - privatized variables defined in scope outside the parallel region

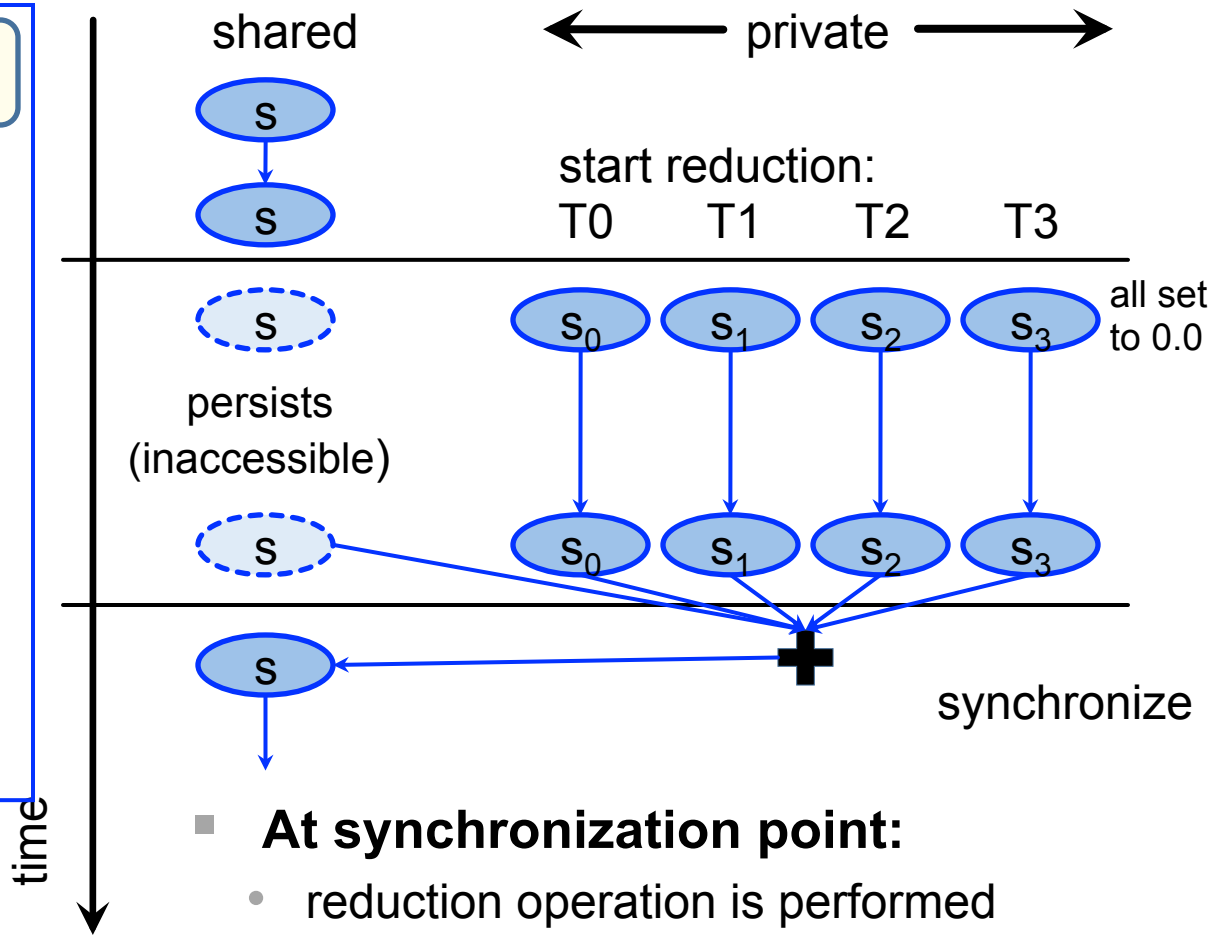
- **Masking also applies to**
  - association status of pointers
  - allocation status of allocatable variables

Reduction operations (1)

```
real :: s

!$omp parallel
!$omp do reduction(+:s)
  do i = …
       :
       :
     s = s + …
  end do
!$omp end do
  … = … * s
!$omp end parallel
```

s is still shared here



Note: this improves on the summation example
(no explicit critical region needed)

- **At synchronization point:**
  - reduction operation is performed
  - result is transferred to master copy
  - restrictions similar to **`firstprivate`**

Reduction operations (2)

- **Initial value of reduction variable**
  - depends on operation

Operation	Initial value
+	0
-	0
*	1
.and.	.true.
.or.	.false.
.eqv.	.true.
.neqv.	.false.
MAX	-HUGE(X)
MIN	HUGE(X)
IAND	all bits set
IEOR	0
IOR	0

C / C++ has an analogous set

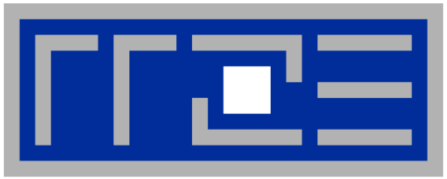
- **Consistency required**
  - operation specified in clause vs. update statement
  - rely on algebraic rules!
  - subtract: `x = expr - x` is **not** allowed
- **Multiple reductions:**
  - multiple scalars, or an array:

```
real :: x, y, z
!$OMP do reduction(+:x, y, z)
```

```
real :: a(n)
!$OMP do reduction(*:a)
```

  - multiple operations:

```
!$OMP do reduction(+:x, y) &
!$OMP    reduction(*:z)
```



Work Sharing Schemes

Loops and loop scheduling

Collapsing loop nests

Parallel sections

Array processing

The `schedule` clause

- **Default scheduling:**
  - implementation dependent
  - **typical:** largest possible chunks of as-equal-as-possible size („static scheduling“)

iteration space
(threads color coded)

- **User-defined scheduling:**

```
                        static
!$OMP do schedule(      dynamic  [,chunk] )
                        guided
```

`chunk`: always a non-negative integer. If omitted, has a schedule dependent default value

- **1. Static scheduling**
  - `schedule(static,10)`

10 iterations

  - minimal overhead (precalculate work assignment)
  - default chunk value: see left
- **2. Dynamic scheduling**
  - after a thread has completed a chunk, it is assigned a new one, until no chunks are left

`schedule(dynamic, 10)`

both threads take long to complete their chunk (workload imbalance)

  - synchronization **overhead**
  - default chunk value is **1**

Collapsing loop nests

- **Flatten nested loops into a single iteration space**

```
!$OMP do collapse(2)
  do k=1, kmax
    do j=1, jmax
      :
    end do
  end do
!$OMP end do
```

collapse clause (since 3.0): argument specifies number of loop nests to flatten

- **Restrictions:**
  - iteration space computable at entry to loop (rectangular)
  - `CYCLE` (Fortran) or `continue` (C) may only appear in innermost loop

- **Logical iteration space**
  - example: kmax=3, jmax=3

	0	1	2	3	4	5	6	7	8
J	1	2	3	1	2	3	1	2	3
K	1	1	1	2	2	2	3	3	3

  - this is what is divided up into chunks and distributed among threads
  - Sequential execution of the iterations in all loops determines the order of iterations in the collapsed iteration space
- **Optimization effect**
  - may improve memory locality properties
  - may reduce data traffic between cores

Performance Tuning: the `nowait` clause

- **Remember:**
  - an OpenMP for/do performs implicit synchronization at loop completion
- **Example: multiple loops in parallel region**

```
!$omp parallel
!$omp do
  do k=1, kmax_1
    a(k) = a(k) + b(k)
  end do
!$omp end do nowait
  : ! code not involving
  : ! reads of a, writes to b
!$omp do
  do k=1, kmax_2
    c(k) = c(k) * d(k)
  end do
!$omp end do
!$omp end parallel
```

do not synchronize

Implicit barrier

- **C syntax**
  - specify on OpenMP pragma **before** code block:
  - #pragma omp for nowait
- **Shooting yourself in the foot**
  - modified variables must not be accessed unless **explicit** synchronization is performed