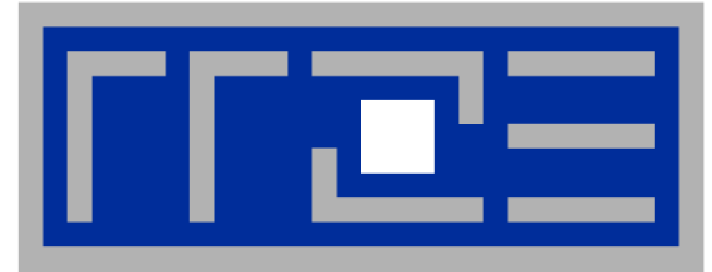


Shared-memory parallel processing with OpenMP

Introduction to OpenMP: Basics

Prof. Dr. G. Wellein[a,b] , Dr. G. Hager[a] , J. Habich[a]

[a]HPC Services – Regionales Rechenzentrum Erlangen
[b]Department für Informatik

University Erlangen-Nürnberg, Sommersemester 2011

Introduction to OpenMP: Basics

- **"Easy", incremental and portable parallel programming of shared-memory computers: OpenMP**
- **Standardized set of compiler directives & library functions: `http://www.openmp.org/`**
  - FORTRAN, C and C++ interfaces are defined
  - Supported by most/all commercial compilers, GNU starting with 4.2
  - Few free tools are available
- **OpenMP program can be written to compile and execute on a single-processor machine just by ignoring the directives**
  - API calls must be masked out though
  - Supports data parallelism

- R.Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon: Parallel programming in OpenMP. Academic Press, San Diego, USA, 2000, ISBN 1-55860-671-8
- B. Chapman, G. Jost, R. v. d. Pas: Using OpenMP.MIT Press, 2007, ISBN 978-0262533027

Introduction to OpenMP: Shared-Memory model

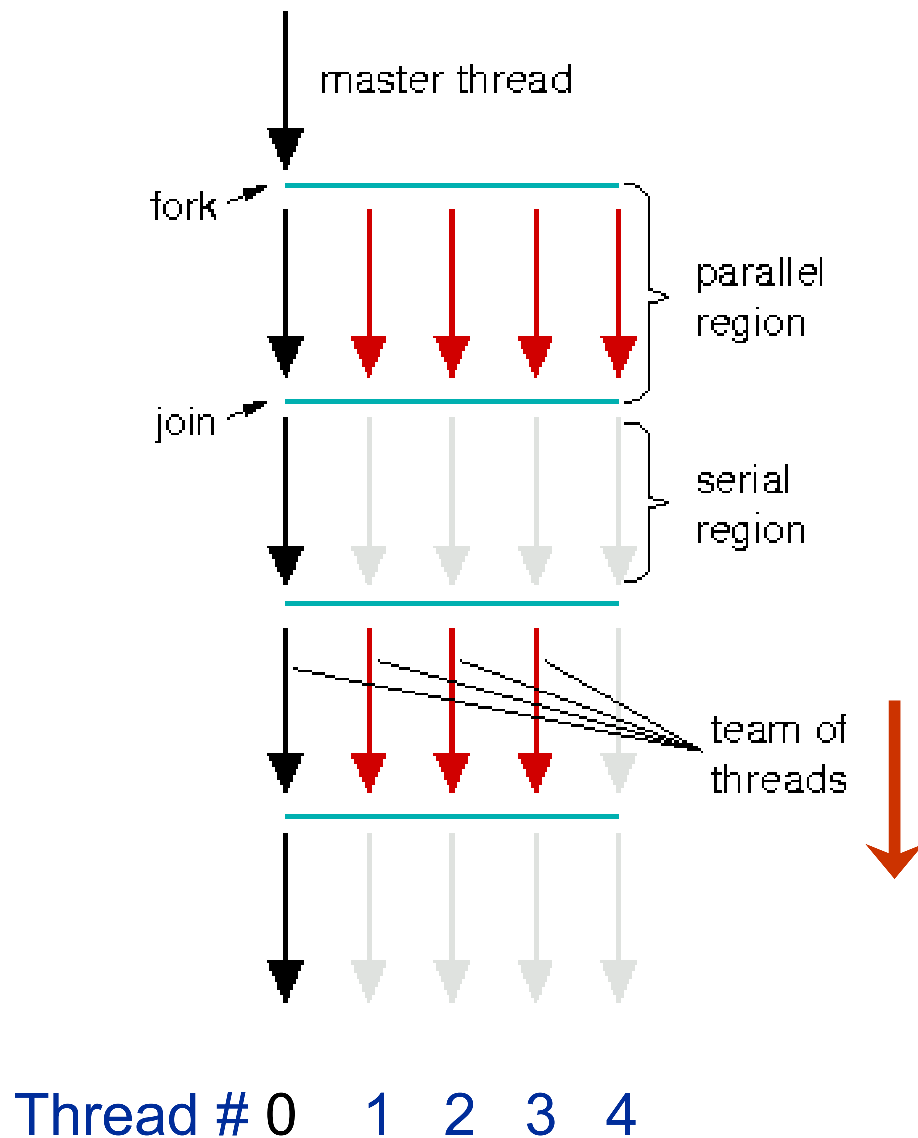
Central concept of OpenMP programming: Threads

T T T T
private private private private
Shared Memory

- Threads access globally shared memory
- Data: shared or private
  - shared data available to all threads (in principle)
  - private data only to thread that owns it
- Data transfer transparent to programmer
- Synchronization takes place, is mostly implicit
- Tailored to data parallel execution

Other threading libs. Available, e.g. pthreads

Introduction to OpenMP: Fork and join execution model



- **Program start: only master thread runs**
- **Parallel region: team of threads is generated (“fork”)**
- **Synchronize when leaving parallel region (“join”)**
- **Only master executes sequential part**
  - worker threads usually sleep
- **task and data distribution possible via directives**
- **Usually optimal:**
  1 thread per processor

- **Each directive starts with sentinel in column 1:**
  - fixed source: !$OMP or C$OMP or *$OMP
  - free source: !$OMP

  followed by a `directive` and, optionally, `clauses`.
- **If OpenMP is not enabled by compiler → redundant comment**

- **Access to OpenMP library calls:**
  - Use include file (omp_lib.h) for API call prototypes (or Fortran 90 module omp_lib if available)
  - Perform conditional compilation of lines starting with !$ or C$ or *$ to ensure compatibility with sequential execution
- **Example:**

```
   myid = 0
!$ myid = omp_get_thread_num()
   numthreads = 1
!$ numthreads = omp_get_num_threads()
```

- **Intel compiler: `-openmp`**

- Include file: `#include <omp.h>`

- Compiler directive:

```
#pragma omp [directive [clause ...]]
   structured block
```

- Conditional compilation: Compiler's OpenMP switch sets preprocessor macro (-D_OPENMP)

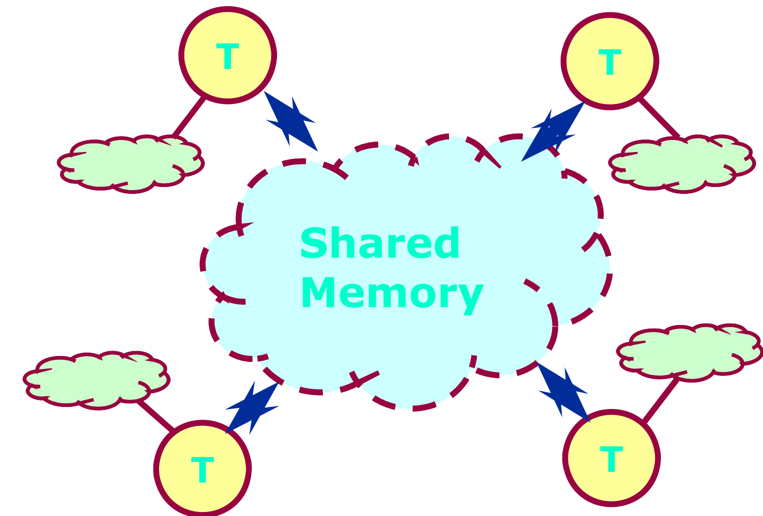
```
#ifdef _OPENMP

 ... do something

 #endif
```

- `#pragma omp parallel`
  `structured block`
  - Makes the structured block a parallel region: All code executed between start and end of this region is executed by all threads.
  - This includes subroutine calls within the region (unless explicitly sequentialized)
  - Local variables inside the block are automatically private to each thread

- **`END PARALLEL` directive is required in Fortran to define boundaries of parallel region**

```
   use omp_lib
   …
!$OMP PARALLEL
  call work(omp_get_thread_num(), omp_get_num_threads())
!$OMP END PARALLEL
```

Introduction to OpenMP: Data scoping – shared vs. private



- **Remember the OpenMP memory model?**

  Within a parallel region, data can either be
- **private to each executing thread**

  → each thread has its own local copy of data

  or be
- **shared between threads**

  → there is only one instance of data available to all threads

  → this does not mean that the instance is always visible to all threads!

- **OMP clause specifies scope of variables:**
  - Default: shared
  - Specify private variables in a parallel region:

```
#pragma omp parallel private(var1, tmp)
```

Introduction to OpenMP: Data scoping – shared vs. private

- **Default: All data in a parallel region is shared**
- **This includes global data (global/static variables, C++ class variables)**
- **Exceptions:**
  1. Local data within enclosed function calls are private
     (Note: Inlining must be treated correctly by compiler!) unless declared `static`
  2. Loop variables of parallel ("sliced") loops are private (cf. workshare constructs)

- **Due to stack size limits it may be necessary to make large arrays static**
  - This presupposes it is safe to do so!
  - If not: make data dynamically allocated
  - As of OpenMP 3.0: `OMP_STACKSIZE` may be set at run time (increase thread-specific stack size):

```
$ setenv OMP_STACKSIZE 100M
```

```
    use omp_lib
    integer myid, numthreads
    …
    myid=0; numthreads=1
!$OMP PARALLEL PRIVATE(myid,numthreads)
!$ myid = omp_get_thread_num()
!$ numthreads= omp_get_num_threads()
   call work(myid, numthreads)
!$OMP END PARALLEL
```

```
    include <omp.h>
 …
#pragma omp parallel{
    int myid=0, numthreads=1
#ifdef _OPENMP
!$ myid = omp_get_thread_num()
!$ numthreads= omp_get_num_threads()
#endif
    work(myid, numthreads) }
```

Local variables are private to each thread!

- **Beware side effects of data scoping:**

  Incorrect `shared` attribute may lead to race conditions and/or performance issues ("false sharing").
  - Use verification tools
  - See later

- **Scoping of local function data and global data**
  - is not (hereby) changed
  - compiler cannot be assumed to have knowledge

- **Recommendation: Use**

```
#pragma omp parallel default(none)
```

to not overlook anything – the compiler will then complain about every variable that has no explicit scoping attribute

- **What if initialization of privatized variables is required?**
  - **`FIRSTPRIVATE( var )`** clause for setting each private copy to the previous global value

- **What if value of last iteration is needed after the loop?**
  - **`LASTPRIVATE( var )`**

    `var` is updated by the thread that computes
    - **the sequentially last iteration (on `do` or `for` loops)**
    - **the last section**

- **What if a global (or COMMON) variable needs to be privatized?**
  - **`THREADPRIVATE`** / **`COPYIN`**
  - cf. standards documents

Introduction to OpenMP: Compile and run

- **Compiler must be instructed to recognize OpenMP directives (Intel compiler: `-openmp`)**
- **Number of threads: Determined by shell variable `OMP_NUM_THREADS`**

```
$ export OMP_NUM_THREADS=4
$ ./a.out
```

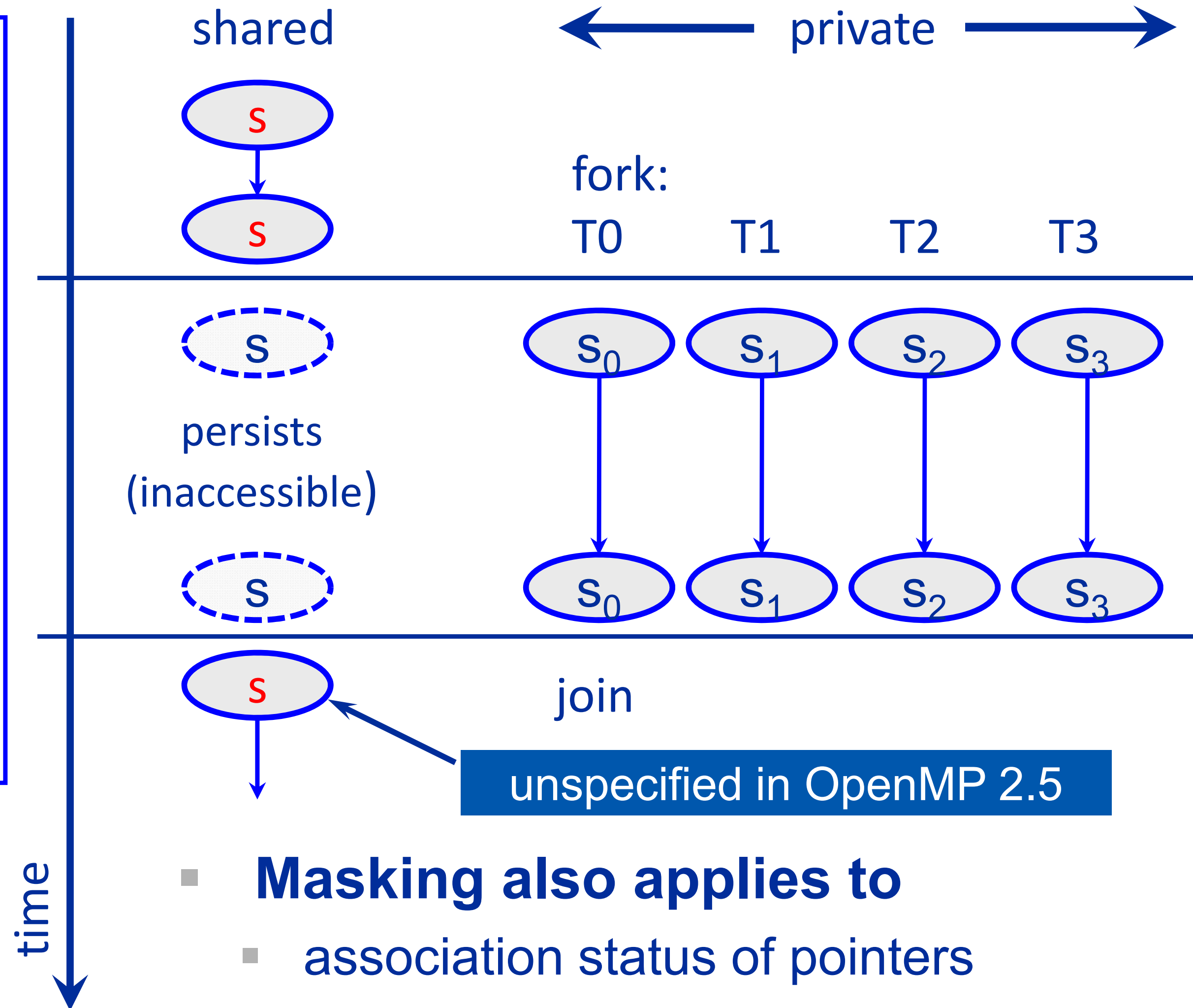
- **More environment variables available:**
  - Loop scheduling: Determined by shell variable `OMP_SCHEDULE`, if the "runtime" schedule type is specified in the source code
  - Stacksize: `OMP_STACKSIZE`
  - Dynamic adjustment of threads: `OMP_DYNAMIC`
- **Executable should be able to run with any number of threads!**
- **Thread pinning & core/thread affinity via LIKWID**

```
$ export OMP_NUM_THREADS=4
$ likwid-pin -c 0-3 -t intel ./a.out
```

Which threading lib. / compiler has been used?

Private variables - Masking

```
real :: s

  s = …
!$omp parallel private(s)

  s = …
  … = … + s


!$omp end parallel
  … = … + s
```



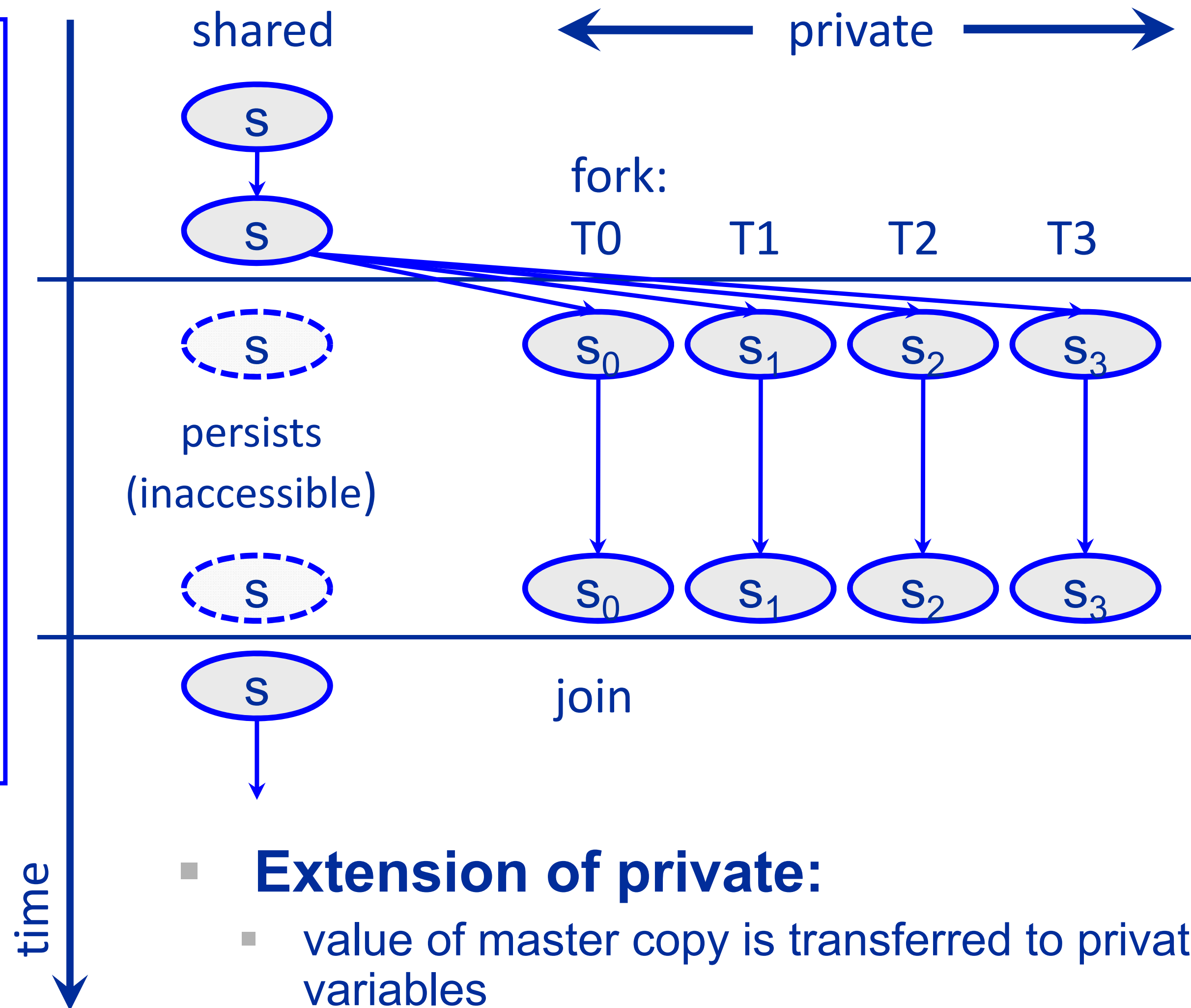
- **Masking relevant for**
  - privatized variables defined in scope outside the parallel region

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

The firstprivate clause

```
real :: s

  s = …
!$omp parallel &
!$omp firstprivate(s)

  … = … + s
  call foo()

!$omp end parallel
  … = … + s
```

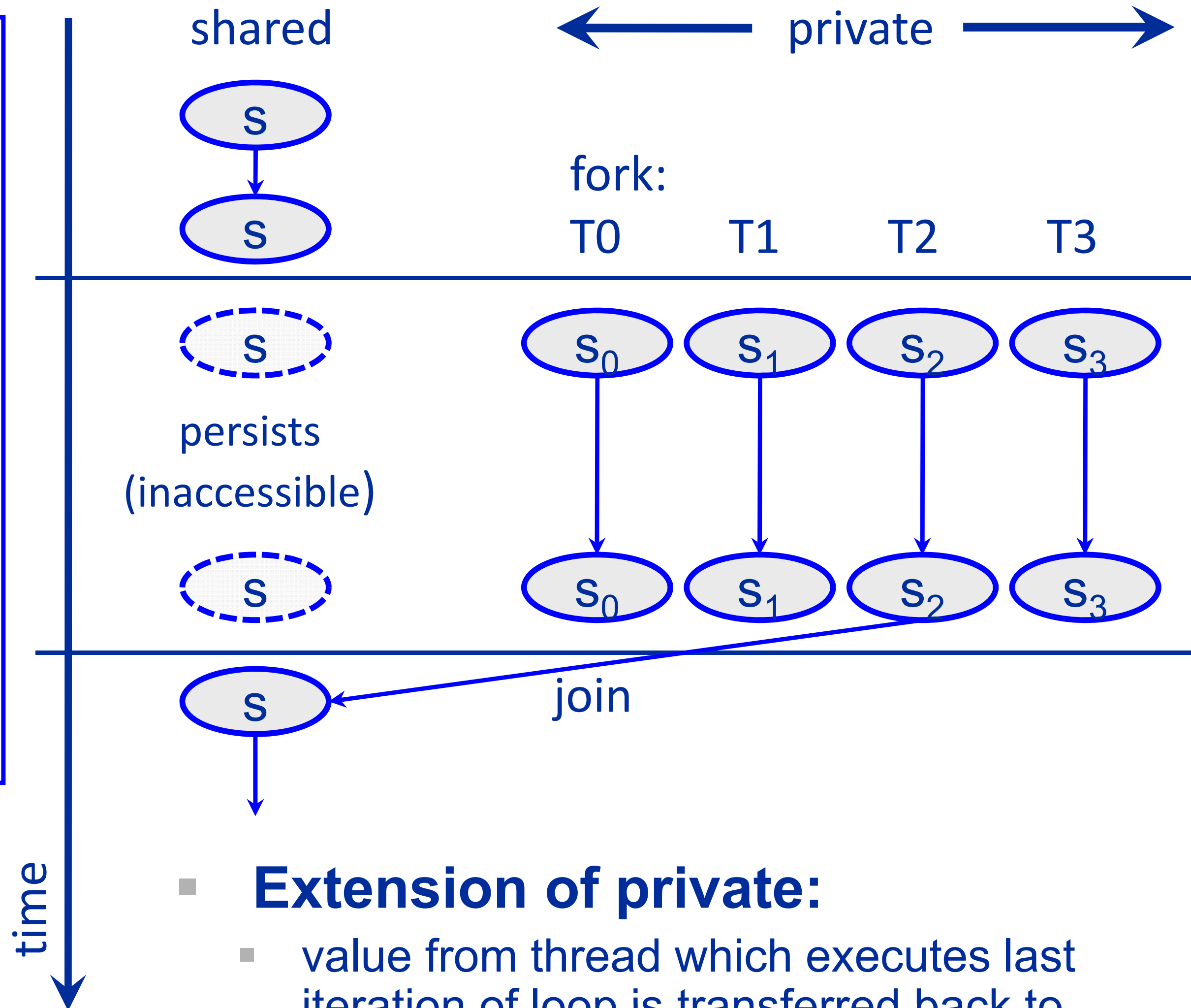
if foo() references or defines s (e.g. by host association), it may work on a copy of s.



- **Extension of private:**
  - value of master copy is transferred to private variables
  - **restrictions:** not a pointer, not assumed shape, not a subobject, master copy not itself private etc.

The lastprivate clause

```
real :: s

  s = …
!$omp parallel &
!$omp lastprivate(s)
!$omp do
  do i = …
     s = …
  end do
!$omp end do
!$omp end parallel
  … = … + s
```



- **Extension of private:**
  - value from thread which executes last iteration of loop is transferred back to master copy (which must be allocated if it is a dynamic entity)
  - restrictions similar to `firstprivate`

```
    use omp_lib
   integer tid, numth, i, bstart, bend, blen, N
   double precision, dimension(N):: a,b,c,d
   …
!$OMP PARALLEL PRIVATE(tid, numth, bstart, bend, blen, i)
   tid=0; numth=1
!$ tid   = omp_get_thread_num()
!$ numth = omp_get_num_threads()
   blen = N/numth
   if(tid.lt.mod(N,numth)) then
      blen=blen+1
      bstart=blen*tid+1
   else
      bstart=blen*tid+mod(N,numth)+1
   endif
   bend=bstart+blen-1
   do i=bstart,bend
      a(i)=b(i)+c(i)*d(i)
   enddo
!$OMP END PARALLEL
```

Not a low overhead solution…………….

Introduction to OpenMP: Workshare construct

- `!$OMP DO[clause]` declares the loop following to be divided up if within a parallel region ("sliced")
- Loop counter of parallel loop is declared private implicitly

```
    integer i, N
    double precision, dimension(N):: a,b,c,d
    …
!$OMP PARALLEL
!$OMP DO                ! Parallelize loop
   do i=1,N
       a(i)=b(i)+c(i)*d(i)
    enddo
!$OMP END DO
!$OMP END PARALLEL
```

- Implicit thread synchronization at END DO and END PARALLEL
- Suppress barrier at END DO: clause=NOWAIT

!$OMP PARALLEL DO[clause] **Combined workshare construct**

```
    integer i, N
    double precision, dimension(N):: a,b,c,d
    …
!$OMP PARALLEL DO ! Fork team of threads & parallelize
   do i=1,N
       a(i)=b(i)+c(i)*d(i)
    enddo
!$OMP END PARALLEL DO
```

- **Distribute the execution of the enclosed code region among the members of the team**
  - Must be enclosed dynamically within a parallel region
  - Threads do not (usually) launch new threads
  - No implied barrier on entry

- **Directives**
  - `do` directive (Fortran), `for` directive (C/C++)

  - `section(s)` directives (we ignore this)
  - `workshare` directive (Fortran 90 only)
  - Tasking constructs (advanced – available since OpenMP 3.0)

`#pragma omp for [clause]` & `!$OMP DO [clause]`

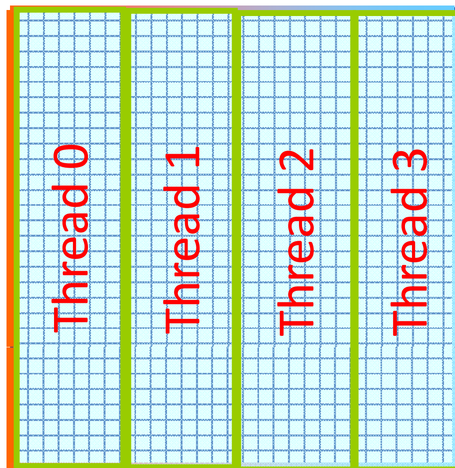
- **Only the loop immediately following the directive is workshared**
- **Restrictions on parallel loops (especially in C/C++)**
  - trip count must be computable (no `do while`)
  - loop body with single entry and single exit point
- **Standard random access iterator loops are supported as of OpenMP 3.0:**

```
#pragma omp for
for(vector::iterator i=v.begin(); i!=v.end(); ++i) {
  ... do stuff using *i etc ... }
```

- **Nested loops in Fortran: If outer looper is parallelized (via `!$OMP DO`) all inner loop counters are private automatically**

```
!$OMP PARALLEL DO
  do i=1,N
     do j=1,N
        a(i,j)=b(j,i)
     enddo
   enddo
!$OMP END PARALLEL DO
```

- **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   ! sliced
  do i=1, ndim
    …
    a(i, j) = …
  end do
end do
!$omp end do      synchronization
  …   ! further parallel stuff
!$omp end parallel
```

Introduction to OpenMP: Workshare constructs

- *clause* can be one of the following:
  - `private, firstprivate, lastprivate`
  - `reduction` (*operator*: *list*) [see later]
  - `schedule` (*type* [, *chunk*]) [see next slide]
  - `nowait` [see below]
  - `collapse(`*n*`)`
  - ... and a few others
- **Implicit barrier at the end of loop unless `nowait` is specified**
- **If `nowait` is specified, threads do not synchronize at the end of the parallel loop**
- **`collapse`: Fuse nested loops to a single (larger one) and parallelize it**
- **`schedule` clause specifies how iterations of the loop are distributed among the threads of the team.**
  - Default is implementation-dependent

Introduction to OpenMP: `schedule` clause

Within `schedule(` *type* [, *chunk*] `)` *type* can be one of the following:

- `static`: Iterations are divided into pieces of a size specified by *chunk*. The pieces are statically assigned to threads in the team in a round-robin fashion in the order of the thread number.
  Default chunk size: one contiguous piece for each thread.
- `dynamic`: Iterations are broken into pieces of a size specified by *chunk*. As each thread finishes a piece of the iteration space, it dynamically obtains the next set of iterations. *Default chunk size: 1.*
- `guided`: The chunk size is reduced in an exponentially decreasing manner with each dispatched piece of the iteration space.
  chunk specifies the smallest piece (except possibly the last).
  Default chunk size: 1. Initial chunk size is implementation dependent.
- `runtime`: The decision regarding scheduling is deferred until run time. The schedule type and chunk size can be chosen at run time by setting the `OMP_SCHEDULE` environment variable.

Default `schedule`: implementation dependent.

Iteration	STATIC	STATIC,3	DYNAMIC[,1]	DYNAMIC,3	GUIDED[,1]
1	T0	T0	T0	T0	T0
2	T0	T0	T1	T0	T0
3	T0	T0	T2	T0	T0
4	T0	T1	T0	T1	T0
5	T0	T1	T1	T1	T0
6	T0	T1	T0	T1	T0
7	T0	T2	T2	T2	T1
8	T1	T2	T2	T2	T1
9	T1	T2	T0	T2	T1
10	T1	T0	T1	T1	T2
11	T1	T0	T2	T1	T2
12	T1	T0	T1	T1	T2
13	T1	T1	T1	T2	T0
14	T1	T1	T0	T2	T0
15	T2	T1	T2	T2	T1
16	T2	T2	T2	T0	T1
17	T2	T2	T1	T0	T0
18	T2	T2	T2	T0	T1
19	T2	T0	T1	T2	T2
20	T2	T0	T2	T2	T0

T0 T1 T2

- Dense matrix-vector multiplication

```
#pragma omp parallel
{
  for(int j=0; j<niter; j++){

#pragma omp for schedule(...)
    for(int m=0; m<size; m++){
      for(int n=0; n<size; n++){
        y[m]+=a[m*size+n]*x[n];
      }
    }
    if(y[size>>1]<0){
      dummy(a,x,y,0);
    }

  }
}
```

Introduction to OpenMP: Dense MVM performance

P P P P
C C C C
C C C C
C
MI
Memory

P P P P
C C C C
C C C C
C
MI
Memory

MFlop/s
25000
20000
15000
10000
5000
0

serial
static 1T
static 2T
static 4T
static 8T
dynamic 4T
dynamic 8T
guided 4T
guided 8T

100
1000
N

Introduction to OpenMP: `schedule` clause

- **Dense triangular MVM**

```
#pragma omp parallel
{
  for(int j=0; j<niter; j++){

#pragma omp for schedule(...)
    for(int m=0; m<size; m++){
      for(int n=m; n<size; n++){
        y[m]+=a[m*size+n]*x[n];
      }
    }
    if(y[size>>1]<0){
      dummy(a,x,y,0);
    }

  }
}
```

Excercise

- **Dense MVM + scalar product**

```
#pragma omp parallel
{
  for(int j=0; j<niter; j++){

#pragma omp for schedule(...)
    for(int m=0; m<size; m++){
      for(int n=m; n<size; n++){
        y[m]+=a[m*size+n]*x[n];
      }
    }
    s=0;
    for(int m=0; m<size;m++)
      s+=x[m]*y[m];
  ...
  }
}
```

?

Reduction

Introduction to OpenMP: Reduction operations

```
real :: s

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


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

Note: this improves on the summation example (critical region not needed)

- **Initial values of reduction variable**
  - depend 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)
```

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

Introduction to OpenMP: Reduction operations

- Dense MVM + scalar product

```
double s;
…
#pragma omp parallel
{
  for(int j=0; j<niter; j++){

#pragma omp for schedule(...)
    for(int m=0; m<size; m++){
      for(int n=m; n<size; n++){
        y[m]+=a[m*size+n]*x[n];
      }
    }
#pragma omp for reduction(+:s)
    for(int m=0; m<size;m++) s+=x[m]*y[m];
…
  }
}
```

```
double precision, dimension(0:N+1,0:N+1,0:1) :: phi
Double precision :: maxdelta, eps
Integer :: t0,t1,i,k,it
eps=1.d-14;maxdedlta=2.d0*eps; t0=0;t1=1

do while(maxdelta.gt.eps)
   maxdelta=0.d0

!$OMP parallel do reduction(max:maxdelta)
   do k=1,N
      do i=1,N
         phi(i,k,t1)=0.25d0*(phi(i+1,k,t0)+phi(i-1,k,t0)+
                             phi(i,k+1,t0)+phi(i,k-1,t0) )
         maxdelta=max( maxdelta,abs(phi(i,k,t1)-phi(i,k,t0)) )
      enddo
   enddo
!$OMP end parallel do
t0 ←→ t1
enddo
```

Intel Xeon5160; 3.0 GHz;

Performance [MLUPs/sec]

N

8 MB L2

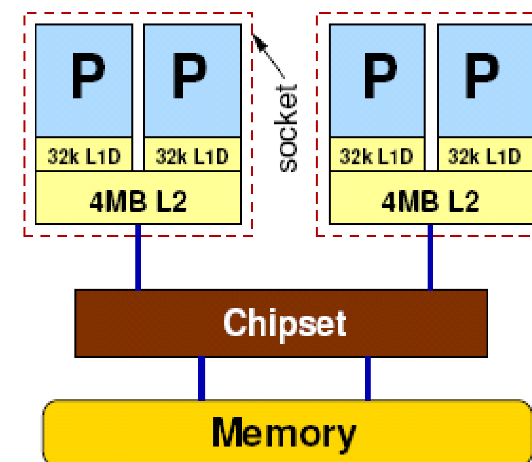
4 MB L2

5.4x speedup

1 thread

2 threads 1 socket

4 threads



Thread pinning via LIKWID

Synchronization and its issues

Memory model

Additional directives

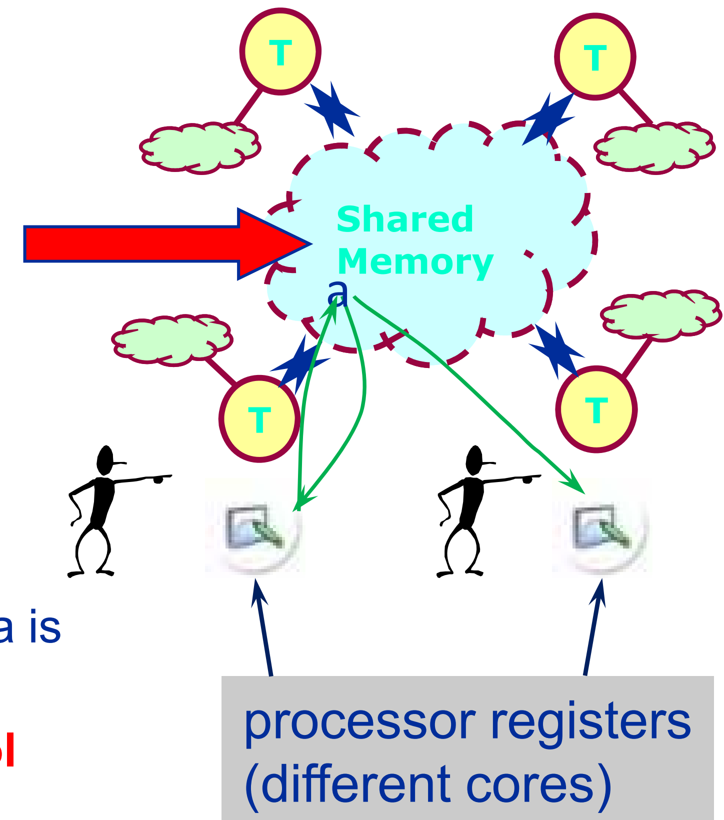
Performance issues

User-defined synchronization

Synchronization: Why do we need it?

Remember OpenMP Memory Model

- ✓ private (thread-local):
  - no access by other threads
- shared: two views
  - **temporary view:** thread has modified data in its registers (or other intermediate device)
  - content becomes inconsistent with that in cache/memory
  - **other threads:** cannot know that their copy of data is **invalid**
  - **Note:** on the **cache** level, the **coherency protocol** guarantees this knowledge



processor registers (different cores)

two threads execute

a = a + 1

race condition

Synchronization: Consequences and (theoretical) remedies

- **For threaded code without synchronization this means**
  - multiple threads write to same memory location → resulting value is **unspecified**
  - one thread reads and another writes → result on (any) reading thread **unspecified**
- **Flush Operation**
  - is performed on a set of (shared) variables

    → flush-set
  - **discards** temporary view:

    → modified values forced to cache/memory (requires exclusive ownership)

    → next read access must be from cache/memory
  - **further** memory operations only allowed after all involved threads complete flush:

    → restrictions on memory in-struction reordering (by compiler)
- **Ensure consistent view of memory:**
  - assumption: want to write a data item with first thread, read it with second
  - order of execution **required**:

1. thread 1 writes to shared variable
2. thread 1 flushes variable
3. thread 2 flushes same variable
4. thread 2 reads variable

... and what must the programmer do?

```
integer :: isync(0:nthrmax)
:
isync(:) = 0    ! dummy for
isync(0) = 1    ! thread 0
!$omp parallel &
!$omp private(myid,neigh,...)
myid = omp_get_thread_num() + 1
neigh = myid - 1
    : ! work chunk 1
isync(myid) = 1
!$omp flush(isync)
do while (isync(neigh) == 0)
!$omp flush(isync)
end do
    : ! work chunk 2,
    : ! dependency on data
    : ! from chunk 1 for myid > 1
!$omp end parallel
```

Producer/consumer example:
relies on state change for scalar integers

- **OpenMP directive for explicit flushing**

`!$omp flush [(var1[,var2,…])]`

- **applicable to all variables with shared scope**
  - including: SAVE, COMMON/module globals, shared dummy arguments, shared pointer dereferences
- **If no variables specified, flush-set**
  - encompasses **all** shared variables (→ potentially slower)
  - which are **accessible** in the scope of the FLUSH directive
- **Note: volatile variables**
  - flush not required

Synchronization: Barrier synchronization

- **Explicit via directive:**

  `!$omp barrier`

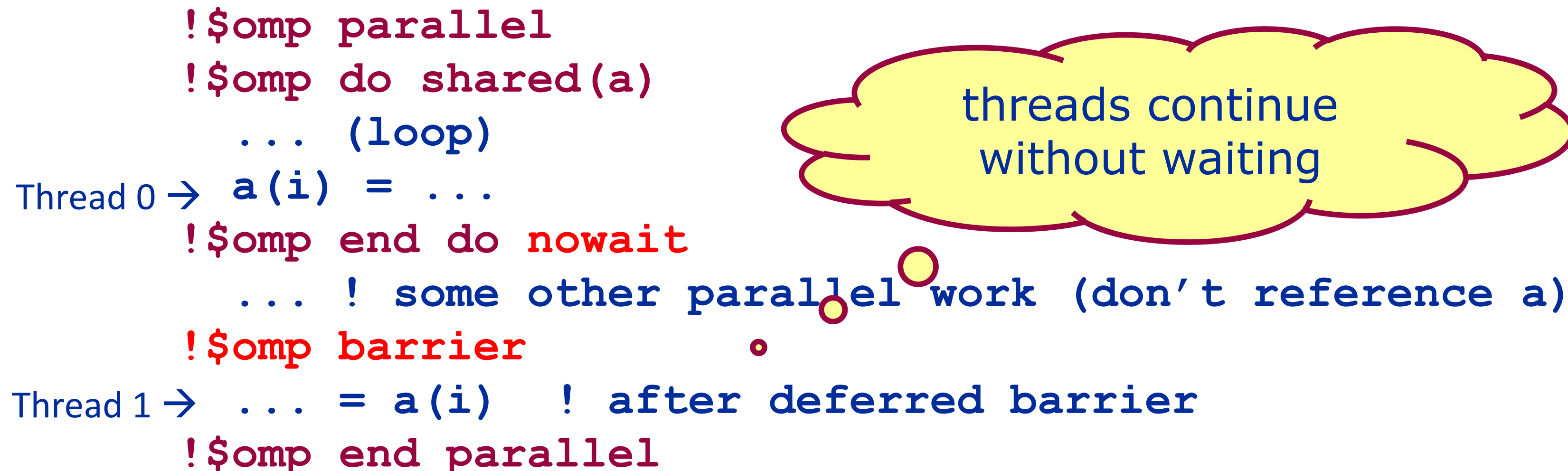
  - synchronization requirement:

    the execution flow of each thread blocks upon reaching the barrier until **all** threads have reached the barrier
  - flush synchronization of all accessible shared variables happens before all threads continue → after the barrier, all shared variables have consistent value visible to all threads
  - barrier may **not** appear within work-sharing code block (e.g. , `!$omp do` block), since this would imply deadlock

- **Implicit for some directives:**
  - at the **beginning and end** of parallel regions
  - at the **end** of `do, single, sections, workshare` blocks unless a `nowait` clause is allowed and specified
  - all threads in the executing team are synchronized
  - this is what makes these directives "easy-to-use"

Relaxing synchronization requirements

- **Use a `nowait` clause**
  - on end do / end sections / end single / end workshare (Fortran)
  - on for / sections / single (C/C++)
  - **removes** the synchronization at end of block
  - potential performance **improvement** (especially if load imbalance occurs within construct)

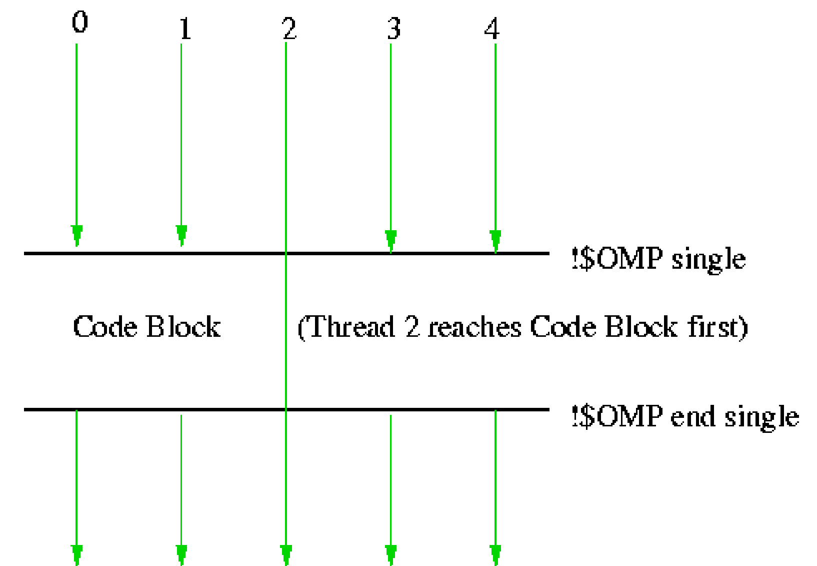


- programmer's responsibility to prevent races

Synchronization Constructs: `single` directive

- The enclosed code is executed by exactly one thread, which one is unspecified
- C/C++:
  `#pragma omp single` *[clause[[,]clause]…] [nowait] new-line*
  structured-block
- The other threads in the team skip the enclosed section of code and continue execution. There is an implied barrier at the exit of the `single` section!
- `single` may not appear within a `parallel do` (deadlock!)

- `nowait` clause at start of parallel region suppresses synchronization



Synchronization: `critical` regions

- **The `critical` and `atomic` directives:**
  - **each** thread executes code (in contrast to `single`)
  - but only **one at a time** within code
  - implies synchronization at exit of code block
  - `atomic`: code block must be a **single line** update of a scalar entity with an intrinsic operation

Fortran:

```
!$omp critical
  block
!$omp end critical
```

```
!$omp atomic
  x = x <op> ...
```

C/C++:

```
# pragma omp critical
  { block }
```

```
# pragma omp atomic
  x = x <op> ... ;
```

Synchronization: `master` directive

Fortran:

```
!$omp master
  block
!$omp end master
```

C/C++:

```
# pragma omp master
  { block }
```

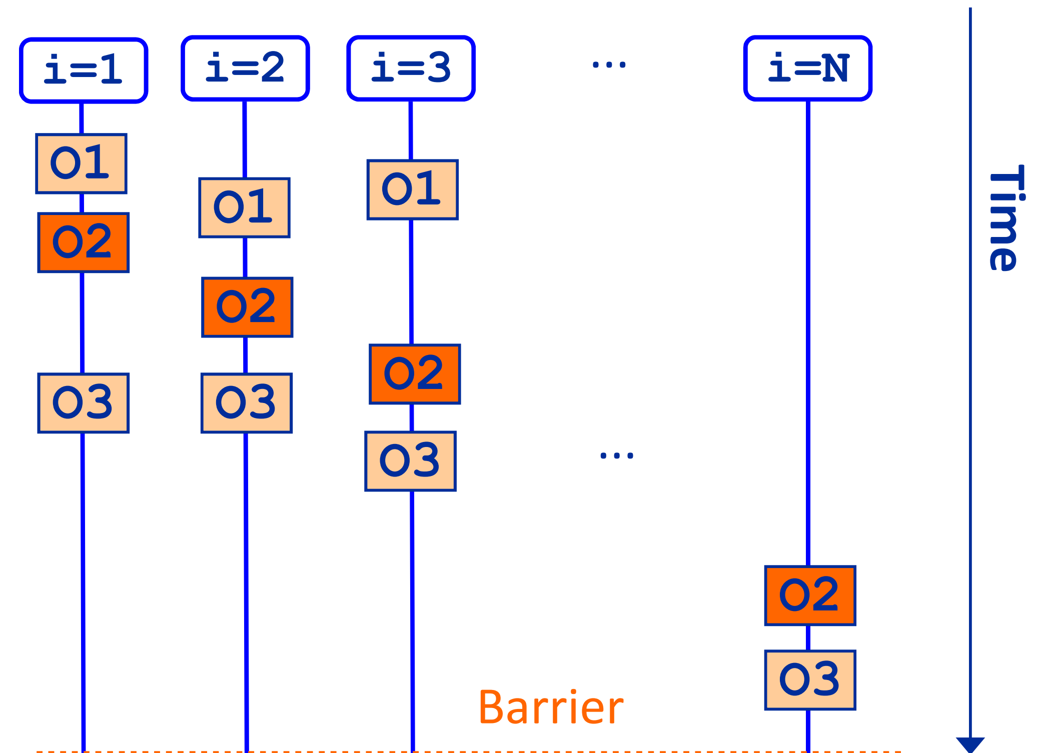
- **Only thread zero (from the current team) executes the enclosed code block**
  - other threads continue **without** synchronization
- **Not all threads must reach the construct**

Synchronization: `ordered` clause and directive

Statements must be within body of a loop

- directive acts similar to **`single`**, threads do work ordered as in sequential execution
- requires **`ordered`** clause on enclosing **`$!OMP do`**
- only effective if code is executed in parallel
- only **one** ordered region per loop
- execution scheme:

```
!$OMP do ordered
do I=1,N
   O1
!$OMP ordered
   O2
!$OMP end ordered
   O3
end do
!$OMP end do
```



Synchronization: Two applications of `ordered`

- **Loop contains recursion**
  - dependency requires serialization
  - only small part of loop (otherwise performance issue)

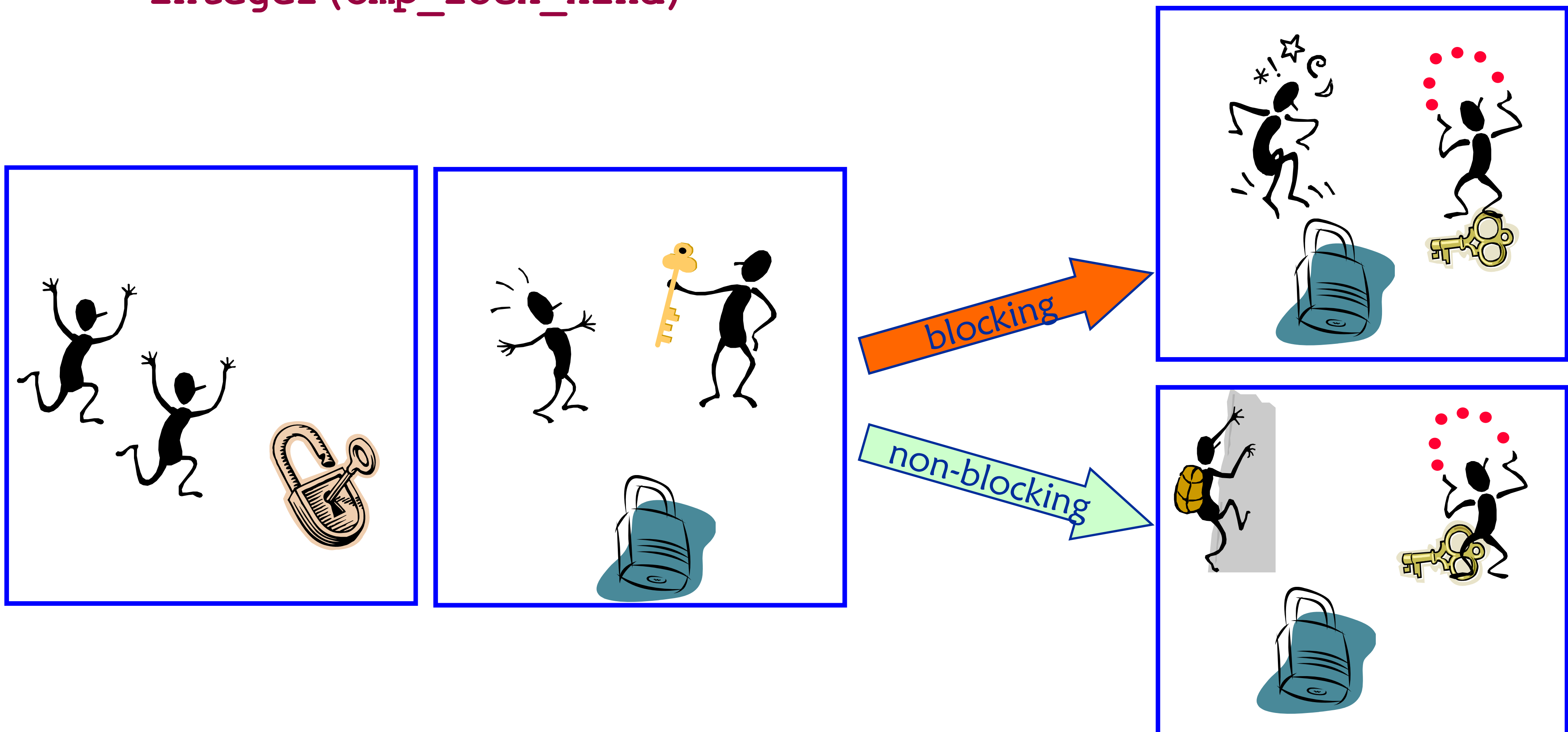
```
!$OMP do ordered
do I=2,N
   ... ! large block
!$OMP ordered
   a(I) = a(I-1) + ...
!$OMP end ordered
end do
!$OMP end do
```

- **Loop contains I/O**
  - it is desired that output (file) be consistent with serial execution

```
!$OMP do ordered
do I=1,N
   ... ! calculate a(:,I)
!$OMP ordered
   write(unit,...) a(:,I)
!$OMP end ordered
end do
!$OMP end do
```

A shared lock variable can be used to implement specifically designed synchronization mechanisms

- In the following, `var` is an `INTEGER` of implementation-dependent kind: `integer(omp_lock_kind)`



Synchronization: Lock routines (1)

- `OMP_INIT_LOCK(var)`

  initialize a lock
  - lock is labeled by `var`
  - objects protected by lock: defined by **programmer** (red balls on previous slide)

  initial state is unlocked

  `var` **not associated with a lock before this subroutine is called**

- `OMP_DESTROY_LOCK(var)`

  disassociate `var` **from lock**

  `var` **must have been initialized (see above)**

Synchronization: Lock routines (2)

For all following calls: lock variable `var` must have been initialized

- `OMP_SET_LOCK(var)`:
  blocks if lock not available
  set ownership and continue execution if lock available
- `OMP_UNSET_LOCK(var)`:
  release ownership of lock
  ownership must have been established before
- **logical** function
  `OMP_TEST_LOCK(var)`:
  does **not** block, **tries to set** ownership
  → thread receiving failure can go away
  and do something else

Note: before OpenMP 2.5 lock variables strictly required an

```
!$omp flush(var)
```

before dereferencing

Conditional parallelism

- **Put an IF clause on a parallel region**

```
!$omp parallel if (n > 8000)
!$omp do
  do i=1, n
   a(i) = b(i) + c(i)*d(i)
  end do
!$omp end do
!$omp end parallel
```

  - specify a scalar logical argument
  - requires manual tuning for properly dealing with thread count dependency etc.

- **Specific use:**
  - suppress nested parallelism in a library routine
  - use the logical function **omp_in_parallel()** from the OpenMP run time

```
!$omp parallel if &
!$omp (.not. omp_in_parallel())
  … ! parallel region
!$omp end parallel
```

- **Allows execution of a code region in serial or parallel, depending on a condition**

- **C/C++:**

```
#pragma omp parallel if(condition)
        structured-block
```

- **Usage:**
  - disable parallelism dynamically
  - define crossover points for optimal performance
    - may require manual or semi-automatic tuning

Example for crossover points:
Vector triad with 4 threads on 4-CPU Itanium2

Vector Triads on 1.3 GHz IA64 SMP (4 Threads)

Default

With IF

```
... if (N >= 7000)
```

Performance (MFlops/s)

8000
7000
6000
5000
4000
3000
2000
1000
0

10 100 1000 10000 100000 1e+06

Vector Length

Parallel loop overhead

Binding of directives
Global variables

Notes for library writers

Which parallel region does a directive refer to?

- **`do, sections, single, master, barrier, task:`**

  to (dynamically) closest enclosing parallel region, if one exists

  if not → directive is "orphaned": only one thread used if not bound to a parallel region
  - close nesting of `do, sections` **not** allowed
  - close nesting of barriers inside explicit `task`s (see later) **not** allowed
- **`ordered`:**
  - binds to dynamically enclosing `do`
  - not in dynamical extent of `critical` region.
- **`atomic, critical`: mutual exclusion applies for all threads, not just current team**

Orphaning

```
!$OMP parallel
   …
   call foo(…)
  …
!$OMP end parallel
   call foo(…)
```

```
subroutine foo(…)
   …
!$OMP do
   do I=1,N
   …
   end do
!$OMP end do
```

Inside parallel region:

`foo` called by all threads

Outside parallel region:

`foo` called by one thread

- **OpenMP directives in `foo` are orphaned**
  - since they may or may not bind to a parallel region
  - decided at runtime
  - in both cases executed correctly

Example for incorrect nesting

```
!$OMP parallel
!$OMP do
   do i=1,n
     call foo(…)
   end do
!$OMP end do
!$OMP end parallel
```

```
subroutine foo(…)
   …
!$OMP do
   do I=1,N
   …
   end do
!$OMP end do
```

Not allowed:

`do` **nested within a** `do`

Scoping for global variables: the `threadprivate` directive

- **Remember that**
  - global module variables
  - variables in COMMON

  by default have `shared` scope?
- **It is possible to change this:**

```
module mod_stuff
  integer :: my_priv
!$omp threadprivate mypriv
  :
end module
```

```
COMMON / my_pr_block/ :: …
!$omp threadprivate my_pr_block
```

  - named COMMON blocks only

- **Notes:**
  - every instance of the COMMON block must be privatized
  - all variables in the COMMON block become threadprivate
  - in a serial region, either the default-initialized value or the definition on the master thread is valid. Masking applies as for private variables
- **Threadprivate storage:**
  - easier to get threading „right"
  - but not always an option

Threadprivate entities in parallel regions

- **Start of first parallel region:**
  - thread-individual copies are created
  - these are **undefined** unless a `copyin` clause is given:

```
!$omp parallel copyin(mypriv)
   : ! threaded execution
   … = mypriv
!$omp end parallel
```

- **Subsequent parallel regions:**
  - thread-individual copies retain their values (by thread) if

1. second parallel region not nested inside first
2. same number of threads is used
3. no dynamic threading is used

Note: none of the potential viola-tions of the above three rules have been dealt with in this course