Advanced OpenMP Programming

R. Bader (LRZ)

G. Hager (NHR@FAU)

V. Weinberg (LRZ)

M. Wittmann

Work Sharing Schemes

Loops and loop scheduling

Collapsing loop nests

Parallel sections

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:**

```
#pragma omp for schedule(...)
```

```
!$OMP do schedule(...)
```

```
                static
schedule( dynamic [,chunk] )
                guided
```

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

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

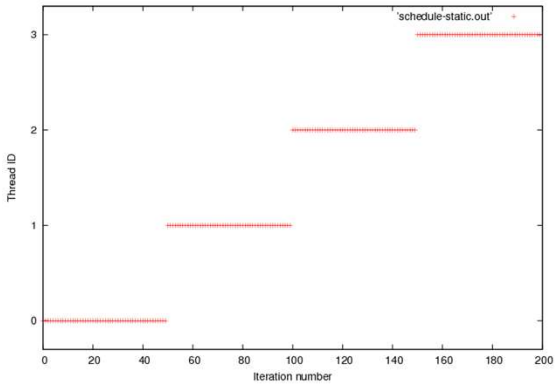
  - minimal overhead (precalculated work assignment)

- **Dynamic scheduling**
  - `schedule(dynamic, 10)`

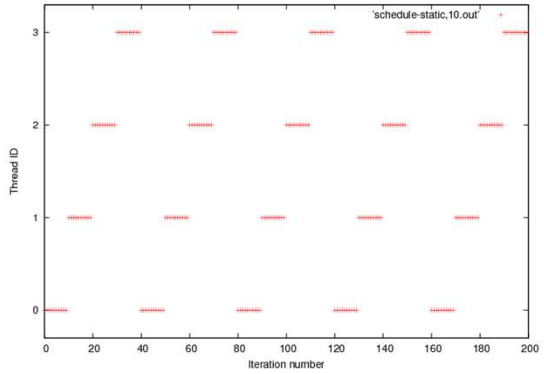
both threads take long to complete their chunk (workload imbalance)

  - after a thread has completed a chunk, it is assigned a new one, until no chunks are left
  - synchronization **overhead**
  - default chunk value is **1**

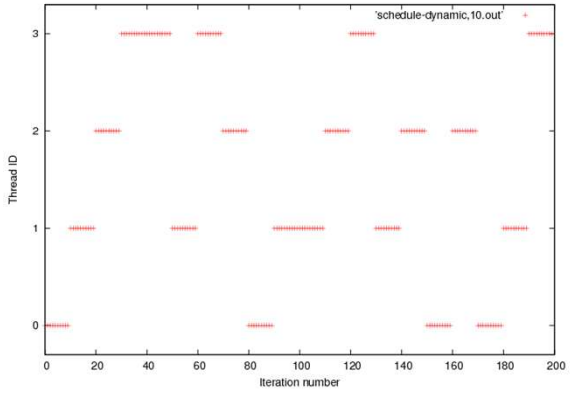
OpenMP Scheduling of simple for loops



OMP_SCHEDULE=static



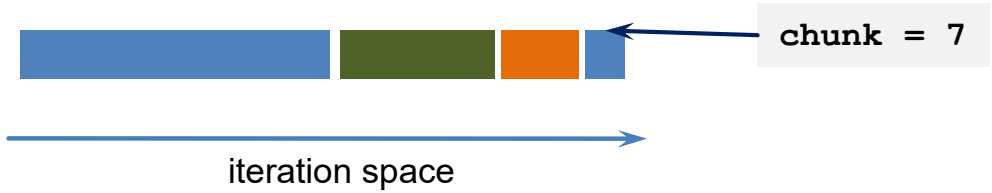
OMP_SCHEDULE=static,10



OMP_SCHEDULE=dynamic,10

Guided scheduling

- **Size of chunks in dynamic schedule**
  - too small → large overhead
  - too large → load imbalance
- **Guided scheduling: dynamically vary chunk size.**
  - Size of each chunk is proportional to the number of unassigned iterations divided by the number of threads in the team, decreasing to chunk-size (default = 1).
- **Chunk size:**
  - means minimum chunk size (except perhaps final chunk)
  - default value is 1



  - Both dynamic and guided scheduling useful for handling poorly balanced and unpredictable workloads.

Deferred scheduling

- **auto: automatic scheduling**
  - Programmer gives implementation the freedom to use any possible mapping.
- **Decided at run time:**

```
!$OMP do schedule(runtime)
```

```
#pragma omp for schedule(runtime)
```

- **runtime:**
  - determine by either setting `OMP_SCHEDULE`, and/or calling `omp_set_schedule()` (overrides env. setting)
  - find which is active by calling `omp_get_schedule()`

- **Examples:**
  - environment setting:

```
export OMP_SCHEDULE="guided,4"
./a.out
```

  - call to API routine:

```
call omp_set_schedule(omp_sched_dynamic,4)
!$OMP parallel
!$OMP do schedule(runtime)
   do
      …
   end do
!$OMP end do
```

```
omp_set_schedule(omp_sched_dynamic, 4)
#pragma omp parallel
#pragma omp schedule(runtime)
   for (…) { }
```

runtime scheduling and `OMP_SCHEDULE` is not set: implementation chooses a schedule

Collapsing loop nests

- **Collapse 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
```

argument specifies number of loop nests to flatten

```
#pragma omp for collapse(2)
for (k=0; k<kmax; ++k)
  for (j=0; j<jmax; ++j)
    ...
```

- **Restrictions:**
  - iteration space computable at entry to loop (rectangular)
  - **`CYCLE`** (Fortran) or **`continue`** (C/C++) only 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
- **Shooting yourself in the foot**
  - modified variables must not be accessed unless explicit synchronization is performed
  - use a `barrier` for this
- **Example: multiple loops in parallel region**

```
!$omp parallel
!$omp do
  do k=1, kmax_1
    a(k) = a(k) + b(k)
  end do
!$omp end do nowait
  ! code not involving
  ! r/w of a, writes to b
!$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

```
#pragma omp parallel
{
  #pragma omp for nowait
  for (int k = 0; k < kmax_1; ++k) {
    a[k] += b[k]
  }
  /* code not involving */
  /* r/w of a, writes to b */
  #pragma omp for
  for (int k = 0; k < kmax_2; ++k) {
    c[k] *= d[k]
  }
}
```

Explicit barrier synchronization

- `barrier` construct is a stand-alone directive
- barrier synchronizes all threads
- each barrier must be encountered by all threads in the team or by non at all.

```
!$omp parallel
!$omp do
  do k=1, kmax_1
    a(k) = a(k) + b(k)
  end do
!$omp end do nowait
  ! code not involving
  ! r/w of a, writes to b
!$omp barrier
!$omp do
  do k=1, kmax_1
    a(k) = a(k) + b(k)
  end do
!$omp end do
!$omp end parallel
```

do not synchronize

explicit synchronization

```
#pragma omp parallel
{
  #pragma omp for nowait
  for (int k = 0; k < kmax_1; ++k) {
    a[k] += b[k]
  }
  /* code not involving */
  /* r/w of a, writes to b */
  #pragma omp barrier
  #pragma omp for
  for (int k = 0; k < kmax_1; ++k) {
    a[k] += b[k]
  }
}
```

Parallel sections

- **Non-iterative work-sharing construct**
  - distribute a set of structured blocks

```
!$omp parallel
!$omp sections
!$omp section
   ! code block 1
!$omp section
   ! code block 2
...
!$omp end sections
!$omp end parallel
```

thread 0

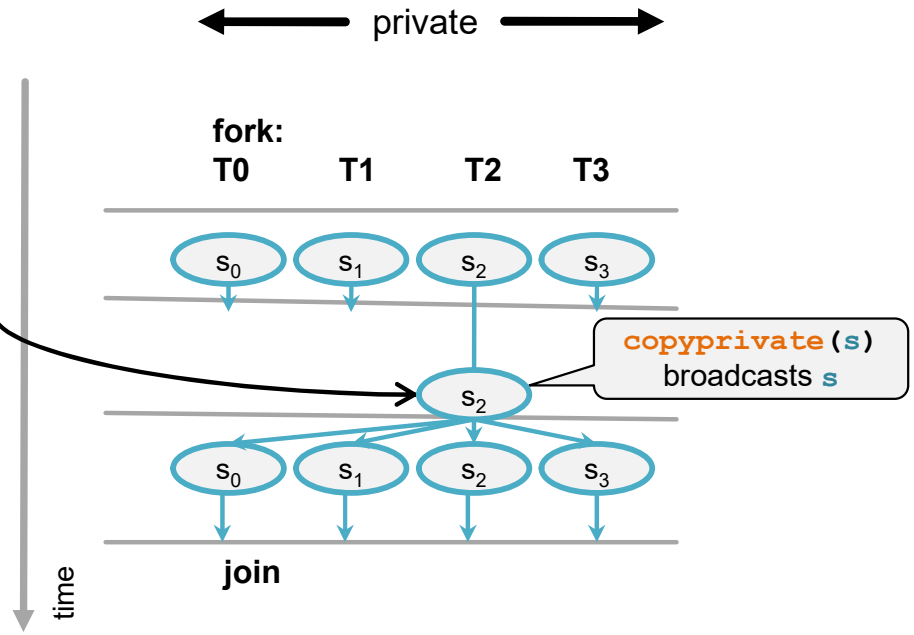
thread 1

synchronization

  - each block executed exactly once by one of the threads in team
- **Allowed clauses on sections:**
  - `private`, `firstprivate`, `lastprivate`, `reduction`, `nowait`
- **Restrictions:**
  - `section` directive must be within lexical scope of sections directive
  - `sections` directive binds to innermost parallel region
  - → only the threads executing the binding parallel region participate in the execution of the section blocks and the implicit barrier (if not eliminated with `nowait`)
- **Scheduling to threads**
  - implementation-dependent
  - if there are more threads than code blocks: excess threads wait at synchronization point

The single directive

```
#pragma omp parallel
{
  double s = …;
  #pragma omp single copyprivate(s)
  {
    s = …
  }
  … = … + s
}
```



- one thread only executes enclosed code block
- all other threads wait until block completes execution
- allowed clauses: **`private`**, **`firstprivate`**, **`copyprivate`**, **`nowait`**
- use for updates of shared entities, but …
  - single – really a worksharing directive?

- **`copyprivate`** and **`nowait`** clauses: appear on **`end single`** in Fortran, on **`single`** in C/C++

```
!$omp single
  s = …
!$omp end single copyprivate(s)
```

Combining `single` with `nowait`

- **Implement a self-written work scheduler**
  - not the most efficient method → preferably use tasking (see later)
  - one possible scheme:

```
  : assign work for iteration 1
!$omp parallel
  do iw=1, nwork
!$omp single
  : ! assign work for iteration iw+1 to threads, "prefetching"
  : ! (using a non-trivial amount of time e.g. I/O)
!$omp end single nowait
  : ! other threads continue and work on iteration iw
!$omp barrier
  end do ! iw
!$omp end parallel
```

no omp do!
all threads
execute this loop

```
 /* assign work for iteration 1 */
#pragma omp parallel
for (int i = 0; i < nwork; ++i) {
  #pragma omp single nowait
  {
  /* assign work for iteration i+1 t
  /* (using a non-trivial amount of
  }
  /* other threads continue and work
  #pragma omp barrier
}
```

Combining regions and work sharing

- **Example:**

```
!$OMP parallel do
...
!$OMP end parallel do
```

```
#pragma omp parallel for
...
```

- **is equivalent to**

```
!$omp parallel
!$omp do
...
!$omp end do
!$omp end parallel
```

```
#pragma omp parallel
#pragma omp for
...
```

- **Applies to most work-sharing constructs**
  - `do/for`
  - `Sections`
  - `Workshare (Fortran only)`

- **Notes:**
  - clauses for work-sharing constructs can appear on combined construct
  - the reverse is not true

    `shared` can only appear in a parallel region

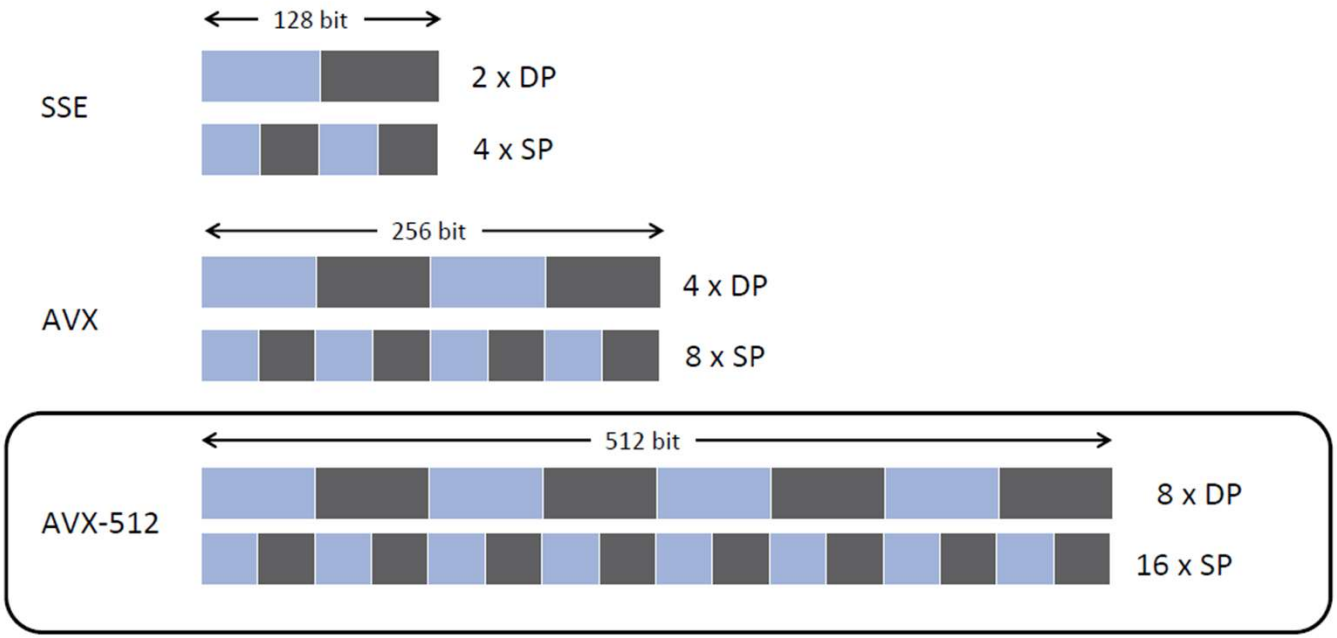
Vectorization with OpenMP SIMD

Acknowledgements:

M. Klemm (OpenMP ARB), C. Terboven (RWTH Aachen)

SIMD on Intel Architecture

- Width of SIMD (Single Instruction, Multiple Data) registers has been growing in the past:



Before OpenMP 4.0

- **Support required vendor-specific extensions**
  - Programming models (e.g. Intel Cilk Plus)
  - Compiler pragmas (e.g. **`#pragma vector`**)
  - Low-level constructs (e.g. **`_mm_add_pd()`**)

```
#pragma omp parallel for
#pragma vector always
#pragma ivdep
for (int i = 0; i < n; i++) {
  a[i] = b[i] + …;
}
```

You need to trust your compiler to do the right thing

SIMD Loop Construct

- **Vectorize a loop nest**
  - Cut loop into chunks that fit a SIMD vector register
  - No parallelization of the loop body

C/C++

```
#pragma omp simd [clause, …]
for-loops
```

Fortran

```
!$omp simd [clause, …]
do-loops
!$omp end simd
```

- **`simd` construct can be applied to a loop to indicate that the loop can be transformed into a SIMD loop**
  - multiple iterations of the loop can be executed concurrently using SIMD instructions

- **`simd` specifies that there are no dependencies among loop iterations**
  - see `safelen` clause

SIMD Loop Construct: Clauses

- **`private (var-list)`**
  uninitialized vectors for variables in `var-list`
- **`reduction (op:var-list)`**
  create private variables for `var-list` and apply reduction operator `op` at the end of the construct
- **`simdlen (length)`**
  length is treated as a hint that specifies the preferred number of iterations to be executed concurrently
- **`safelen (length)`**
  maximum number of iterations that can run concurrently without breaking a dependence
- **`linear (list[:linear-step])`**
  the variable's value is in relationship with the iteration number $x_i = x_{orig} + i * \text{linear-step}$
- **`aligned (list[:alignment])`**
  specifies that the list items have a given alignment
- **`collapse (n)`**
  collapse n nested loops into a single iteration space

Examples of the SIMD construct

```
#pragma omp simd
for (i=0; i<n); i++)
  a[i] = b[i] + c[i];
```

```
#pragma omp simd reduction(+:t1) collapse(2)
for (i=0; i<n; i++)
  for (j=0; j<m; j++)
     t1 += func1(b[i], c[j]);
```

SIMD Worksharing Construct

- **Parallelize and vectorize a loop next**
  - Distribute a loop's iteration space across a thread team
  - Subdivide loop chunks to fit a SIMD vector register

C/C++

```
#pragma omp for simd [clause, …]
for-loops
```

Fortran

```
!$omp do simd [clause, …]
do-loops
!$omp end
```

- **SIMD Function Vectorization**
  - Declare one or more functions to be compiled for calls from a SIMD loop
  - Not covered in this course

Synchronization and its issues

Memory model

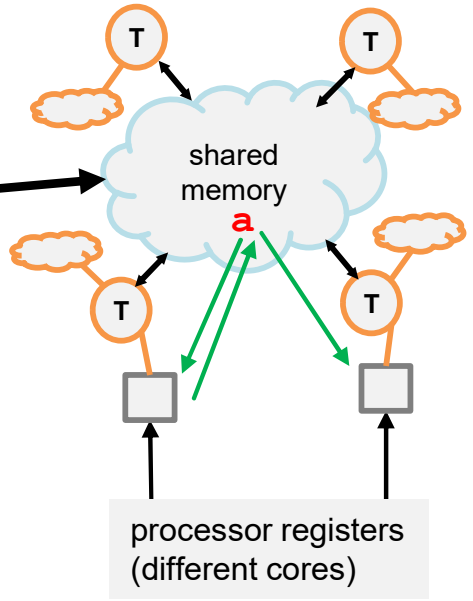
Additional directives

Performance issues

User-defined synchronization

Why do we need synchronization?

- **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**



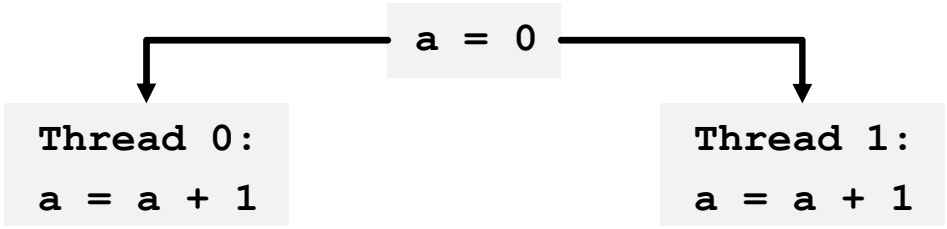
two threads execute

```
a = a + 1
```

in same parallel region

→ **race condition**

Possible results



- **Following results could be obtained on each thread**
  - a after completion of statement:

Thread 0	Thread 1
1	1
1	2
2	1

  - may be different from run to run, depending on which thread is the last one
  - after completion of parallel region, may obtain 1 or 2.

Consequences and (theoretical) remedies

- **For threaded code without synchronization this means**
  - multiple threads write to same memory location
    - resulting value is unspecified
  - some threads read and another writes
    - result on reading threads unspecified
- **Flush Operation**
  - is performed on a set of (shared) variables or on the whole thread-visible data state of a program
    → flush-set
  - discards temporary view:
    → modified values forced to cache/memory
    → next read access must be from cache/memory
  - further memory operations only allowed after all involved threads complete flush:
    - restrictions on memory instruction 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

OpenMP flush syntax

- **OpenMP directive for explicit flushing**

```
!$omp flush [(var1[,var2,…])]
```

- **Stand-alone directive**
- **applicable to all variables with shared scope**
  - including: **SAVE**, **COMMON**/module globals, shared dummy arguments, shared pointer dereferences
- **If no variables specified, the flush-set**
  - encompasses all shared variables which are accessible in the scope of the FLUSH directive
  - potentially slower
- **Implicit flush operations (with no list) occur at:**
  - All explicit and implicit barriers
  - Entry to and exit from critical regions
  - Entry to and exit from lock routines

Barrier synchronization

- **Explicit via directive:**
  - 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 specified (where allowed)
  - all threads in the executing team are synchronized
  - this is what makes these directives “easy-and-safe-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

Critical regions

- **The `critical` and `atomic` directives:**
  - each thread arriving at the code block executes it (in contrast to **`single`**)
  - mutual exclusion: only one at a time within code block
  - **`atomic`**: code block must be a single line update of a scalar entity of intrinsic type with an intrinsic operation

Fortran

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

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

unary operator also allowed

C/C++

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

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

Synchronizing effect of critical regions

- **Mutual exclusion is only assured for the statements inside the block**
  - i.e., subsequent threads executing the block are synchronized against each other
- **If other statements access the shared variable, may be in trouble:**

C/C++

```
#pragma omp parallel
{
  :
  #pragma omp atomic
  x = x + y
  :
  a = f(x, …)
}
```

Fortran

```
!$omp parallel
  :
!$omp atomic
  x = x + y
  :
  a = f(x, …)
!$omp end parallel
```

- Race on read to x.
- A barrier is required before this statement to assure that all threads have executed their atomic updates

Named critical

- **Consider multiple updates**
  - same shared variable

thread 0

```
subroutine foo()
!$omp critical
  x = x + y
!$omp end critical
```

thread 1

```
subroutine bar()
!$omp critical
  x = x + z
!$omp end critical
```

    - critical region is global: OK

  - different shared variables

```
subroutine foo()
!$omp critical
  x = x + y
!$omp end critical
```

```
subroutine bar()
!$omp critical
  w = w + z
!$omp end critical
```

    - mutual exclusion not required
    - unnecessary loss of performance

- **Solution:**
  - use named criticals

```
subroutine foo()
!$omp critical (foo_x)
  x = x + y
!$omp end critical (foo_x)
```

```
subroutine bar()
!$omp critical (foo_w)
  w = w + z
!$omp end critical (foo_w)
```

  - mutual exclusion only if same name is used for `critical`
- **`atomic` is bound to updated variable**
  - problem does not occur

The master directive (depricated)

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**
  - There is no implied barrier either on entry to, or exit from, the master construct. Other threads continue without synchronization
- **Not all threads must reach the construct**
  - if the master thread does not reach it, it will not be executed at all
- **Equivalent to:**

```
if (omp_get_thread_num() == 0) { … }
```

The masked directive

≥ v5.1

Fortran

```
!$omp masked [filter(scalar-integer-expression)]
 block
!$omp end masked
```

C/C++

```
# pragma omp masked [filter(integer-expression)]
{ block }
```

- only threads selected by the filter clause execute the structured block
- other threads in the team do not execute the associated structured block.

- If a filter clause is present on the construct and the parameter specifies the thread number of the current thread in the current team then the current thread executes the associated structured block.

- No implied barrier on entry to, or exit from, the `masked` construct.

The `ordered` clause and directive

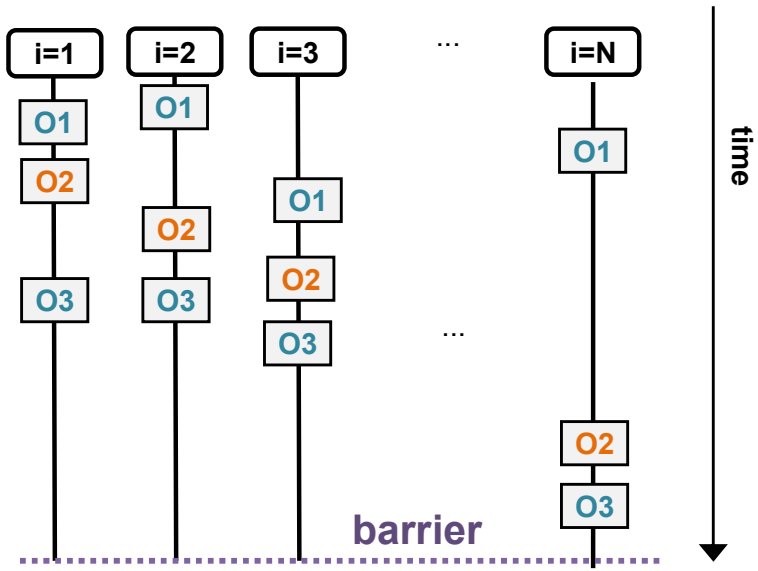
- **Statements must be within body of a loop**
  - directive acts similar to **`single`**
    - threads do work ordered as in sequential execution
    - execution in the order of the loop iterations
  - requires **`ordered`** clause on enclosing **`do`**/**`for`** construct
  - only effective if code is executed in parallel
  - only one ordered region per loop

C/C++

```
#pragma omp for ordered
for (i=0; i<N; ++i) {
  O1
  #pragma omp ordered
  { O2 }
  O3
}
```

Fortran

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



Two applications of ordered

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

Fortran

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

C/C++

```
#pragma omp for ordered
for (i=1; i<N; ++i) {
  ... /* large block */
  #pragma omp ordered
  a[i] = a[i-1] + ...
}
```

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

Fortran

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

C/C++

```
#pragma omp for ordered
for (i=0; i<N; ++i) {
  ... /* calculate a[i] */
  #pragma omp ordered
    printf("%e ", a[i]);
  }
}
```

Mutual exclusion with locks

- **A shared lock variable can be used to implement specifically designed synchronization mechanisms**
  - In the following, var is of type
    - `Fortran: integer(omp_lock_kind)`
    - `C/C++:  omp_lock_t`
  - OpenMP lock variables must be only accessed by the lock routines

- **Mutual exclusion bound to objects**
  - more flexible than critical regions

OpenMP locks

- **An OpenMP lock can be in one of the following 3 stages:**
  - uninitialized
  - unlocked
  - locked
- **The task that sets the lock is then said to own the lock.**
- **Only a task that sets the lock, can unset the lock, returning it to the unlocked stage.**
- **2 types of locks are supported:**
  - simple locks
    - Can only be locked if unlocked.
    - A thread may not attempt to re-lock a lock it already has acquired.
  - nestable locks
    - Owning thread can lock multiple times
    - Owning thread must unlock the same number of times it locked it

Lock routines (1)

- **Fortran:** `omp_init_lock(var)`
  C/C++ `omp_init_lock(omp_lock_t *var)`
  - initialize a lock
  - initial state is unlocked
  - what resources are protected by lock: defined by developer
  - **`var`** not associated with a lock before this routine is called

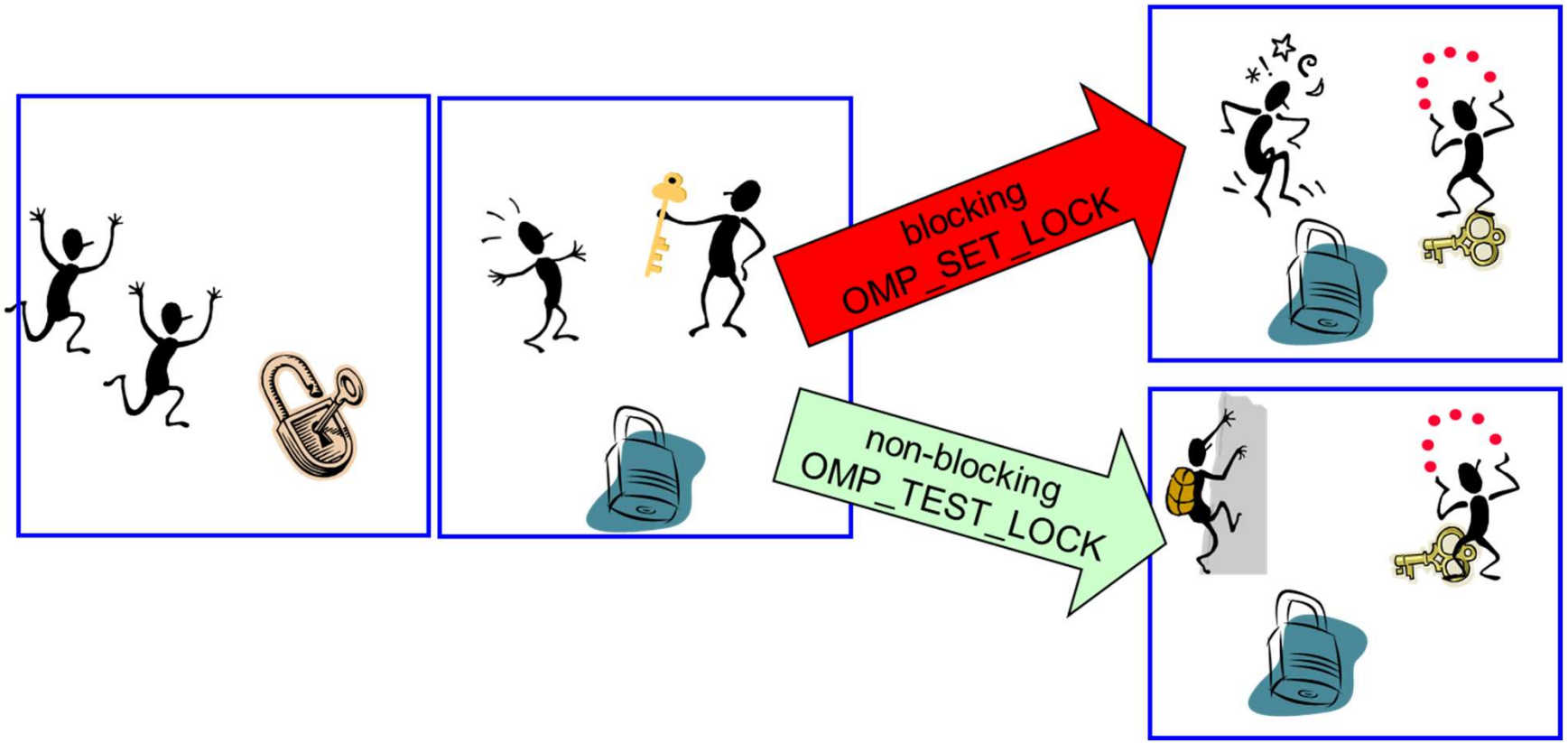
- **Fortran:** `omp_destroy_lock(var)`
  C/C++: `omp_destroy_lock(omp_lock_t *var)`
  - disassociate **`var`** from lock
  - precondition:
    - **`var`** must have been initialized
    - **`var`** must be in unlocked state

Lock routines (2)

- **Assuming: lock variable `var` has been initialized**

- **Fortran:** `omp_set_lock(var)`
  C/C++: `void omp_set_lock(omp_lock_t *var)`
  - blocks if lock not available
  - set ownership and continue execution if lock available
- **Fortran:** `omp_unset_lock(var)`
  C/C++: `void omp_unset_lock(omp_lock_t *var)`
  - release ownership of lock
  - ownership must have been established before
- **Fortran:** `logical function omp_test_lock(var)`
  C/C++: `int omp_test_lock(omp_lock_t *var)`
  - does not block, tries to set ownership
  - returns true if lock was set, false if not
  - allows to do something else while lock is hold by another thread

Lock routines



Example for using locks

```
use omp_lib
integer(omp_lock_kind) :: lock

call omp_init_lock(lock)

!$omp parallel
  ...
  call omp_set_lock(lock)
  ! use resource protected by lock
  call omp_unset_lock(lock)
  ...
!$omp end parallel

call omp_destroy_lock(lock)
```

acts like a **critical** region

```
use omp_lib
integer(omp_lock_kind) :: lock

call omp_init_lock(lock)

!$omp parallel
  ...
  do while (.not. omp_test_lock(lock))
    ! work unrelated to lock protected
    ! resource
  end do
  ! use lock protected resource
  call omp_unset_lock(lock)
  ...
!$omp end parallel

call omp_destroy_lock(lock)
```

loop until lock calling thread hold lock

Example for using locks

```
#include <omp.h>
omp_lock_t lock;

omp_init_lock(&lock);

#pragma omp parallel
{
  ...
  omp_set_lock(&lock)
  /* use resource protected
     by lock */
  omp_unset_lock(&lock)
  ...
}

omp_destroy_lock(&lock)
```

acts like a **critical** region

```
#inclue <omp.h>
omp_lock_t lock;

omp_init_lock(&lock)

#pragma omp parallel
{
  ...
  while (!omp_test_lock(&lock)) {
    /* work unrelated to lock
       protected resource */
  }
  /* use lock protected
     resource */
  omp_unset_lock(&lock)
  ...
}

omp_destroy_lock(&lock)
```

loop until lock calling thread hold lock

Nestable Locks

- **replace `omp_*_lock` by `omp_*_nest_lock`**

- **task owning a nestable lock may re-lock it multiple times**
  - a nestable lock is available if it is either unlocked

  or
  - it is already owned by the task executing `omp_set_nest_lock()` or `omp_test_nest_lock()`

- **re-locking increments nest count**
- **releasing the lock decrements nest count**
- **lock is unlocked once nest count is zero**

Tasking

Work sharing for irregular problems, recursive problems and information structures

Acknowledgements:

M. Klemm (AMD) / L. Meadows / T. Mattson (Intel)

Task Execution Model

- **Supports unstructured parallelism**
  - unbounded loops

```
while (<expr>) {
   ...
}
```

```
do while (<expr>
   ...
end do
```

  - recursive functions

```
void myfunc(<args>)
{
   ...
  myfunc(<newargs>)
   ...
}
```

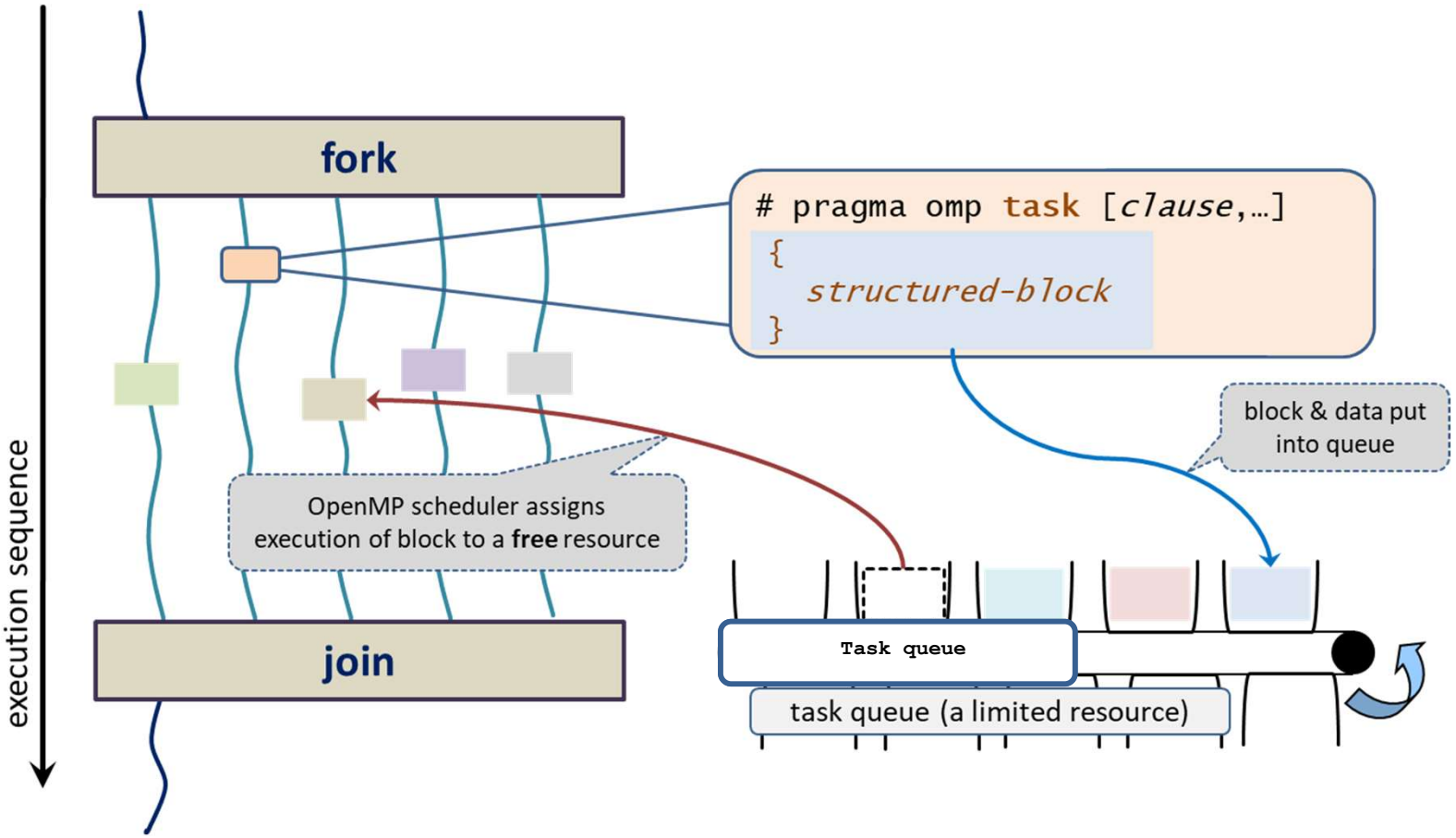
- **Several scenarios are possible**
  - single creator, multiple creators, nested tasks,
  - All threads in the team are candidates to execute tasks

- **Example of unstructured parallelism**

```
#pragma omp parallel
#pragma omp single
while (elem != NULL) {
  #pragma omp task
  compute(elem);

  elem = elem->next;
}
```

The Execution Model



The `task` Construct

- **Deferring (or not) a unit of work (executable for any member of the team)**

```
#pragma omp task [clause[[,] clause]...]
{structured-block}
```

```
!$omp task [clause[[,] clause]...]
…structured-block…
!$omp end task
```

- **Clauses:**
  - data environment:
    - **`private`**, **`fistprivate`**, **`default(shared|none)`**, **`in_reduction(r-id:list)`**
  - Dependencies:
    - **`depend(dep-type: list)`**
  - Scheduler restriction:
    - **`untied`**
  - Scheduler hints:
    - **`priority(priority-value)`**
    - **`affinity(list)`**
  - cutoff strategies:
    - **`if(scalar-expression)`**
    - **`mergable`**
    - **`final(scalar-expression)`**
  - Other clauses:
    - **`allocate(allocator:] list)`**
    - **`detach(event-handler)`**

What is a Task?

- **Make OpenMP worksharing more flexible:**
  - allow the programmer to package code blocks and data items for execution
    - this by definition is a task
  - and assign these to an encountering thread
  - possibly defer execution to a later time („work queue“)
- **Introduced with OpenMP 3.0 and extended over time**

- **When a thread encounters a `task` construct, a task is generated from the code of the associated structured block.**
- **Data environment of the task is created (according to the data-sharing attributes, defaults, …)**
  - „Packaging of data“
- **The encountering thread may immediately execute the task, or defer its execution. In the latter case, any thread in the team may be assigned the task.**

Example: Processing a Linked List

```
typedef struct {
  list *next;
  contents *data;
} list;

void process_list(list *head)
{
  #pragma omp parallel
  {
    #pragma omp single
    {
      list *p = head;
      while(p) {
        #pragma omp task
        { do_work(p->data); }
        p = p->next;
      }
    } /* all tasks done */
  }
}
```

Typical task generation loop:

```
#pragma omp parallel
{
  #pragma omp single
  {
    while(p) {
      #pragma omp task
      { /* taks code */ }
    }
  } /* all tasks done */
}
```

Example: Processing a Linked List

```
typedef struct {
  list *next;
  contents *data;
} list;

void process_list(list *head)
{
  #pragma omp parallel
  {
    #pragma omp single
    {
      list *p = head;
      while (p) {
        #pragma omp task
        { do_work(p->data); }
        p = p->next;
      }
    } /* all tasks done */
  }
}
```

- **Features of this example:**
  - one of the threads has the job of generating all tasks
  - synchronization: at the end of the `single` block for all tasks created inside it
  - no particular order between tasks is enforced here
  - data scoping default for task block:
    - `firstprivate`
    - iterating through p is fine
    - this is the „packaging of data“ mentioned earlier
  - task region: includes call of `do_work()`

The `if` Clause

- **When `if` argument is false –**
  - task becomes an undeferred task
  - task body is executed immediately by encountering thread
  - all other semantics stay the same (data environment, synchronization) as for a „deferred“ task

```
#pragma omp task if (sizeof(p->data) > threshold)
{ do_work(p->data); }
```

- **User-directed optimization:**
  - avoid overhead for deferring small tasks
  - cache locality / memory affinity may be lost by doing so

Further aspects of tasking not covered in detail

- **Task Synchronization**
  - Task Synchronization with `barrier` and `taskwait`
  - Task Synchronization with `taskgroup`
- **Task Switching**
  - The `taskyield` Directive
- **Task Reductions**
  - Task Reductions using the `taskgroup` Construct
- **Task Loops**
  - The `taskloop` Construct
- **Task Dependencies**
  - The `depend` Clause

OpenMP Resources

- OpenMP Webpage https://www.openmp.org/
- Specification https://www.openmp.org/specifications/
- OpenMP Books https://www.openmp.org/resources/openmp-books/
- IWOMP Conference https://www.iwomp.org/
  - IWOMP 2025 will be held in the week of Sep 29-Oct 3 in conjunction with EuroMPI and the MPI Forum meetings.
- OpenMP Reference Guide (Cheat Sheet) https://www.openmp.org/resources/refguides/