Advanced OpenMP Programming

R. Bader (LRZ)
G. Hager (RRZE)
V. Weinberg (LRZ)
20 Years of OpenMP History

In spring, 7 vendors and the DOE agree on the spelling of parallel loops and form the OpenMP ARB. By October, version 1.0 of the OpenMP specification for Fortran is released.

First hybrid applications with MPI* and OpenMP appear.

Minor modifications. 1.1

The merge of Fortran and C/C+ specifications begins.

Unified Fortran and C/C++: Bigger than both individual specifications combined. The first International Workshop on OpenMP is held. It becomes a major forum for users to interact with vendor.

Supports offloading execution to accelerator and coprocessor devices, SIMD parallelism, and more. Expands OpenMP beyond traditional boundaries.

Supports task loops, task priorities, doacross loops, and hints for locks. Offloading now supports asynchronous execution and dependencies to host execution.

© Michael Klemm, OpenMP ARB
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")

- **User-defined scheduling:**

  ```
  !$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)`
   - 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
   - synchronization overhead
   - default chunk value is 1
Guided scheduling

- **Size of chunks in dynamic schedule**
  - too small $\rightarrow$ large overhead
  - too large $\rightarrow$ 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: $\rightarrow$ 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.
OpenMP Scheduling of simple for loops

OMP_SCHEDULE=static

OMP_SCHEDULE=static,10

OMP_SCHEDULE=dynamic,10
Deferred scheduling

- Decided at run time:

  - `auto`: automatic scheduling
    - Programmer gives implementation the freedom to use any possible mapping.

  - `runtime`:
    - Schedule is one of the above or the previous two slides
    - 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"
  - `export OMP_NUM_THREADS=4
  - `./myprog.exe`

- Call to API routine:

```c
!$OMP do schedule(runtime)
call omp_set_schedule(&omp_sched_dynamic,4)
!$OMP parallel
!$OMP do schedule(runtime)
do
  ...
end do
!$OMP end do
...```
Final remarks on scheduling

- There are always implementation-dependent aspects
  - e.g. Intel’s extended OpenMP environment variables: KMP_*

- An implementation may add its own scheduling algorithms
  - code using specific scheduling may be at a disadvantage
  - **recommendation**: Allow changing of schedule during execution

- If runtime scheduling is chosen and OMP_SCHEDULE not set
  - start with implementation-defined setting
Collapsing loop nests

- Flatten nested loops into a single iteration space

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

<table>
<thead>
<tr>
<th>J</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

- 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**

```c
!$omp parallel
!$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
```

- **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
Explicit barrier synchronization

- Example: reduction with nowait

```c
!$omp parallel
!$omp do reduction(+:tsum)
    do k=1, kmax
        tsum = tsum + foo(a, b, c)
    end do
!$omp end do nowait
: ! code not involving tsum
!$omp barrier
    ... = tsum ...
!$omp end parallel
```

- Barrier construct is a stand-alone directive
- potential performance improvement if `foo()` causes load imbalance
- barrier synchronizes all threads
  → subsequent accesses to `tsum` are safe (preceding accesses are unsafe)
- each barrier must be encountered by all threads in the team or by non at all.
Parallel sections

- **Non-iterative work-sharing construct**
  - distribute a set of structured blocks
  ```ompl
  !$OMP sections
  !$OMP section : ! code block 1
  !$OMP section : ! code block 2
  ...
  !$OMP end sections
  ```
  - each block executed exactly once by one of the threads in team

- **Allowed clauses on sections:**
  - private, first/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

• **one thread only** executes enclosed code block
• all other threads wait until block completes execution
• allowed clauses: `private`, `first/copyprivate`, `nowait`

```
real :: s

s = ...
$omp parallel private(s)

$omp single :
! work
s = ...
$omp end single &
$omp copyprivate(s)
... = ... + s
$omp end parallel
```

```
fork:
T0    T1    T2    T3

persists
(inaccessible)

join

• **copyprivate** and **nowait** clauses: appear on `end single` in Fortran, on `single` in C/C++
• use for updates of shared entities, but ...
• **single** – really a worksharing directive?

© 2010-19 LRZ/RRZE
Combining **single** with **nowait**

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

```plaintext
: 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
```
Parallel array processing

- **Fortran only:**
  ```fortran
  real :: a(n), ...  
  real :: s  
  !$OMP parallel  
  !$OMP workshare  
  a = b + c  
  d = b * c  
  !$OMP end workshare  
  !$OMP end parallel
  ```

- **Division of enclosed code block into units of work**
  - units are executed in parallel
  - array expressions, elemental functions: each element a unit of work
  - Only clause: nowait

- **Structured block must consist of one or more of:**
  - Fortran array/scalar assignments
  - FORALL statements and constructs
  - WHERE statements and constructs
  - OpenMP atomic, critical and parallel constructs

- **Function calls must be to elemental functions**
  - [Reminder: Elemental functions are defined as scalar operators, with a single scalar argument and a scalar return value, but they may be invoked with arrays as actual arguments in which case the function will be applied element-wise, with a conforming array return value.]

- **How is the work divided up?**
  - implementation defined
  - may be (and often is) **single-threaded**
  → problem with performance portability
Further properties of **workshare**

- **Unit of work:**
  - may be defined by an OpenMP construct:
  - critical region, (nested) parallel region

- **Example:**
  - updates on shared variables performed in parallel
  - if no dependencies occur
  - else: implementation must insert synchronization points (this makes a full implementation a difficult task)

```c
!$omp workshare

!$omp critical
x(:) = x + a
!$omp end critical

!$omp critical
y(:) = y + b
!$omp end critical

!$omp critical
z(:) = z + y
!$omp end critical

!$omp end workshare
```
Combining regions and work sharing

- **Example:**

```c
!$omp parallel do
   do k=1, kmax
     :
   end do
!$omp end parallel do
```

- is equivalent to

```c
!omp parallel
!omp do
   do k=1, kmax
     :
   end do
!omp end do
!omp end parallel
```

- **Applies to most work-sharing constructs**
  - do/for
  - workshare
  - sections

- **Notes:**
  - clauses for work-sharing constructs can appear on combined construct
  - the reverse is not true
    - `shared` can only appear in a parallel region
  - clauses on a work-sharing construct only apply for the specific construct block
Synchronization and its issues

Memory model
Additional directives
Performance issues
User-defined synchronization
Why do we need synchronization?

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

- two threads execute
  \[ a = a + 1 \]
  in same parallel region → race condition
Following results could be obtained on each thread

- a after completion of statement:
  - Thread0
    - 1
    - 1
    - 2
  - Thread1
    - 1
    - 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 thread **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

  ```c
  !$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 (→ potentially slower)
  - which are **accessible** in the scope of the FLUSH directive
- **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
Ordering properties of flush operation

Incorrect example (pseudo-code)

<table>
<thead>
<tr>
<th>THREAD 1</th>
<th>THREAD 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic(b=1)</td>
<td>atomic(a=1)</td>
</tr>
<tr>
<td>flush(b)</td>
<td>flush(a)</td>
</tr>
<tr>
<td>flush(a)</td>
<td>flush(b)</td>
</tr>
<tr>
<td>atomic(tmp=a)</td>
<td>atomic(tmp=b)</td>
</tr>
<tr>
<td>if (tmp=0) then</td>
<td>if (tmp=0) then</td>
</tr>
<tr>
<td>protected section</td>
<td>protected section</td>
</tr>
<tr>
<td>end if</td>
<td>end if</td>
</tr>
</tbody>
</table>

Correct example (pseudo-code)

<table>
<thead>
<tr>
<th>THREAD 1</th>
<th>THREAD 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic(b=1)</td>
<td>atomic(a=1)</td>
</tr>
<tr>
<td>flush(a,b)</td>
<td>flush(a,b)</td>
</tr>
<tr>
<td>atomic(tmp=a)</td>
<td>atomic(tmp=b)</td>
</tr>
<tr>
<td>if (tmp=0) then</td>
<td>if (tmp=0) then</td>
</tr>
<tr>
<td>protected section</td>
<td>protected section</td>
</tr>
<tr>
<td>end if</td>
<td>end if</td>
</tr>
</tbody>
</table>

compiler could move flush on b to a position completely after the protected section

compiler is prohibited to move the flush at all
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)
  - but only one at a time within code block → mutual exclusion
  - atomic: code block must be a single line update of a scalar entity of intrinsic type with an intrinsic operation

Fortran:

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

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

C/C++:

```c
#pragma omp critical
    { block }

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

Unary operator also allowed
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, they may be in trouble:

```
!$omp parallel
  :
!$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**
  - **a)** same shared variable
    - **thread 0**
      ```
      subroutine foo()
      !$omp critical
      x = x + y
      !$omp end critical
      
      subroutine bar()
      !$omp critical
      x = x + z
      !$omp end critical
      ```
    - critical region is **global** → OK
  - **b)** 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 → unnessecary 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

- 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.

**Fortran:**

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

**C/C++:**

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

© 2010-19 LRZ/RRZE
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 !$OMP do
- only effective if code is executed in parallel
- only one ordered region per loop
- execution scheme:

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

orders & sequentializes
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
```

- **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
```
Mutual exclusion with locks

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

- In the following, `var` is an `integer(omp_lock_kind)`
- OpenMP lock variables must be only accessed by the lock routines described on the following slides.

- mutual exclusion bound to objects \(\rightarrow\) more flexible than critical regions
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.

A task that sets the lock, can unset the lock, returning it to the unlocked stage.

2 types of locks are supported:

- simple locks
- nestable locks
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)
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 dereference / after definition

Now, flushes are implied by invocation of the lock routines
Example for using locks

- **„Simplest possible“**
  - blocking and non-blocking

```fortran
use omp_lib
integer(omp_lock_kind) :: lock
got_it
!
! starts in unlocked state
!$omp parallel
call omp_set_lock(lock)
! play with red balls
call omp_unset_lock(lock)

do
  got_it = omp_test_lock(lock)
  if (got_it) exit
! work unrelated to red balls
end do
! play with red balls
!$omp end parallel
call omp_destroy_lock(lock)
```

- **Locks are an expensive synchronization mechanism**
  - cost of locking is similar to a full barrier (time spent in the thread successfully acquiring the lock)
  - all threads (not just the current team) must be accounted for

- **Disallowed usage patterns**
  e.g., a thread may not attempt to re-lock a lock it already has acquired

- like **critical**, but more flexible

- loop until lock is received
Nestable locks:

- replace `omp_*_lock(var)` by `omp_*_nest_lock(var)`
- task owning a nestable lock may re-lock it multiple times
  
  put differently:
  
  a nestable lock is available if it is either unlocked or
  
  it is owned by the task executing `omp_get_nest_lock(var)` or `omp_test_nest_lock(var)`

- re-locking increments nest count
- releasing the lock decrements nest count
- lock is unlocked once nest count is zero
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
  ```
Binding of directives
Global variables

Notes for library writers
Binding of Directives

Which parallel region does a directive refer to?

→ Binding specified for every directive in the standard.

- **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 **tasks** (see later) not allowed
- **ordered:** binds to innermost enclosing loop region
- **taskwait:** binds to innermost enclosing task region
- **atomic, critical:** mutual exclusion applies for all threads, not just current team
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

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

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

Not allowed:

OMP do nested within a OMP do
Scoping for global variables: the threadprivate directive

- How to deal with global data? (e.g. static in C and common blocks in Fortran)
- By default, global data is shared
- Effect of the threadprivate directive is that the named global-lifetime objects are replicated, so each thread has its own copy.

**C:**
- `#pragma omp threadprivate(list)`
- List is a comma-separated list of file-scope, namespace-scope, or static block-scope variables that do not have incomplete types

**Fortran:**
- `!$omp threadprivate(list)`
- List is a comma-separated list of named variables and named common blocks. Common block names must appear between slashes.

E.g. an array type whose dimension you have not yet specified.
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
    ```

    (works similar to `firstprivate`)

- **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 violations of the above three rules are dealt with in this course
Tasking

Work sharing for irregular problems, recursive problems and information structures

Acknowledgements: L. Meadows/T. Mattson (Intel)
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**
  - some additional features and improved Fortran support added in 3.1
- 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**.
The task construct

- **Fortran syntax**
  
  ```fortran
  !$omp task [clause,...]  
  structured-block  
  !$omp end task
  ```

- **C/C++ syntax**

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

- "Clause" is one of
  - `if` (scalar-expr)
    (logical in Fortran)
  - `private`, `firstprivate`, `shared`
    (each with a list of variables as argument)

- **Clause**
  - `untied`
  - `default`
    `(shared|none|private|firstprivate)`
    (the latter two only in Fortran)
  - `mergeable`
  - `final(scalar-logical)`

- **Structured block**
  - the usual restraints already known (single entry and single exit point) apply

- **Binding**
  - a task binds to the innermost enclosing parallel region
Example: Processing a linked list

```c
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:

- Typical task generation loop:

```c
#include <omp.h>

int main() {
    list *head = ...;

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

    return 0;
}
```
Example: Processing a linked list

```c
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()**
Example: Fibonacci numbers

```c
int fib(int n){
    int i, j;
    if (n<2)
        return n;
    else {
        #pragma omp task shared(i)
        i=fib(n-1);
        #pragma omp task shared(j)
        j=fib(n-2)
        #pragma omp taskwait
        return i+j;
    }
}
```

- **Features of this example:**
  - should be called from a parallel region for the different specified tasks to be executed in parallel
  - only one thread of a parallel region should call fib() (unless multiple concurrent Fibonacci computations are desired.)
  - All threads of the team will participate in executing the tasks generated.
Data scoping in task regions

- **Implicit determination:**
  - if entity is *shared* in all lexically enclosing constructs, it is *shared* within the task region
  - if it is a serial loop variable, it is *private* – this also applies to implied-do loops
  - else it is *firstprivate*

- **Change by**
  - using a *default* clause
  - using *private*, *firstprivate*, or *shared* clauses on individual entities

- **Shared entities in a task region**
  - may become invalid while some tasks are unfinished → programmer must add synchronization to prevent this
The **if** clause

- **When if argument is false** –
  - task body is executed immediately by encountering thread (an "undeferred task"
  - but otherwise semantics are the same (data environment, synchronization) as for a "deferred" task

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

- **User-directed optimization:**
  - avoid overhead for deferring small task
  - cache locality / memory affinity may be lost by doing so
Using the taskwait directive

- suspends until immediate child tasks of current task complete (but not their descendants)

Syntax:

```c
conew
#pragma omp task
{ process_tree(root->left); } 
#endif
if (root->right) {
#pragma omp task
{ process_tree(root->right); }
#endif
#pragma omp taskwait
do_work(root->data);
```
Task switching

- It is allowed for a thread to
  - suspend a task during execution
  - start (or resume) execution of another task (assigned to the same team)
  - resume original task later

- Pre-condition:
  - a **task scheduling point** is reached

- Example from previous slide:
  - the **taskwait** directive implies a task scheduling point

- Further scheduling points:
  - immediately after generation of an explicit **task**
  - after last instruction of a **task** region
  - at a **taskyield** directive (new in OpenMP 3.1)
  - in implicit and explicit barrier regions
  - implementation defined in **untied** (see later) tasks

- Another example:
  - very many tasks are generated → implementation can suspend generation of tasks and start processing the existing queue
Example: Dealing with large number of tasks

```
#define LARGE_NUMBER 10000000
double item[LARGE_NUMBER]
extern void process(double);

int main() {
#pragma omp parallel
{
#pragma omp single
{
    int i;
    for(i=0;i<=LARGE_NUMBER;i++)
        #pragma omp task
        process(item[i]);
}
}
```

### Features of this example:
- generates a large number of tasks with one thread and executes them with the threads in the team
- implementation may reach its limit on unassigned tasks
- if it does, the implementation is allowed to cause the thread executing the task generating loop to suspend its task at the scheduling point and start executing unassigned tasks.
- once the number of unassigned tasks is sufficiently low, the thread may resume executing of the task generating loop.
Thread switching

- **Default behaviour:**
  - a task assigned to a thread must be (eventually) completed by that thread → task is **tied** to the thread

- **Change this via the `untied` clause**
  - execution of task block may change to another thread of the team at any task scheduling point
  - implementation may add task scheduling points beyond those previously defined (outside programmer's control!)

- **Deployment of untied tasks**
  - starvation scenarios: running out of tasks while generating thread is still working on something

- **Dangers:**
  - more care required (compared with tied tasks) wrt. scoping and synchronization

```c
#pragma omp task untied
structured-block
```
Final and mergeable tasks

Final tasks
- use a `final` clause with a condition
- always undeferred, i.e. executed immediately by the encountering thread
- reducing the overhead of placing tasks in the “task pool”
- all tasks created inside task region are also final (different from an `if` clause)
- inside a task block, a `omp_in_final()` can be used to check whether the task is final

Merged tasks
- using a `mergeable` clause may create a merged task if it is undeferred or final
- a merged task has the same data environment as its creating task region

Final and/or mergeable
- can be used for optimization purposes
- optimize wind-down phase of a recursive algorithm
Final remarks

- **Use of threadprivate data:**
  - value of threadprivate variables cannot be assumed to be unchanged across a task scheduling point. Can be modified by another task executed by the same thread.

- **Tasks and locks:**
  - if a lock is held **across** a task scheduling point, interleaved code trying to acquire (with the same thread) it may cause **deadlock**
  - this especially is a problem for untied tasks

- **Tasks and critical regions:**
  - similar issue if suspension of a task happens inside a critical region and the same thread tries to access a critical region in another scheduled task

- **Later in this course:**
  - treat some of the correctness and performance issues that can occur
OpenMP 4.x Offloading

Acknowledgements:
Simon McIntosh-Smith (University of Bristol)
Tom Scogland and Bronis R. de Supinski (LLNL)
Execution Model before OpenMP 4.x
OpenMP 4.x Execution Model

- Create and destroy threads,
- create and destroy leagues of thread teams,
- assign / distribute work (tasks) to threads and devices,
- specify which data is shared and which is private,
- specify which data must be available to the device,
- coordinate thread access to shared data.

Execution model after OpenMP 4.x executing one team:
OpenMP 4.x Execution Model

- Create and destroy threads,
- create and destroy leagues of thread teams,
- assign / distribute work (tasks) to threads and devices,
- specify which data is shared and which is private,
- specify which data must be available to the device,
- coordinate thread access to shared data.

Execution model after OpenMP 4.x executing multiple teams:
OpenMP Device Constructs

- Execute code on a target device
  - `omp target [clause[[], clause],…]` structured-block
  - `omp declare target [function-definitions-or-declarations]

- Manage the device data environment
  - `map ([map-type:] list) // map clause`
    - `map-type := alloc | tofrom | to | from`
  - `omp target data [clause[[], clause],…]` structured-block
  - `omp target update [clause[[], clause],…]`
  - `omp declare target [variable-definitions-or-declarations]
  - `omp target enter / exit data [clause[[], clause],…]` (new: OpenMP 4.5)

- Workshare for acceleration
  - `omp teams [clause[[], clause],…]` structured-block
  - `omp distribute [clause[[], clause],…]` for-loops
Offloading Computation

- Use target construct to
  - Transfer control from the host to the target device
  - Map variables between the host and target device data environments
- Host thread waits until offloaded region completed
- Use nowait for asynchronous execution

```c
#pragma omp target map(to:b,c,d) map(from:a)
{
    #pragma omp parallel for
    for (i=0; i<count; i++) {
        a[i] = b[i] * c + d;
    }
}
```
Target Construct

- Map variables to a **device data environment** and **execute** the construct on that device.

```
#pragma omp target [clause[ [,] clause] ... ] new-line structured-block
```

where **clause** is one of the following:

- `if([ target :] scalar-expression)`
- `device(integer-expression)`
- `private(list)`
- `firstprivate(list)`
- `map([[map-type-modifier[,]] map-type: ] list)`
- `is_device_ptr(list)`
- `defaultmap(tofrom:scalar)`
- `nowait`
- `depend(dependence-type: list)`
Data mapping

map Clause

```c
extern void init(float*, float*, int);
extern void output(float*, int);

void vec_mult(float *p, float *v1, float *v2, int N)
{
    int i;
    init(v1, v2, N);
    #pragma omp target map(to:v1[0:N], v2[0:N]) map(from:p[0:N])
    #pragma omp parallel for
    for (i=0; i<N; i++)
        p[i] = v1[i] * v2[i];
    output(p, N);
}

module mults
contains
subroutine vec_mult(p,v1,v2,N)
    real dimension(*) :: p, v1, v2
    integer :: N,i
    call init(v1, v2, N)
!omp target map(to: v1(i:N), v2(i:N)) map(from:p(i:N))
!omp parallel do
    do i=1,N
        p(i) = v1(i) * v2(i)
    end do
!omp end target
    call output(p, N)
end subroutine
end module
```

- On entry to the target region:
  - Allocate corresponding variables v1, v2, and p in the device data environment.
  - Assign the corresponding variables v1 and v2 the value of their respective original variables.
  - The corresponding variable p is undefined.

- On exit from the target region:
  - Assign the original variable p the value of its corresponding variable.
  - The original variables v1 and v2 are undefined.
  - Remove the corresponding variables v1, v2, and p from the device data environment.
Teams construct

- The teams construct creates a league of thread teams and the master thread of each team executes the region.
- `#pragma omp teams [clause[ [,] clause] ... ] new-line structured-block`
  - where `clause` is one of the following:
    - `num_teams(integer-expression)`
    - `thread_limit(integer-expression)`
    - `default(shared | none)`
    - `private(list)`
    - `firstprivate(list)`
    - `shared(list)`
    - `reduction(reduction-identifier : list)`

The teams construct creates a league of thread teams
- The master thread of each team executes the `teams` region
- The (max.) number of teams is specified by the `num_teams` clause
- Each team executes with (max.) `thread_limit` threads
- Threads in different teams cannot synchronize with each other
The distribute construct specifies that the iterations of one or more loops will be executed by the thread teams in the context of their implicit tasks. The iterations are distributed across the master threads of all teams that execute the teams region to which the distribute region binds.

```
#pragma omp distribute [clause[ ,] clause] ...
for-loops
```

Where `clause` is one of the following:

- `private(list)`
- `firstprivate(list)`
- `lastprivate(list)`
- `collapse(n)`
- `dist_schedule(kind[, chunk_size])`
Composite constructs and shortcuts in OpenMP 4.5

- 2.10.9  omp distribute simd
- 2.10.10 omp distribute parallel for
- 2.10.11 omp distribute parallel for simd
- 2.11.5  omp target parallel
- 2.11.6  omp target parallel for
- 2.11.7  omp target parallel for simd
- 2.11.8  omp target simd
- 2.11.9  omp target teams
- 2.11.10 omp teams distribute
- 2.11.11 omp teams distribute simd
- 2.11.12 omp target teams distribute
- 2.11.13 omp target teams distribute simd
- 2.11.14 omp teams distribute parallel for
- 2.11.15 omp target teams distribute parallel for
- 2.11.16 omp teams distribute parallel for simd
- 2.11.17 omp target teams distribute parallel for simd
Composite constructs and shortcuts

- **omp distribute**
  - **omp distribute simd**
  - **omp distribute parallel for**
  - **omp distribute parallel for simd**

  Iterations distributed across the master threads of all teams in a teams region
  dito + executed concurrently using SIMD instructions
  executed in parallel by multiple threads that are members of multiple teams
  dito + executed concurrently using SIMD instructions

- **omp teams**
  - **omp teams distribute**
  - **omp teams distribute simd**
  - **omp teams distribute parallel for**
  - **omp teams distribute parallel for simd**

  creates a league of thread teams and the master thread of each team executes the region

- **omp target**
  - **omp target simd**
  - **omp target parallel**
  - **omp target parallel for**
  - **omp target parallel for simd**

  map variables to a device data environment and execute the construct on that device

- **omp target teams**
  - **omp target teams distribute**
  - **omp target teams distribute simd**
  - **omp target teams distribute parallel for**
  - **omp target teams distribute parallel for simd**
Device Runtime Support

- Runtime support routines
  - `void omp_set_default_device(int dev_num )`
  - `int omp_get_default_device(void)`
  - `int omp_get_num_devices(void);`
  - `int omp_get_num_teams(void)`
  - `int omp_get_team_num(void);`
  - `int omp_is_initial_device(void);`
  - `int omp_get_initial_device(void);`
  - 7 Memory routines: Added in OpenMP 4.5 to allow explicit allocation, deallocation, memory transfers and memory associations

- Environment variable
  - Control default device through `OMP_DEFAULT_DEVICE`
  - Control default contention group size `OMP_THREAD_LIMIT`
Current Compiler Support

- Quite a few implementations have emerged:
  - **Intel** began support for OpenMP 4.0 targeting the Xeon Phi coprocessors in 2013.
  - **Cray** provided the first vendor implementation targeting NVIDIA GPUs in late 2015. Now supports OpenMP 4.0, and a subset of OpenMP 4.5.
  - **IBM** has recently completed a compiler implementation using Clang, that supports OpenMP 4.5. This is being introduced into the Clang main trunk.
  - **GCC 6.1** introduced support for OpenMP 4.5, and can target Intel Xeon Phi, or HSA enabled AMD GPUs.

- Levels of maturity are quite mixed but improved a lot in the last year!
Other new features in OpenMP 4.x

- Added the concept of **SIMD parallelism** and associated constructs *(simd construct (#pragma omp simd), declare simd construct, parallel loop SIMD Construct (#pragma omp parallel for simd))*

- The **declare reduction construct** is added to allow user defined reductions.

- The **proc_bind** clause and the **OMP_PLACES** environment variable were added to support **thread affinity policies**.
New features in OpenMP 5.0

- Adds memory allocators to support complex memory hierarchies
- User defined mappers provide deep copy support for map clauses
- Supports better control of device usage and specialization for devices
  - Can require unified shared memory
  - Can use functions specialized for a type of device
- Adds concurrent construct to support compiler optimization
- Well-defined interfaces for (debugging) tools
- Adds C11, C++11 and C++14 as normative base languages
- Expands task dependency mechanism for greater flexibility and control
- Release/acquire semantics added to memory model
OpenMP has a well-defined roadmap:
- 5-year cadence for major releases
- One minor release in between
- (At least) one Technical Report (TR) with feature previews in every year
The mission of the OpenMP ARB (Architecture Review Board) is to standardize directive-based multi-language **high-level parallelism** that is **performant, productive and portable**.
OpenMP Book

USING OPENMP – THE NEXT STEP
Affinity, Accelerators, Tasking, and SIMD

Ruud van der Pas, Eric Stotzer, and Christian Terboven

Covers all of OpenMP 4.5