Programming OpenMP

Christian Terboven
Michael Klemm
Agenda (in total 7 Sessions)

- **Session 1**: OpenMP Introduction
- **Session 2**: Tasking
  - Review of Session 1 / homework assignments
  - Tasking Motivation
  - Task Model in OpenMP
  - Scoping
  - Taskloop
  - Dependencies
  - Cut-off strategies
  - Homework assignments 😊
- **Session 3**: Optimization for NUMA and SIMD
- **Session 4**: What Could Possibly Go Wrong Using OpenMP
- **Session 5**: Introduction to Offloading with OpenMP
- **Session 6**: Advanced OpenMP Offloading Topics
- **Session 7**: Selected / Remaining Topics
Programming OpenMP

Review

Christian Terboven
Michael Klemm
Questions?
Solution of Homework Assignments
Example: Hello

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

int main()
{
    #pragma omp parallel
    {
        int thread_num = omp_get_thread_num();
        int num_threads = omp_get_num_threads();

        printf("Hello from thread %d of %d.\n", thread_num, num_threads);
    }

    return 0;
}
```
Example: Pi

double CalcPi(int n)
{
    const double fH = 1.0 / (double) n;
    double fSum = 0.0;
    double fX;
    int i;

    #pragma omp parallel for private(fX) reduction(+:fSum)
    for (i = 0; i < n; i += 1)
    {
        fX = fH * ((double)i + 0.5);
        fSum += f(fX);
    }
    return fH * fSum;
}
Example: Jacobi / 1

```c
167 /***********************************************************/
168 * Initializes data
169 * Assumes exact solution is u(x,y) = (1-x^2)*(1-y^2)
170 *
171 **************************************************************/
void initialize(
    int n,
    int m,
    double alpha,
    double *dx,
    double *dy,
    double *u,
    double *f)
{
    int i, j, xx, yy;

    *dx = 2.0 / (n-1);
    *dy = 2.0 / (m-1);

    /* Initialize initial condition and RHS */
    #pragma omp parallel for private(i, xx, yy) // or collapse(2) instead of private(i)
    for (j=0; j<m; j++){
        for (i=0; i<n; i++){
            xx = -1.0 + *dx * (i-1);
            yy = -1.0 + *dy * (j-1);
            U(j, i) = 0.0;
            F(j, i) = -alpha * (1.0 - xx*xx) * (1.0 - yy*yy)
                      - 2.0 * (1.0 - xx*xx) - 2.0 * (1.0 - yy*yy);
        }
    }
}```
Example: Jacobi / 2

```c
#include <stdio.h>

#define N 100
#define M 100

#define AX 0.5
#define AY 0.5
#define B 1.0
#define OM 0.1

int main()
{
    double U[N][M];

    // Initial guess
    for (int j = 0; j < M; j++)
        for (int i = 0; i < N; i++)
            U[i][j] = 1.0;

    // Jacobi iteration
    double resid = 0.0;
    for (int j = 1; j < M-1; j++)
        for (int i = 1; i < N-1; i++)
        {
            resid += AX * (U[i-1][j] + U[i+1][j]) + AY * (U[i][j-1] + U[i][j+1]) + B * U[i][j] - F(j, i)
            / b;
            U[i][j] = UOLD[i][j] = U[i][j] - resid;
        }

    return 0;
}
```

OpenMP Tutorial
Members of the OpenMP Language Committee
Example: for

```c
double do_some_computation(int i)
{
    double t = 0.0;
    int j;
    for (j = 0; j < i*i; j++)
    {
        t += sin((double)j) * cos((double)j);
    }
    return t;
}

int main(int argc, char* argv[])
{
    const int dimension = 500;
    int i;
    double result = 0.0;
    double t1 = omp_get_wtime();
    #pragma omp parallel for schedule(dynamic) reduction(+:result)
    for (i = 0; i < dimension; i++)
    {
        result += do_some_computation(i);
    }
    double t2 = omp_get_wtime();
```
Example: minmaxreduction

```c
#pragma omp parallel
{
    #pragma omp for reduction(min: dMin) reduction(max: dMax)
    for (i = 0; i < dimension; i++)
    {
        dArray[i] = func(i);
        dMin = fmin(dMin, dArray[i]);
        dMax = fmax(dMax, dArray[i]);
    }
}
```
Programming OpenMP

Tasking Introduction

Christian Terboven
Michael Klemm
Tasking Motivation
Sudoku for Lazy Computer Scientists

- Let's solve Sudoku puzzles with brute multi-core force

<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>15 11</th>
<th>16 14</th>
<th>12</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>11</td>
<td>16 14</td>
<td>12</td>
<td>6</td>
<td>16</td>
</tr>
<tr>
<td>13</td>
<td>9</td>
<td>12</td>
<td>3</td>
<td>16</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>11</td>
<td>15 10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>10</td>
<td>16</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>1</td>
<td>12</td>
<td>9</td>
<td>15</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>16</td>
<td>12</td>
<td>13</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>2</td>
<td>16</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>15</td>
<td>7</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>10</td>
<td>16</td>
<td>12</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>15</td>
<td>9</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>12</td>
<td>13</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>14</td>
<td>16</td>
<td>10</td>
<td>7</td>
<td>6</td>
<td>12</td>
</tr>
</tbody>
</table>

- (1) Search an empty field
- (2) Try all numbers:
  - (2 a) Check Sudoku
    - If invalid: skip
    - If valid: Go to next field
- Wait for completion
Parallel Brute-force Sudoku

- This parallel algorithm finds all valid solutions

<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>15</th>
<th>11</th>
<th>16</th>
<th>8</th>
<th>11</th>
<th>15</th>
<th>14</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>11</td>
<td>16</td>
<td>14</td>
<td>12</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>9</td>
<td>12</td>
<td>3</td>
<td>16</td>
<td>14</td>
<td>15</td>
<td>11</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>11</td>
<td>15</td>
<td>10</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>10</td>
<td>16</td>
<td>2</td>
<td>13</td>
<td>8</td>
<td>9</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>2</td>
<td>3</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>12</td>
<td>9</td>
<td>15</td>
<td>11</td>
<td>10</td>
<td>7</td>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>10</td>
<td>11</td>
<td>6</td>
<td>5</td>
<td>13</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>15</td>
<td>11</td>
<td>16</td>
<td></td>
<td>12</td>
<td>13</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>1</td>
<td>2</td>
<td>16</td>
<td>10</td>
<td></td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>13</td>
<td>7</td>
<td>11</td>
<td>3</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>4</td>
<td>6</td>
<td>1</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>15</td>
<td></td>
<td>16</td>
<td>9</td>
<td>12</td>
<td>13</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>16</td>
<td></td>
<td>11</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>8</td>
<td>12</td>
<td>13</td>
<td>10</td>
<td>11</td>
<td>2</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>10</td>
<td>7</td>
<td>6</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- (1) Search an empty field
  - #pragma omp parallel
  - #pragma omp single
  - such that one task starts the execution of the algorithm

- (2) Try all numbers:
  - (2 a) Check Sudoku
    - If invalid: skip
    - If valid: Go to next field
  - #pragma omp task
  - needs to work on a new copy of the Sudoku board

- Wait for completion
  - #pragma omp taskwait
  - wait for all child tasks
Performance Evaluation

Sudoku on 2x Intel Xeon E5-2650 @2.0 GHz

- Intel C++ 13.1, scatter binding
- speedup: Intel C++ 13.1, scatter binding

Is this the best we can can do?
Tasking Overview
What is a task in OpenMP?

- Tasks are work units whose execution
  - may be deferred or…
  - … can be executed immediately

- Tasks are composed of
  - code to execute, a data environment (initialized at creation time), internal control variables (ICVs)

- Tasks are created…
  - … when reaching a parallel region → implicit tasks are created (per thread)
  - … when encountering a task construct → explicit task is created
  - … when encountering a taskloop construct → explicit tasks per chunk are created
  - … when encountering a target construct → target task is created
Tasking execution model

- Supports unstructured parallelism
  - unbounded loops
    ```c
    while ( <expr> ) {
      ...
    }
    ```
  - recursive functions
    ```c
    void myfunc( <args> )
    {
      ...; myfunc( <newargs> ); ...;
    }
    ```
- Several scenarios are possible:
  - single creator, multiple creators, nested tasks (tasks & WS)
- All threads in the team are candidates to execute tasks

Example (unstructured parallelism)

```c
#pragma omp parallel
#pragma omp masked ! Also: single
while (elem != NULL) {
  #pragma omp task
  compute(elem);
  elem = elem->next;
}
```
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
```

- Where clause is one of:

  - private(list)
  - firstprivate(list)
  - shared(list)
  - default(shared | none)
  - in_reduction(r-id: list)
  - allocate([allocator:] list)
  - detach(event-handler)

<table>
<thead>
<tr>
<th>Data Environment</th>
<th>Cutoff Strategies</th>
</tr>
</thead>
<tbody>
<tr>
<td>if(scalar-expression)</td>
<td></td>
</tr>
<tr>
<td>mergeable</td>
<td></td>
</tr>
<tr>
<td>final(scalar-expression)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Synchronization</th>
</tr>
</thead>
<tbody>
<tr>
<td>depend(dep-type: list)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task Scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td>untied</td>
</tr>
<tr>
<td>priority(priority-value)</td>
</tr>
<tr>
<td>affinity(list)</td>
</tr>
</tbody>
</table>
Task scheduling: tied vs untied tasks

- Tasks are tied by default (when no untied clause present)
  - tied tasks are executed always by the same thread (not necessarily creator)
  - tied tasks may run into performance problems

- Programmers may specify tasks to be untied (relax scheduling)

```c
#pragma omp task untied
{structured-block}
```

  - can potentially switch to any thread (of the team)
  - bad mix with thread based features: thread-id, threadprivate, critical regions...
  - gives the runtime more flexibility to schedule tasks
  - but most of OpenMP implementations doesn’t “honor” untied 😊
Task scheduling: taskyield directive

- Task scheduling points (and the taskyield directive)
  - tasks can be suspended/resumed at TSPs → some additional constraints to avoid deadlock problems
  - implicit scheduling points (creation, synchronization, ...)
  - explicit scheduling point: the taskyield directive

```c
#pragma omp taskyield
```

- Scheduling [tied/untied] tasks: example

```c
#pragma omp parallel
#pragma omp single
{
  #pragma omp task untied
  {
    foo();
    #pragma omp taskyield
    bar()
  }
}
```

```
tied:
```
```
untied:
```
Task synchronization: taskwait directive

- The taskwait directive (shallow task synchronization)
  - It is a stand-alone directive
  - wait on the completion of child tasks of the current task; just direct children, not all descendant tasks;
    - includes an implicit task scheduling point (TSP)

```
#pragma omp taskwait
```

// implicit barrier will wait for C.x
Task synchronization: barrier semantics

- OpenMP barrier (implicit or explicit)
  - All tasks created by any thread of the current team are guaranteed to be completed at barrier exit

```cpp
#pragma omp barrier
```

- And all other implicit barriers at parallel, sections, for, single, etc…
Task synchronization: taskgroup construct

- The taskgroup construct (deep task synchronization)
  - attached to a structured block; completion of all descendants of the current task; TSP at the end

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

- where clause (could only be): reduction(reduction-identifier: list-items)
Data Environment
Explicit data-sharing clauses

Explicit data-sharing clauses (shared, private and firstprivate)

- #pragma omp task shared(a)
  {  
    // Scope of a: shared  
  }

- #pragma omp task private(b)
  {  
    // Scope of b: private  
  }

- #pragma omp task firstprivate(c)
  {  
    // Scope of c: firstprivate  
  }

If **default** clause present, what the clause says

- shared: data which is not explicitly included in any other data sharing clause will be **shared**

- none: compiler will issue an error if the attribute is not explicitly set by the programmer (very useful!!!)

- #pragma omp task default(shared)
  {  
    // Scope of all the references, not explicitly included in any other data sharing clause, and with no pre-determined attribute: shared  
  }

- #pragma omp task default(none)
  {  
    // Compiler will force to specify the scope for every single variable referenced in the context  
  }

  *Hint: Use default(none) to be forced to think about every variable if you do not see clearly.*
Pre-determined data-sharing attributes

- threadprivate variables are threadprivate (1)
- dynamic storage duration objects are shared (malloc, new,...) (2)
- static data members are shared (3)
- variables declared inside the construct
  → static storage duration variables are shared (4)
  → automatic storage duration variables are private (5)
- the loop iteration variable(s)...

```c
int A[SIZE];
#pragma omp threadprivate(A)
// ...
#pragma omp task
{ // A: threadprivate
}
```

```c
int *p;
p = malloc(sizeof(float)*SIZE);
#pragma omp task
{ // *p: shared
}
```

```c
void foo(void){
    static int s = MN;
}
#pragma omp task
{ // s@foo(): shared
}
```
Implicit data-sharing attributes (in-practice)

- Implicit data-sharing rules for the task region
  - the **shared** attribute is lexically inherited
  - in any other case the variable is **firstprivate**

```c
int a = 1;
void foo() {
    int b = 2, c = 3;
    #pragma omp parallel private(b)
    {
        int d = 4;
        #pragma omp task
        {
            int e = 5;
            // Scope of a:
            // Scope of b:
            // Scope of c:
            // Scope of d:
            // Scope of e:
        }
    }
}
```

- Pre-determined rules (could not change)
- Explicit data-sharing clauses (+ default)
- Implicit data-sharing rules

- (in-practice) variable values within the task:
  - value of a: 1
  - value of b: x // undefined (undefined in parallel)
  - value of c: 3
  - value of d: 4
  - value of e: 5
Task reductions (using taskgroup)

- **Reduction operation**
  - perform some forms of recurrence calculations
  - associative and commutative operators

- The (taskgroup) scoping reduction clause

  ```
  #pragma omp taskgroup task_reduction(op: list)
  {structured-block}
  ```

  - Register a new reduction at [1]
  - Computes the final result after [3]

- The (task) in_reduction clause [participating]

  ```
  #pragma omp task in_reduction(op: list)
  {structured-block}
  ```

  - Task participates in a reduction operation [2]
Task reductions (+ modifiers)

Reduction modifiers
- Former reductions clauses have been extended
- task modifier allows to express task reductions
- Registering a new task reduction [1]
- Implicit tasks participate in the reduction [2]
- Compute final result after [4]

The (task) in_reduction clause [participating]

```c
int res = 0;
node_t* node = NULL;
...
#pragma omp parallel reduction(task,+: res)
{
  // [1][2]
  #pragma omp single
  {
    #pragma omp taskgroup
    {
      #pragma omp task in_reduction(+: res)
        firstprivate(node)
      {
        // [3]
        res += node->value;
      }
      node = node->next;
    }
  }
} // [4]
```

- Task participates in a reduction operation [3]
Tasking illustrated
Fibonacci illustrated

Only one Task / Thread enters fib() from main(), it is responsible for creating the two initial work tasks

Taskwait is required, as otherwise x and y would get lost
- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks
- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks
- …
The `taskloop` Construct
Tasking use case: saxpy (taskloop)

```c
for (i = 0; i<SIZE; i+=1) {
}
```

```c
for (i = 0; i<SIZE; i+=TS) {
    UB = SIZE < (i+TS) ? SIZE: i+TS;
    for (ii = i; ii<UB; ii++) {
    }
}
```

#pragma omp parallel
#pragma omp single
for (i = 0; i<SIZE; i+=TS) {
    UB = SIZE < (i+TS) ? SIZE: i+TS;
    #pragma omp task private(ii) firstprivate(i, UB) shared(S, A, B)
    for (ii = i; ii<UB; ii++) {
    }
}

#pragma omp taskloop grainsize(TS)
for (i = 0; i<SIZE; i+=1) {
}
```

- Difficult to determine grain
  - 1 single iteration → to fine
  - whole loop → no parallelism
- Manually transform the code
  - blocking techniques
- Improving programmability
  - OpenMP taskloop
- Hiding the internal details
- Grain size ~ Tile size (TS) → but implementation decides exact grain size
The taskloop Construct

- Task generating construct: decompose a loop into chunks, create a task for each loop chunk

```
#pragma omp taskloop [clause[[], clause]...]
{structured-for-loops}
```

- Where clause is one of:

  - shared(list)
  - private(list)
  - firstprivate(list)
  - lastprivate(list)
  - default(sh | pr | fp | none)
  - reduction(r-id: list)
  - in_reduction(r-id: list)
  - grainsize(grain-size)
  - num_tasks(num-tasks)

```
!$omp taskloop [clause[[], clause]...]
...structured-do-loops...
!$omp end taskloop
```

Cutoff Strategies

- if(scalar-expression)
- final(scalar-expression)
- mergeable

Scheduler (R/H)

- untied
- priority(priority-value)
- collapse(n)
- nogroup
- allocate([allocator:] list)

Miscellaneous

- default(sh | pr | fp | none)
- reduction(r-id: list)
- in_reduction(r-id: list)
- grainsize(grain-size)
- num_tasks(num-tasks)
Worksharing vs. taskloop constructs (1/2)

```fortran
subroutine worksharing
  integer :: x
  integer :: i
  integer, parameter :: T = 16
  integer, parameter :: N = 1024

  x = 0
  !$omp parallel shared(x) num_threads(T)

  !$omp do
    do i = 1,N
      !$omp atomic
      x = x + 1
    !$omp end atomic
  !$omp end do
  !$omp end parallel
  write (*,'(A,I0)') 'x = ', x
end subroutine

subroutine taskloop
  integer :: x
  integer :: i
  integer, parameter :: T = 16
  integer, parameter :: N = 1024

  x = 0
  !$omp parallel shared(x) num_threads(T)

  !$omp taskloop
    do i = 1,N
      !$omp atomic
      x = x + 1
    !$omp end atomic
  !$omp end taskloop
  !$omp end parallel
  write (*,'(A,I0)') 'x = ', x
end subroutine
```

Result: \( x = 1024 \)

Result: \( x = 16384 \)
Worksharing vs. taskloop constructs (2/2)

**subroutine worksharing**

```fortran
integer :: x
integer :: i
integer, parameter :: T = 16
integer, parameter :: N = 1024

x = 0
$omp parallel shared(x) num_threads(T)

$omp do
do i = 1,N
$omp atomic
    x = x + 1
$omp end atomic
doi
$omp end do
$omp end parallel

write (*,'(A,I0)') 'x = ', x
end subroutine
```

**subroutine taskloop**

```fortran
integer :: x
integer :: i
integer, parameter :: T = 16
integer, parameter :: N = 1024

x = 0
$omp parallel shared(x) num_threads(T)
$omp single
$omp taskloop
do i = 1,N
$omp atomic
    x = x + 1
$omp end atomic
doi
$omp end taskloop
$omp end single
$omp end parallel

write (*,'(A,I0)') 'x = ', x
end subroutine
```

Result: $x = 1024$

Result: $x = 1024$
Taskloop decomposition approaches

- **Clause: grainsize(grain-size)**
  - Chunks have at least grain-size iterations
  - Chunks have maximum 2x grain-size iterations

```c
int TS = 4 * 1024;
#pragma omp taskloop grainsize(TS)
for ( i = 0; i<SIZE; i+=1) {
}
```

- **Clause: num_tasks(num-tasks)**
  - Create num-tasks chunks
  - Each chunk must have at least one iteration

```c
int NT = 4 * omp_get_num_threads();
#pragma omp taskloop num_tasks(NT)
for ( i = 0; i<SIZE; i+=1) {
}
```

- If none of previous clauses is present, the *number of chunks* and the *number of iterations per chunk* is implementation defined

- Additional considerations:
  - The order of the creation of the loop tasks is unspecified
  - Taskloop creates an implicit taskgroup region; `nogroup` → no implicit taskgroup region is created
Collapsing iteration spaces with taskloop

- The collapse clause in the taskloop construct
  
  ```
  #pragma omp taskloop collapse(n)
  {structured-for-loops}
  ```
  
  → Number of loops associated with the taskloop construct (n)
  → Loops are collapsed into one larger iteration space
  → Then divided according to the `grainsize` and `num_tasks`

- Intervening code between any two associated loops
  
  → at least once per iteration of the enclosing loop
  → at most once per iteration of the innermost loop

```c
#pragma omp taskloop collapse(2)
for ( i = 0; i<SX; i+=1) {
  for ( j = 0; i<SY; j+=1) {
    for ( k = 0; i<SZ; k+=1) {
      A[f(i,j,k)]=<expression>;  
    }
  }
}
```

```c
#pragma omp taskloop
for ( ij = 0; i<SX*SY; ij+=1) {
  for ( k = 0; i<SZ; k+=1) {
    i = index_for_i(ij);
    j = index_for_j(ij);
    A[f(i,j,k)]=<expression>;
  }
}
```
Task reductions (using taskloop)

- **Clause: reduction(r-id: list)**
  - It defines the scope of a new reduction
  - All created tasks participate in the reduction
  - It cannot be used with the `nogroup` clause

- **Clause: in_reduction(r-id: list)**
  - Reuse an already defined reduction scope
  - All created tasks participate in the reduction
  - It can be used with the `nogroup*` clause, but it is user responsibility to guarantee result

```c
double dotprod(int n, double *x, double *y) {
    double r = 0.0;
    #pragma omp taskloop reduction(+: r)
    for (i = 0; i < n; i++)
        r += x[i] * y[i];
    return r;
}
```

```c
double dotprod(int n, double *x, double *y) {
    double r = 0.0;
    #pragma omp taskgroup task_reduction(+: r)
    {
        #pragma omp taskloop in_reduction(+: r)
        for (i = 0; i < n; i++)
            r += x[i] * y[i];
    }
    return r;
}
```
Composite construct: taskloop simd

- Task generating construct: decompose a loop into chunks, create a task for each loop chunk
- Each generated task will apply (internally) SIMD to each loop chunk

→ C/C++ syntax:

```c
#pragma omp taskloop simd [clause[, clause]…]
{structured-for-loops}
```

→ Fortran syntax:

```fortran
 !$omp taskloop simd [clause[, clause]…]
 ...structured-do-loops...
 !$omp end taskloop
```

- Where clause is any of the clauses accepted by `taskloop` or `simd` directives
Improving Tasking Performance: Task dependences
Motivation

Task dependences as a way to define task-execution constraints

```c
int x = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task
    std::cout << x << std::endl;
    #pragma omp taskwait
    #pragma omp task
    x++;
}
```

**OpenMP 3.1**

```c
int x = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(in: x)
    std::cout << x << std::endl;
    #pragma omp task depend(inout: x)
    x++;
}
```

**OpenMP 4.0**

![Diagram showing task creation and execution times for OpenMP 3.1 and 4.0]
**Motivation**

- Task dependences as a way to define task-execution constraints

```cpp
int x = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task
    std::cout << x << std::endl;
    #pragma omp taskwait
    #pragma omp task
    x++;
}
```

```
int x = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(in: x)
    std::cout << x << std::endl;
    #pragma omp taskwait
    #pragma omp task depend(inout: x)
    x++;
}
```

Task dependences can help us to remove “strong” synchronizations, increasing the look ahead and, frequently, the parallelism!!!!

OpenMP 3.1

```
int x = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task
    std::cout << x << std::endl;
    #pragma omp taskwait
    #pragma omp task
    x++;
}
```

OpenMP 4.0

```
int x = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(in: x)
    std::cout << x << std::endl;
    #pragma omp taskwait
    #pragma omp task depend(inout: x)
    x++;
}
```
Motivation: Cholesky factorization

void cholesky(int ts, int nt, double* a[nt][nt]) {
  for (int k = 0; k < nt; k++) {
    // Diagonal Block factorization
    potrf(a[k][k], ts, ts);
    // Triangular systems
    for (int i = k + 1; i < nt; i++) {
      #pragma omp task
      trsm(a[k][k], a[k][i], ts, ts);
    }
    #pragma omp taskwait
    // Update trailing matrix
    for (int i = k + 1; i < nt; i++) {
      for (int j = k + 1; j < i; j++) {
        #pragma omp task
        dgemm(a[k][i], a[k][j], a[j][j], ts, ts);
      }
      #pragma omp task
      syrk(a[k][i], a[i][i], ts, ts);
    }
    #pragma omp taskwait
  }
}

OpenMP 3.1

void cholesky(int ts, int nt, double* a[nt][nt]) {
  for (int k = 0; k < nt; k++) {
    // Diagonal Block factorization
    #pragma omp task depend(inout: a[k][k])
    potrf(a[k][k], ts, ts);
    // Triangular systems
    for (int i = k + 1; i < nt; i++) {
      #pragma omp task depend(in: a[k][k])
      depend(inout: a[k][i])
      trsm(a[k][k], a[k][i], ts, ts);
    }
    #pragma omp taskwait
    // Update trailing matrix
    for (int i = k + 1; i < nt; i++) {
      for (int j = k + 1; j < i; j++) {
        #pragma omp task depend(inout: a[j][i])
        depend(in: a[k][i], a[k][j])
        dgemm(a[k][i], a[k][j], a[j][i], ts, ts);
      }
      #pragma omp task depend(inout: a[i][i])
      depend(in: a[k][i])
      syrk(a[k][i], a[i][i], ts, ts);
    }
    #pragma omp taskwait
  }
}

OpenMP 4.0
Motivation: Cholesky factorization

```c
void cholesky(int ts, int nt, double* a[nt][nt]) {
    for (int k = 0; k < nt; k++) {
        // Diagonal Block factorization
        potrf(a[k][k], ts, ts);
        // Triangular systems
        for (int i = k + 1; i < nt; i++) {
            #pragma omp task
            trsm(a[k][k], a[k][i], ts, ts);
            #pragma omp taskwait
        }
        #pragma omp taskwait
        // Update trailing matrix
        for (int i = k + 1; i < nt; i++) {
            for (int j = k + 1; j < nt; j++) {
                #pragma omp task
                dgemm(a[k][i], a[k][j], a[j][i], ts, ts);
                #pragma omp task
                syrk(a[k][i], a[i][i], ts, ts);
            }
            #pragma omp taskwait
        }
    }
}
```

Using 2017 Intel compiler
What’s in the spec
What’s in the spec: a bit of history

OpenMP 4.0

- The `depend` clause was added to the `task` construct

OpenMP 4.5

- The `depend` clause was added to the target constructs
- Support to doacross loops

OpenMP 5.0

- lvalue expressions in the `depend` clause
- New dependency type: `mutexinoutset`
- Iterators were added to the `depend` clause
- The `depend` clause was added to the `taskwait` construct
- Dependable objects
What’s in the spec: syntax depend clause

\[\text{depend([depend-modifier,] dependency-type: list-items)}\]

where:

- depend-modifier is used to define iterators
- dependency-type may be: in, out, inout, mutexinout, set and depobj
- A list-item may be:
  - C/C++: A lvalue expr or an array section \(\text{depend(in: x, v[i], p, w[10:10])}\)
  - Fortran: A variable or an array section \(\text{depend(in: x, v(i), w(10:20))}\)
What’s in the spec: `sema depend` clause (1)

- A task cannot be executed until all its predecessor tasks are completed.

- If a task defines an `in` dependence over a list-item,
  - the task will depend on all previously generated sibling tasks that reference that list-item in an `out` or `inout` dependence.

- If a task defines an `out/inout` dependence over list-item,
  - the task will depend on all previously generated sibling tasks that reference that list-item in an `in`, `out` or `inout` dependence.
What’s in the spec: depend clause (1)

- A task cannot be executed until all its predecessor tasks are completed.

- If a task defines an "in" dependence over a variable, the task will depend on all previously generated sibling tasks that reference the variable.

- If a task defines an "out" or "in-out" dependence over a variable, the task will depend on all previously generated sibling tasks that reference the variable.

```c
int x = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(inout: x) //T1
    { ... }

    #pragma omp task depend(in: x) //T2
    { ... }

    #pragma omp task depend(in: x) //T3
    { ... }

    #pragma omp task depend(inout: x) //T4
    { ... }
}
```
What’s in the spec: depend clause (2)

- **New dependency type:** `mutexinoutset`

```c
int x = 0, y = 0, res = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(out: res)  //T0
        res = 0;

    #pragma omp task depend(out: x)   //T1
        long_computation(x);

    #pragma omp task depend(out: y)   //T2
        short_computation(y);

    #pragma omp task depend(in: x)    depend(mutexinoutset;/T8es) //T3
        res += x;

    #pragma omp task depend(in: y)    depend(mutexinoutset;/T8es) //T4
        res += y;

    #pragma omp task depend(in: res)  //T5
        std::cout << res << std::endl;
}
```

1. **inoutset property:** tasks with a `mutexinoutset` dependence create a cloud of tasks (an inout set) that synchronizes with previous & posterior tasks that dependent on the same list item

2. **mutex property:** Tasks inside the inout set can be executed in any order but with mutual exclusion
What’s in the spec: depend clause (4)

- Task dependences are defined among sibling tasks
- List items used in the depend clauses [...] must indicate identical or disjoint storage

```cpp
//test1.cc
int x = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(inout: x) //T1
    {
        #pragma omp task depend(inout: x) //T1.1
        x++;
        #pragma omp taskwait
    }
    #pragma omp task depend(in: x) //T2
    std::cout << x << std::endl;
}
```

```cpp
//test2.cc
int a[100] = {0};
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(inout: a[50:99]) //T1
    compute(/* from */ &a[50], /*elems*/ 50);

    #pragma omp task depend(in: a) //T2
    print(/* from */ a, /* elem */ 100);
}
```
What’s in the spec: depend clause (5)

- Iterators + deps: a way to define a dynamic number of dependences

```c++
std::list<int> list = ...;
int n = list.size();

#pragma omp parallel
#pragma omp single
{
    for (int i = 0; i < n; ++i)
        #pragma omp task depend(out: list[i])   //Px
        compute_elem(list[i]);

    #pragma omp task depend(iterator(j=0:n), in : list[j]) //C
    print elems(list);
}
```

It seems innocent but it’s not:
```c++
depend(out: list.operator[](i))
```

Equivalent to:
```c++
depend(in: list[0], list[1], ..., list[n-1])
```
Use case
Use case: intro to Gauss-seidel

```c
void serial_gauss_seidel(int tsteps, int size, int (*p)[size]) {
    for (int t = 0; t < tsteps; ++t) {
        for (int i = 1; i < size-1; ++i) {
            for (int j = 1; j < size-1; ++j) {
                p[i][j] = 0.25 * (p[i][j-1] + // left
                                p[i][j+1] + // right
                                p[i-1][j] + // top
                                p[i+1][j]); // bottom
            }
        }
    }
}
```

Access pattern analysis

For a specific $t$, $i$ and $j$

Each cell depends on:
- two cells (north & west) that are computed in the current time step, and
- two cells (south & east) that were computed in the previous time step
Use case: Gauss-seidel (2)

```c
void serial_gauss_seidel(int tsteps, int size, int (*p)[size]) {
    for (int t = 0; t < tsteps; ++t) {
        for (int i = 1; i < size-1; ++i) {
            for (int j = 1; j < size-1; ++j) {
                p[i][j] = 0.25 * (p[i][j-1] + // left
                                   p[i][j+1] + // right
                                   p[i-1][j] + // top
                                   p[i+1][j]); // bottom
            }
        }
    }
}
```

1\textsuperscript{st} parallelization strategy

For an specific \( t \), we can exploit the wavefront to obtain parallelism!!
void gauss_seidel(int tsteps, int size, int TS, int (*p)[size]) {
  int NB = size / TS;
  #pragma omp parallel
  for (int t = 0; t < tsteps; ++t) {
    // First NB diagonals
    for (int diag = 0; diag < NB; ++diag) {
      #pragma omp for
      for (int d = 0; d <= diag; ++d) {
        int ii = d;
        int jj = diag - d;
        for (int i = 1+ii*TS; i < ((ii+1)*TS); ++i)
          for (int j = 1+jj*TS; i < ((jj+1)*TS); ++j)
            p[i][j] = 0.25 * (p[i][j-1] + p[i][j+1] +
                             p[i-1][j] + p[i+1][j]);
      }
    }
    // Lasts NB diagonals
    for (int diag = NB-1; diag >= 0; --diag) {
      // Similar code to the previous loop
    }
  }
}
Use case : Gauss-seidel (4)

```c
void serial_gauss_seidel(int tsteps, int size, int (*p)[size]) {
    for (int t = 0; t < tsteps; ++t) {
        for (int i = 1; i < size-1; ++i) {
            for (int j = 1; j < size-1; ++j) {
                p[i][j] = 0.25 * (p[i][j-1] + // left
                                  p[i][j+1] + // right
                                  p[i-1][j] + // top
                                  p[i+1][j]); // bottom
            }
        }
    }
}
```

2\textsuperscript{nd} parallelization strategy

We can exploit the wavefront of multiple time steps to obtain MORE parallelism!!
Use case: Gauss-seidel (5)

```c
void gauss_seidel(int tsteps, int size, int TS, int (*p)[size]) {
    int NB = size / TS;

    #pragma omp parallel
    #pragma omp single
    for (int t = 0; t < tsteps; ++t)
        for (int ii=1; ii < size-1; ii+=TS)
            for (int jj=1; jj < size-1; jj+=TS) {
                #pragma omp task depend(inout: p[ii:TS][jj:TS])
                depend(in: p[ii-TS:TS][jj:TS], p[ii+TS:TS][jj:TS],
                       p[ii:TS][jj-TS:TS], p[ii:TS][jj+TS])
                {
                    for (int i=ii; i<(1+ii)*TS; ++i)
                        for (int j=jj; j<(1+jj)*TS; ++j)
                            p[i][j] = 0.25 * (p[i][j-1] + p[i][j+1] +
                                              p[i-1][j] + p[i+1][j]);
                }
            }
}
```

Q: Why do the input dependences depend on the whole block rather than just a column/row?

inner matrix region

VS
Use case: Gauss-seidel (5)

```c
void gauss_seidel(int tsteps, int size, int TS, int (*p)[size]) {
    int NB = size / TS;
    #pragma omp parallel
    #pragma omp single
    for (int t = 0; t < tsteps; ++t)
        for (int ii=1; ii < size - 1; ii+=TS)
            for (int jj=1; jj < size - 1; jj+=TS) {
                #pragma omp task depend(inout: p[ii:TS][jj:TS])
                depend(in: p[ii-TS:TS][jj:TS], p[ii+TS:TS][jj:TS],
                       p[ii:TS][jj-TS:TS], p[ii:TS][jj:TS])
                {
                    for (int i=ii; i<(1+ii)*TS; ++i)
                        for (int j=jj; j<(1+jj)*TS; ++j)
                            p[i][j] = 0.25 * (p[i][j-1] + p[i][j+1] +
                                  p[i-1][j] + p[i+1][j]);
                }
            }
}
```

Use case: Gauss-seidel (5)

Matrix region

Q: Why do the input dependences depend on the whole block rather than just a column/row?

VS
OpenMP 5.0: (even) more advanced features
Advanced features: deps on `taskwait`

- Adding dependences to the `taskwait` construct
  - Using a `taskwait` construct to explicitly wait for some predecessor tasks
  - Syntactic sugar!

```c
int x = 0, y = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(inout: x)    //T1
    x++;
    #pragma omp task depend(in: y)      //T2
    std::cout << y << std::endl;
    #pragma omp taskwait depend(in: x)
    std::cout << x << std::endl;
}
```
Improving Tasking Performance: Cutoff clauses and strategies
Example: Sudoku revisited
Parallel Brute-force Sudoku

- This parallel algorithm finds all valid solutions

<table>
<thead>
<tr>
<th>6</th>
<th>8</th>
<th>11</th>
<th>15</th>
<th>14</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>11</td>
<td>16</td>
<td>14</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>13</td>
<td>9</td>
<td>12</td>
<td>3</td>
<td>16</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>11</td>
<td>15</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>10</td>
<td>16</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>12</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>1</td>
<td>11</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>15</td>
<td>11</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>2</td>
<td>16</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>15</td>
<td>16</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>16</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>12</td>
<td>13</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>10</td>
<td>7</td>
<td>6</td>
<td>12</td>
</tr>
</tbody>
</table>

- (1) Search an empty field
  - first call contained in a
  #pragma omp parallel
  #pragma omp single
  such that one tasks starts the execution of the algorithm

- (2) Try all numbers:
  - (2 a) Check Sudoku
    - If invalid: skip
    - If valid: Go to next field

- Wait for completion
  - #pragma omp task
    needs to work on a new copy of the Sudoku board
  - #pragma omp taskwait
    wait for all child tasks
Performance Evaluation

Sudoku on 2x Intel Xeon E5-2650 @2.0 GHz

- Runtime [sec] for 16x16
- Speedup: Intel C++ 13.1, scatter binding

Intel C++ 13.1, scatter binding
Performance Analysis

Event-based profiling provides a good overview:

Every thread is executing ~1.3m tasks...

... in ~5.7 seconds.

=> average duration of a task is ~4.4 \( \mu \)s

Tracing provides more details:

Tasks get much smaller down the call-stack.
Performance Analysis

Event-based profiling provides a good overview:

Every thread:

- ... in ~5.7 seconds.
- => average duration of a task is ~4.4 μs

If you have enough parallelism, stop creating more tasks!!

- if-clause, final-clause, mergeable-clause
- natively in your program code

Tracing provides more details:

- Duration: 0.16 sec
- Duration: 0.001 sec
- Duration: 2.2 μs

Tasks get much smaller down the call-stack.
Performance Evaluation (with cutoff)

Sudoku on 2x Intel Xeon E5-2650 @2.0 GHz

- Intel C++ 13.1, scatter binding
- Intel C++ 13.1, scatter binding, cutoff
- speedup: Intel C++ 13.1, scatter binding
- speedup: Intel C++ 13.1, scatter binding, cutoff
The if clause

- **Rule of thumb: the if(expression) clause as a “switch off” mechanism**
  - Allows lightweight implementations of task creation and execution but it reduces the parallelism

- **If the expression of the if clause evaluates to false**
  - The encountering task is suspended
  - The new task is executed immediately (task dependences are respected!!)
  - The encountering task resumes its execution once the new task is completed
  - This is known as **undeferred task**

- **Even if the expression is false, data-sharing clauses are honored**

```c
int foo(int x) {
    printf("entering foo function\n");
    int res = 0;
    #pragma omp task shared(res) if(false)
    {
        res += x;
    }
    printf("leaving foo function\n");
}
```

- Really useful to debug tasking applications!
The final clause

The final(expression) clause

- Nested tasks / recursive applications
- allows to avoid future task creation \(\rightarrow\) reduces overhead but also reduces parallelism

If the expression of the final clause evaluates to true

- The new task is created and executed normally but in its context all tasks will be executed immediately by the same thread (included tasks)

Data-sharing clauses are honored too!

```c
#pragma omp task final(e)
{
    #pragma omp task
    { ... }
    #pragma omp task
    { ... #C.1; #C.2 ... }
    #pragma omp taskwait
}
```
The mergeable clause

- **The mergeable clause**
  - Optimization: get rid of “data-sharing clauses are honored”
  - This optimization can only be applied in `undefined` or `included tasks`

- A Task that is annotated with the mergeable clause is called a **mergeable task**
  - A task that may be a `merged task` if it is an `undeferred task` or an `included task`

- A **merged task** is:
  - A task for which the data environment (inclusive of ICVs) may be the same as that of its generating task region

- A good implementation could execute a merged task without adding any OpenMP-related overhead

Unfortunately, there are no OpenMP commercial implementations taking advantage of `final neither mergeable` =(
Hands-on Exercises
Exercises

- We have implemented a series of small hands-on examples that you can use and play with.
  - Subfolder: Session-2-Tasking/exercises, with instructions in Exercises_OMP_2024.pdf
  - Build:  `make`
  - You can then find the compiled executable to run with the sample Slurm script
  - We use the GCC compiler mostly

- Each hands-on exercise has a folder “solution”
  - It shows the OpenMP directive that we have added
  - You can use it to cheat 😊, or to check if you came up with the same solution
## Exercises: Overview

<table>
<thead>
<tr>
<th>Exercise no.</th>
<th>Exercise name</th>
<th>OpenMP Topic</th>
<th>Day / Order (proposal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Fibonacci</td>
<td>Familiarize yourself with Tasking</td>
<td>Second day</td>
</tr>
<tr>
<td>2</td>
<td>For / Work-Distribution</td>
<td>Task + Taskloop</td>
<td>Second day</td>
</tr>
<tr>
<td>3</td>
<td>Quicksort</td>
<td>Tasking, Cut-off</td>
<td>Second day</td>
</tr>
</tbody>
</table>