OpenMP Tasking, Part 1: Fundamentals

Xavier Teruel
Slides and Exercises

https://tinyurl.com/9sxmvsm2
Outline: tasking overview

- Introduction: definition and motivation
- The task construct, creating a single instance of a task
  - Task scheduling: restrictions and hints
  - Basic synchronization constructs
- Data environment: variables within a task
- Tasking use cases
- The taskloop construct, dividing the loop iteration space in tasks

* These slides are part of the tutorial “Mastering Tasking with OpenMP”; presented at SC and ISC conferences. Authors: Christian Terboven, Michael Klemm, Xavier Teruel, and Bronis R. de Supinski. All of them members of the OpenMP Language Committee.
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, usually by means of parallel / single
  - multiple creators, work-sharing and nested tasks
- All threads in the team are candidates to execute tasks

Example (unstructured parallelism)
```c
#pragma omp parallel
#pragma omp 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:

<table>
<thead>
<tr>
<th>Data Environment</th>
<th>Cutoff Strategies</th>
<th>Synchronization</th>
<th>Task Scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td>private(list)</td>
<td>if(scalar-expression)</td>
<td>depend(dep-type: list)</td>
<td>untied</td>
</tr>
<tr>
<td>firstprivate(list)</td>
<td>mergeable</td>
<td>final(scalar-expression)</td>
<td>priority(priority-value)</td>
</tr>
<tr>
<td>shared(list)</td>
<td>mergeable</td>
<td></td>
<td>priority(priority-value)</td>
</tr>
<tr>
<td>default(shared</td>
<td>none)</td>
<td></td>
<td>priority(priority-value)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>depend(dep-type: list)</td>
<td></td>
<td>priority(priority-value)</td>
</tr>
<tr>
<td>allocate([allocator:] list)</td>
<td></td>
<td></td>
<td>affinity(list)</td>
</tr>
<tr>
<td>detach(event-handler)</td>
<td></td>
<td></td>
<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()
  }
}
```

- untied: 
  ```c
  single
  foo()
  bar()
  ```
- tied: 
  ```c
  single
  foo()
  ```
  ```c
  bar()
  ```
  (default)
Task scheduling: programmer’s hints

- Programmers may specify a priority value when creating a task

```
#pragma omp task priority(pvalue)
{structured-block}
```

→ pvalue: the higher the best (will be scheduled earlier)

→ once a thread becomes idle, gets one of the highest priority tasks

```
#pragma omp parallel
#pragma omp single
{
    for (i = 0; i < SIZE; i++) {
        #pragma omp task priority(1)
        { code_A; }
    }
    #pragma omp task priority(100)
    { code_B; }
    ...
}
```
Task synchronization: taskwait directive

- The taskwait directive (shallow task synchronization)
  - It is a stand-alone directive

```cpp
#pragma omp taskwait
```

- 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 parallel
#pragma omp single
{
    #pragma omp task
    {
        #pragma omp task
        {
            #pragma omp task
            {
                ... 
            }  
            #pragma omp task
            {
                ...  
            }
        }
        #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
  
  ```
  #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

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

→ where clause (could only be): reduction(reduction-identifier: list-items)

```c
#pragma omp parallel
#pragma omp single
{
#pragma omp taskgroup
{
#pragma omp task
{ ... }
#pragma omp task
{ ... #C.1; #C.2; ...}
}
// end of taskgroup
```
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
  }
  
  #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.
  }
  ```

- 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!!!)
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

```
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 (can 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
  ```c
  #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]
  ```c
  #pragma omp task in_reduction(op: list)
  {structured-block}
  ```
  - Task participates in a reduction operation [2]

```c
int res = 0;
node_t* node = NULL;
...
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp taskgroup task_reduction(+: res)
    { // [1]
      while (node) {
        #pragma omp task in_reduction(+: res) \ firstprivate(node)
        { // [2]
          res += node->value;
        }
        node = node->next;
      } // [3]
    }
  }
}...
```
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
#pragma omp task in_reduction(op: list)
{structured-block}
```

- Task participates in a reduction operation [3]

```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]
```
Tasking Use Cases
Tasking use case: Fibonacci (recursion)

```c
int comp_fib_numbers ( int n) {
    int fn1, fn2;
    if ( n == 0 || n == 1 ) return(n);
    #pragma omp task shared(fn1)
    fn1 = comp_fib_numbers(n-1);
    #pragma omp task shared(fn2)
    fn2 = comp_fib_numbers(n-2);
    #pragma omp taskwait
    return(fn1 + fn2);
}
```

- Functionally correct
- Poor performance
  - Tasks are very fine-grained
  - Too much parallelism?
- Improving programmability
  - Cut-off strategies
Tasking use case: Cholesky (synchronization)

```c
void cholesky(int ts, int nt, double* a[nt][nt]) {
    for (int k = 0; k < nt; k++) {
        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][i], ts, ts);
            }
            #pragma omp task
            syrk(a[k][i], a[i][i], ts, ts);
        }
        #pragma omp taskwait
    }
}
```

- Complex synchronization patterns
  - Splitting computational phases
  - taskwait or taskgroup
  - Needs complex code analysis

- Improving programmability
  - OpenMP dependences
  - It also improves composability
Tasking use case: saxpy (blocking)

Difficult to determine grain
- 1 single iteration → to fine
- whole loop → no parallelism

Manually transform the code
- blocking techniques

Improving programmability
- OpenMP taskloop

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

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++) {
    }
};
```
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++) {
    }
}
```

```c
#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++) {
    }
}
```

- 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)

  - if(scalar-expression)
  - final(scalar-expression)
  - mergeable
  - untied
  - priority(priority-value)
  - collapse(n)
  - nogroup
  - allocate([allocator:] list)

Chunks/Grain

Data Environment

Cutoff Strategies

Scheduler (R/H)

Miscellaneous

Chunks/Grain

Data Environment

Cutoff Strategies

Scheduler (R/H)

Miscellaneous

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)

  - if(scalar-expression)
  - final(scalar-expression)
  - mergeable
  - untied
  - priority(priority-value)
  - collapse(n)
  - nogroup
  - allocate([allocator:] list)
Taskloop decomposition approaches

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

  ```
  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

  ```
  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;
}
```
OpenMP Tasking, Part 1: Fundamentals
Xavier Teruel

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:

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

→ Fortran syntax:

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

- Where clause is any of the clauses accepted by `taskloop` or `simd` directives
Thanks!
Slides and Exercises

- https://tinyurl.com/9sxmvsm2