

Programming OpenMP

Tasking Model

Christian Terboven



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

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

→ recursive functions

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

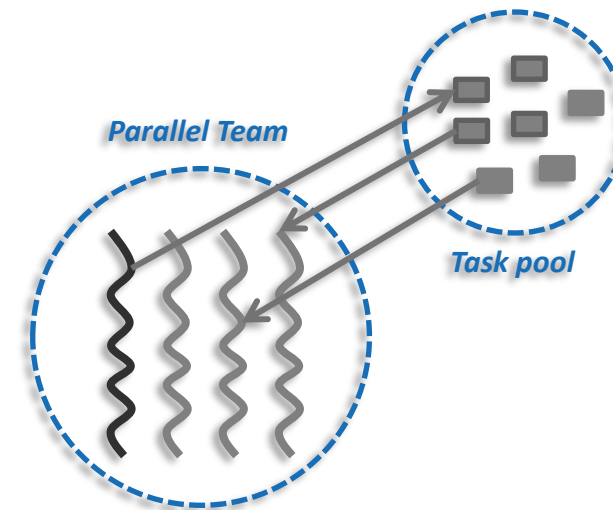
- Several scenarios are possible:

→ single creator, multiple creators, nested tasks (tasks & WS)

- All threads in the team are candidates to execute tasks

- Example (unstructured parallelism)

```
#pragma omp parallel
#pragma omp master
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)

Data Environment

- allocate([allocator:] list)
- detach(event-handler)

Miscellaneous

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

Cutoff Strategies

- depend(dep-type: list)

Synchronization

- untied
- priority(priority-value)
- affinity(list)

Task Scheduling

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)

```
#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 ☹️

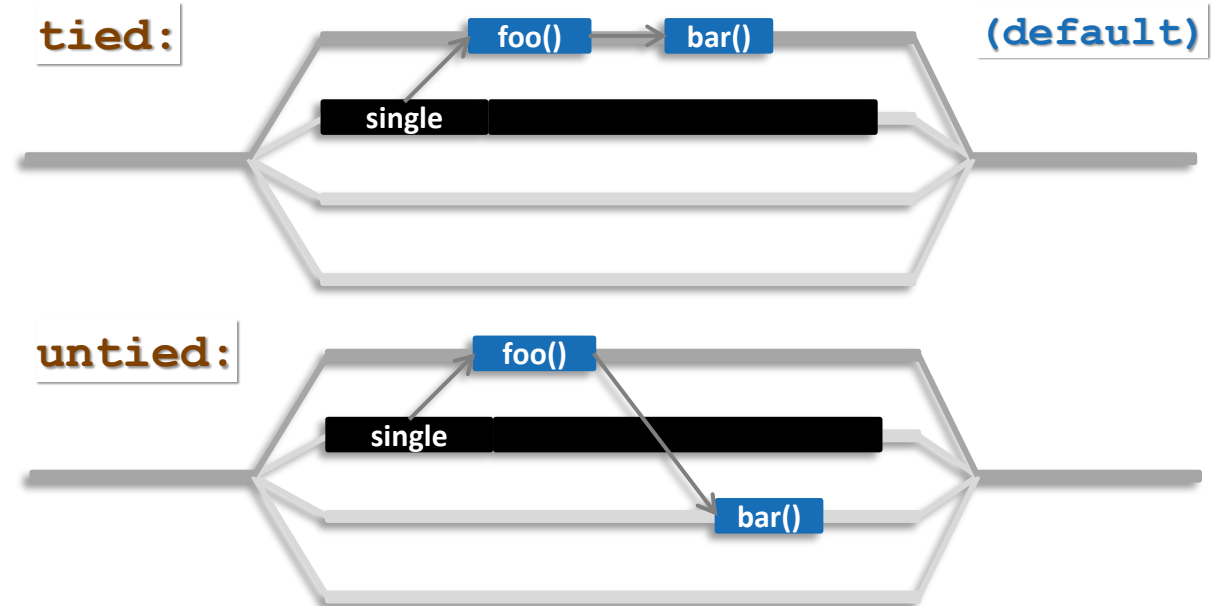
Task scheduling: taskyield directive

- 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

```
#pragma omp taskyield
```

- Scheduling [tied/untied] tasks: example

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



Task synchronization: taskwait directive

- The taskwait directive (shallow task synchronization)

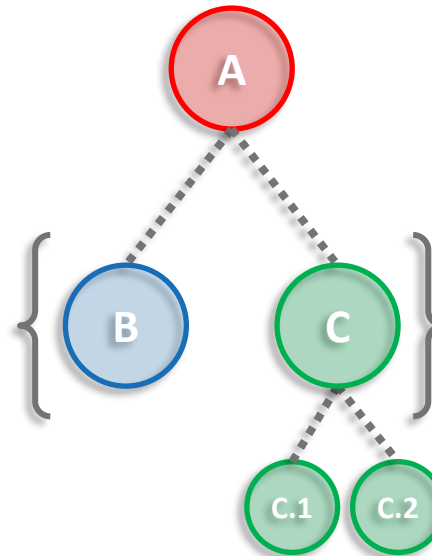
→ It is a stand-alone directive

```
#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 :A
  {
    #pragma omp task :B
    { ... }
    #pragma omp task :C
    { ... #C.1; #C.2; ... }
    #pragma omp taskwait
  }
} // implicit barrier will wait for C.x
```

wait for...



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)

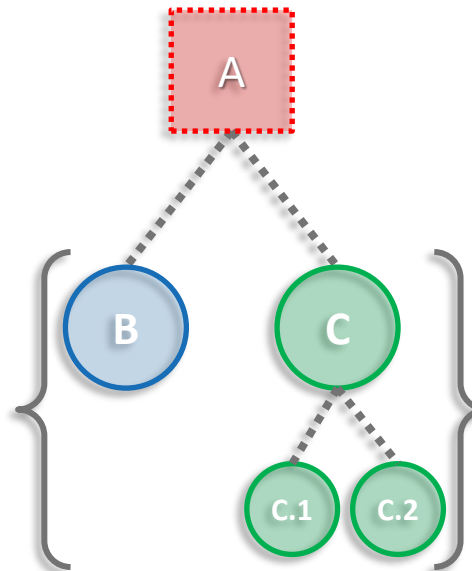
→ attached to a structured block; completion of all descendants of the current task; TSP at the end

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

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

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

wait for...



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)...

```

int A[SIZE];
#pragma omp threadprivate(A)

// ...
#pragma omp task
{
    // A: threadprivate
}
  
```

1

```

int *p;

p = malloc(sizeof(float)*SIZE);

#pragma omp task
{
    // *p: shared
}
  
```

2

```

void foo(void){
    static int s = MN;
}

#pragma omp task
{
    foo(); // s@foo(): shared
}
  
```

3

```

#pragma omp task
{
    int x = MN;
    // Scope of x: private
}
  
```

5

```

#pragma omp task
{
    static int y;
    // Scope of y: shared
}
  
```

4

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**

- Pre-determined rules (could not change)

- Explicit data-sharing clauses (+ default)

- Implicit data-sharing rules

```
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:
        }
    }
}
```

- (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]

```
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]

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

- Task participates in a reduction operation [3]

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

Intel compiler 2021 still not able to compile!
Can be compiled with Clang 11!

Tasking illustrated

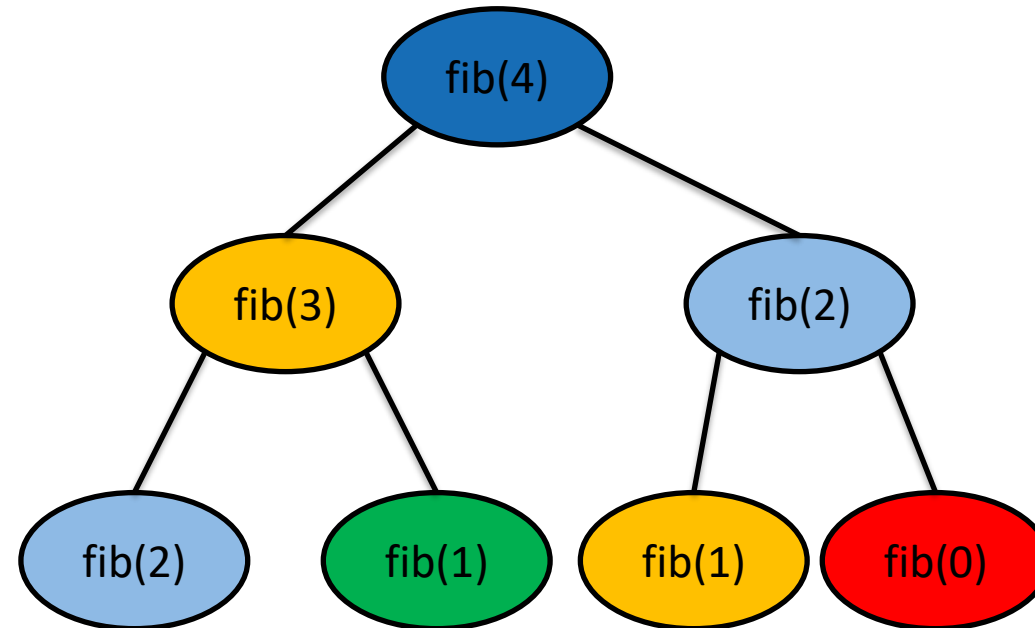
Fibonacci illustrated

```
1  int main(int argc,  
2      char* argv[])  
3  {  
4      [...]  
5      #pragma omp parallel  
6      {  
7          #pragma omp single  
8          {  
9              fib(input);  
10         }  
11     }  
12     [...]  
13 }
```

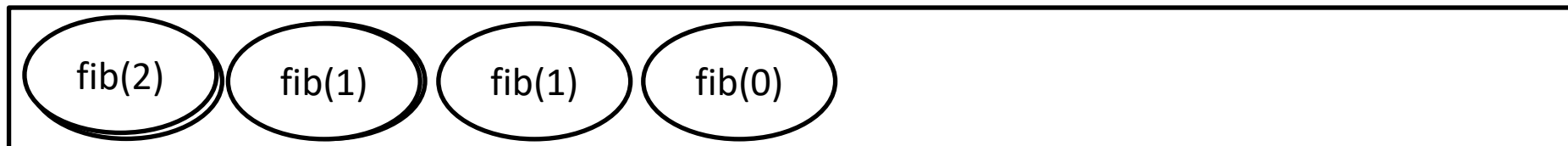
```
14  int fib(int n)  {  
15      if (n < 2) return n;  
16      int x, y;  
17      #pragma omp task shared(x)  
18      {  
19          x = fib(n - 1);  
20      }  
21      #pragma omp task shared(y)  
22      {  
23          y = fib(n - 2);  
24      }  
25      #pragma omp taskwait  
26      return x+y;  
27 }
```

- 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



Task Queue



- 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
- ...

