INSTITUT POLYTECHNIQUE DE PARIS

Task parallelism with OpenMP task

Elisabeth Brunet,

Based on Raymond Namyst, Pierre-André Wacrenier IT224 course

Institut Mines-Télécom

Why task parallelism ?

- Loops parallelism is no longer enough

- Heterogeneous parallelism
  - More parallelism
  - Fine grain parallelism
  - Fit non iterative data structure like grpah, trees, etc.
  - Producer/consumer problem
  - Recursive execution

Why task parallelism ?

- Loops parallelism is no longer enough

- Heterogeneous parallelism
  - More parallelism
  - Fine grain parallelism
  - Fit non iterative data structure like grpah, trees, etc.
  - Producer/consumer problem
  - Recursive execution

→ Since OpenMP4.0, all is task :
Threads are executed in an implicit task

OpenMP task

- Tasks are code chunks which are implicitly placed in a “pool of task” to be executed in parallel
  - Task are generated using the #pragma omp task directive
  - Task execution is potentially postponed until it get picked by a thread
    - Task scheduling is performed by a dynamic runtime system

```
{

 {
  printf ("Start\n");


  printf ("Middle (executed by %d)\n",
        omp_get_thread_num ());

  printf ("End\n");
 }
}
```

OpenMP task

- Tasks are code chunks which are implicitly placed in a “pool of task” to be executed in parallel
  - Task are generated using the #pragma omp task directive
  - Task execution is potentially postponed until it get picked by a thread
    - Task scheduling is performed by a dynamic runtime system

```
{
#pragma omp parallel
 {
  printf ("Start\n");

#pragma omp task
  printf ("Middle (executed by %d)\n",
        omp_get_thread_num ());

  printf ("End\n");
 }
}
```

OpenMP task

- In this example
  - Each thread generates one task
  - Tasks can be executed by any thread
- All tasks must complete before the next synchronization point
  - Barrier
  - End of parallel region

```
{
#pragma omp parallel
 {
  printf ("Start\n");

#pragma omp task
  printf ("Middle (executed by %d)\n",
        omp_get_thread_num ());

  printf ("End\n");
 }
}
```

first-task.c and second-task.c

OpenMP task

- In the general case, only one thread generate tasks
  - Omp master or single
    - Only one thread executes the code, i.e. generates tasks
    - The others wait on an implicit barrier
- All threads cooperate to empty the pool of ready-tasks

```
#pragma omp parallel
#pragma omp single
{
#pragma omp task
 {
  printf ("Task 1 executed by Thread %d\n", omp_get_thread_num ());
  sleep (1);
  printf ("End of Task 1\n");
 }
#pragma omp task
 {
  printf ("Task 2 executed by Thread %d\n", omp_get_thread_num ());
  sleep (1);
  printf ("End of Task 2\n");
 }
}
```

See task.c

OpenMP task

- Warning
  - The following code behaves quite differently!

```
#pragma omp parallel
{
#pragma omp single
#pragma omp task
 {
  printf ("Task 1 executed by Thread %d\n", omp_get_thread_num ());
  sleep (1);
  printf ("End of Task 1\n");
 }
#pragma omp single
#pragma omp task
 {
  printf ("Task 2 executed by Thread %d\n", omp_get_thread_num ());
  sleep (1);
  printf ("End of Task 2\n");
 }
}
```

See task.c

OpenMP task

- Warning
  - The following code behaves quite differently!
  - Adding nowait allows task creations to take place in parallel

```
#pragma omp parallel
{
#pragma omp single nowait
#pragma omp task
 {
  printf ("Task 1 executed by Thread %d\n", omp_get_thread_num ());
  sleep (1);
  printf ("End of Task 1\n");
 }
#pragma omp single
#pragma omp task
 {
  printf ("Task 2 executed by Thread %d\n", omp_get_thread_num ());
  sleep (1);
  printf ("End of Task 2\n");
 }
}
```

See task.c

OpenMP task – undeterministic task number

- Handle an arbitrary number of elements
  - Not known *a priori*
- Caution to variable

```
#pragma omp parallel
#pragma omp single
{
 element_t elt;

 while (elt = get_next ())
#pragma omp task firstprivate (elt)
   treat (elt);
}
```

OpenMP task – tied by default !

- Tied to the 1st thread that starts its execution
- When a tied task is interrupted, no other thread can continue its execution…
- Codes using omp_get_thread_num are guaranteed to stick to the same thread

```
#pragma omp parallel
#pragma omp single
{
 element_t elt;

 while (elt = get_next ())
#pragma omp task firstprivate (elt)
   treat (elt);
}
```

OpenMP task – tied by default !

- Tied to the 1st thread that starts its execution
- When a tied task is interrupted, no other thread can continue its execution…
- Codes using omp_get_thread_num are guaranteed to stick to the same thread

```
#pragma omp parallel
#pragma omp single
{
 element_t elt;

 while (elt = get_next ())
#pragma omp task firstprivate (elt)
   treat (elt);
}
```

◯ Implicit task

OpenMP task – tied by default !

- Tied to the 1st thread that starts its execution
- When a tied task is interrupted, no other thread can continue its execution…
- Codes using omp_get_thread_num are guaranteed to stick to the same thread

```
#pragma omp parallel
#pragma omp single
{
 element_t elt;

 while (elt = get_next ())
#pragma omp task firstprivate (elt)
   treat (elt);
}
```

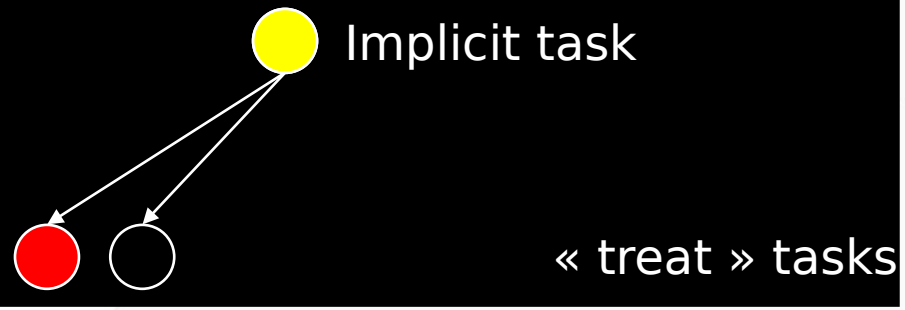
Implicit task

OpenMP task – tied by default !

- Tied to the 1st thread that starts its execution
- When a tied task is interrupted, no other thread can continue its execution…
- Codes using omp_get_thread_num are guaranteed to stick to the same thread

```
#pragma omp parallel
#pragma omp single
{
 element_t elt;

 while (elt = get_next ())
#pragma omp task firstprivate (elt)
   treat (elt);
}
```

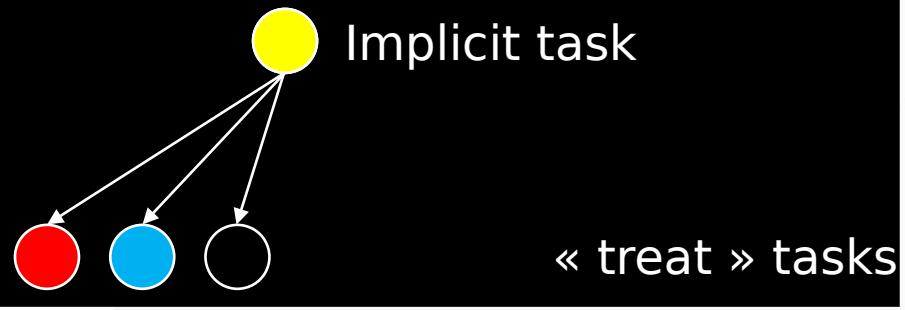


OpenMP task – tied by default !

- Tied to the 1st thread that starts its execution
- When a tied task is interrupted, no other thread can continue its execution…
- Codes using omp_get_thread_num are guaranteed to stick to the same thread

```
#pragma omp parallel
#pragma omp single
{
 element_t elt;

 while (elt = get_next ())
#pragma omp task firstprivate (elt)
   treat (elt);
}
```

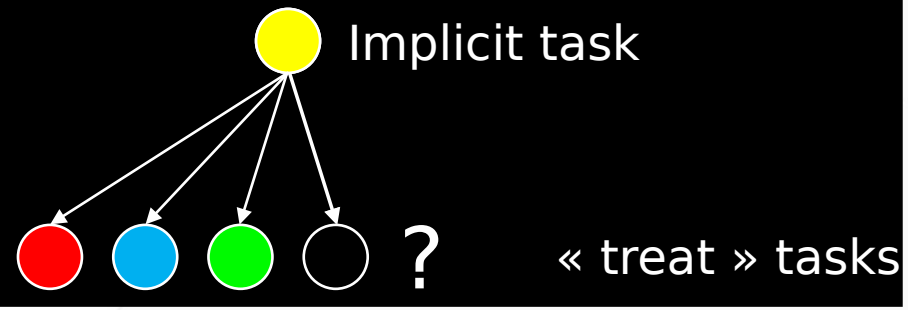


OpenMP task – tied by default !

- Tied to the 1st thread that starts its execution
- When a tied task is interrupted, no other thread can continue its execution…
- Codes using omp_get_thread_num are guaranteed to stick to the same thread

```
#pragma omp parallel
#pragma omp single
{
 element_t elt;

 while (elt = get_next ())
#pragma omp task firstprivate (elt)
   treat (elt);
}
```

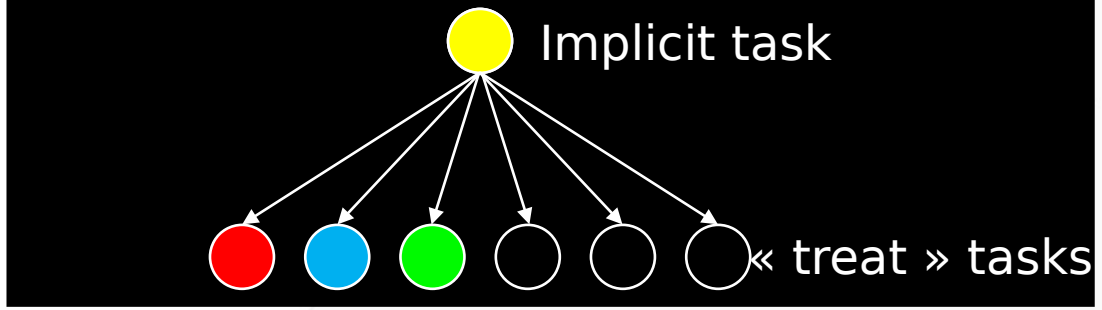


OpenMP task – tied by default !

- Tied to the 1st thread that starts its execution
- When a tied task is interrupted, no other thread can continue its execution…
- Codes using omp_get_thread_num are guaranteed to stick to the same thread

```
#pragma omp parallel
#pragma omp single
{
 element_t elt;

 while (elt = get_next ())
#pragma omp task firstprivate (elt)
   treat (elt);
}
```



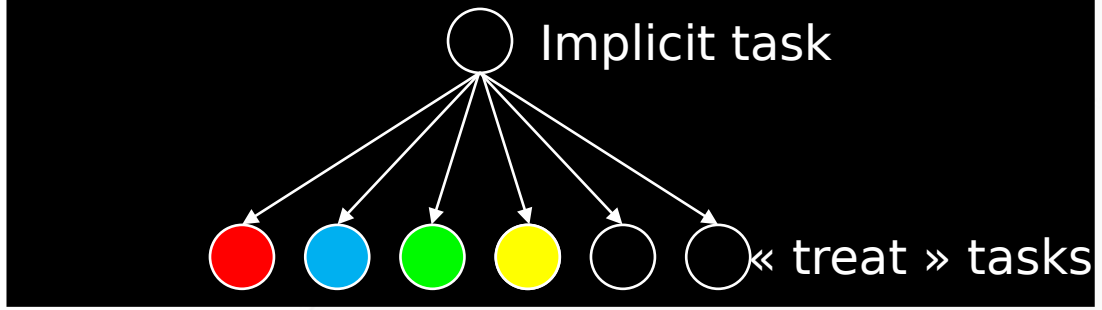
Either…

OpenMP task – tied by default !

- Tied to the 1st thread that starts its execution
- When a tied task is interrupted, no other thread can continue its execution…
- Codes using omp_get_thread_num are guaranteed to stick to the same thread

```
#pragma omp parallel
#pragma omp single
{
 element_t elt;

 while (elt = get_next ())
#pragma omp task firstprivate (elt)
   treat (elt);
}
```



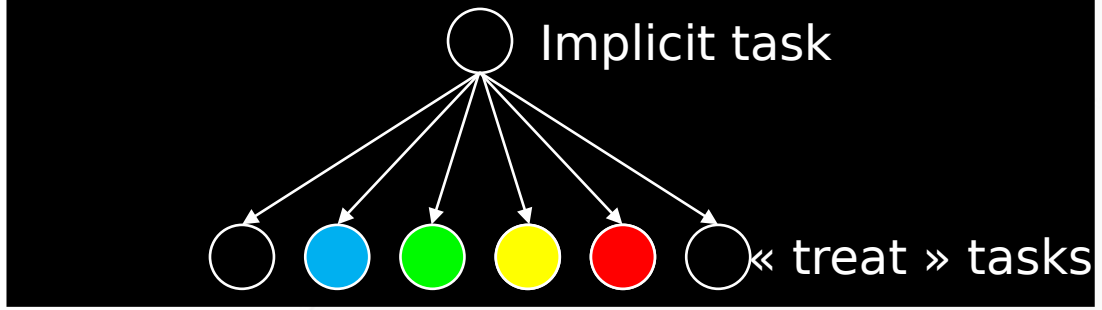
Either…

OpenMP task – tied by default !

- Tied to the 1st thread that starts its execution
- When a tied task is interrupted, no other thread can continue its execution…
- Codes using omp_get_thread_num are guaranteed to stick to the same thread

```
#pragma omp parallel
#pragma omp single
{
 element_t elt;

 while (elt = get_next ())
#pragma omp task firstprivate (elt)
   treat (elt);
}
```



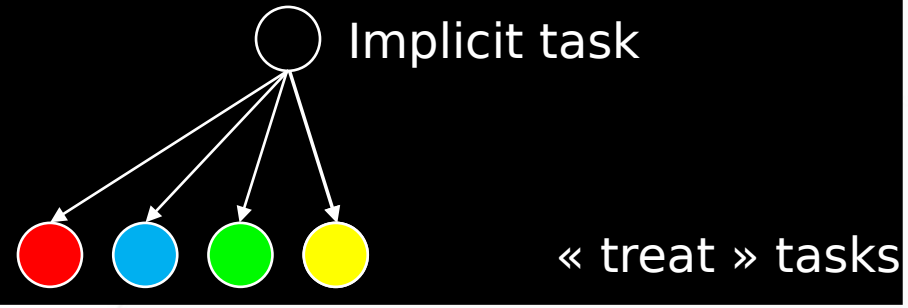
Either…

OpenMP task – tied by default !

- Tied to the 1st thread that starts its execution
- When a tied task is interrupted, no other thread can continue its execution…
- Codes using omp_get_thread_num are guaranteed to stick to the same thread

```
#pragma omp parallel
#pragma omp single
{
 element_t elt;

 while (elt = get_next ())
#pragma omp task firstprivate (elt)
   treat (elt);
}
```



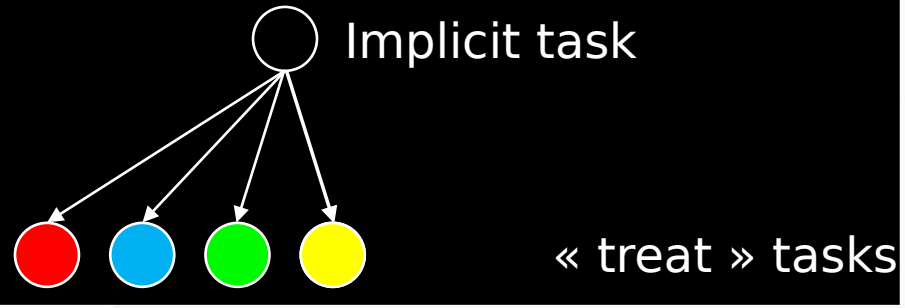
… or

OpenMP task – tied by default !

- Tied to the 1st thread that starts its execution
- When a tied task is interrupted, no other thread can continue its execution…
- Codes using omp_get_thread_num are guaranteed to stick to the same thread

```
#pragma omp parallel
#pragma omp single
{
 element_t elt;

 while (elt = get_next ())
#pragma omp task firstprivate (elt)
   treat (elt);
}
```

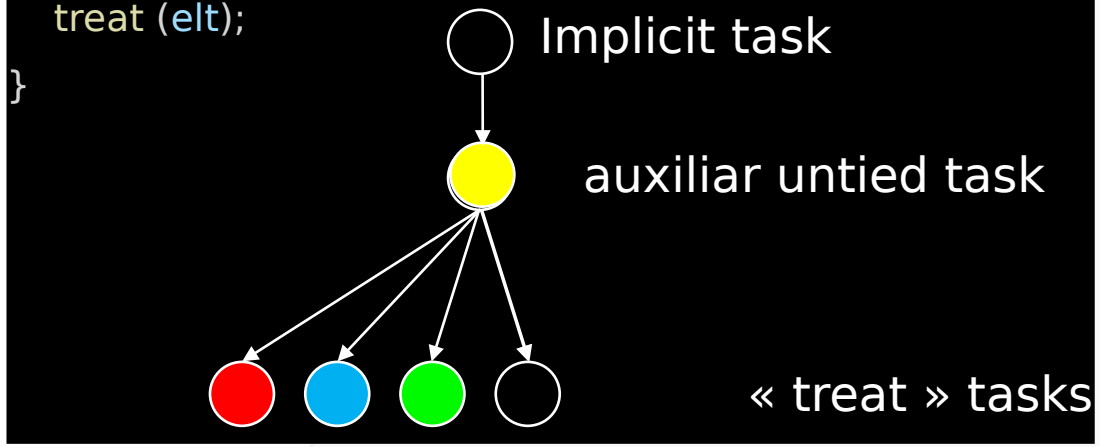


Potentialy stuck for long time, as only the yellow thread can run the implicit task!

OpenMP task – but can be untied !

- Tied to the 1st thread that starts its execution
- When a tied task is interrupted, no other thread can continue its execution…
- Codes using omp_get_thread_num are guaranteed to stick to the same thread

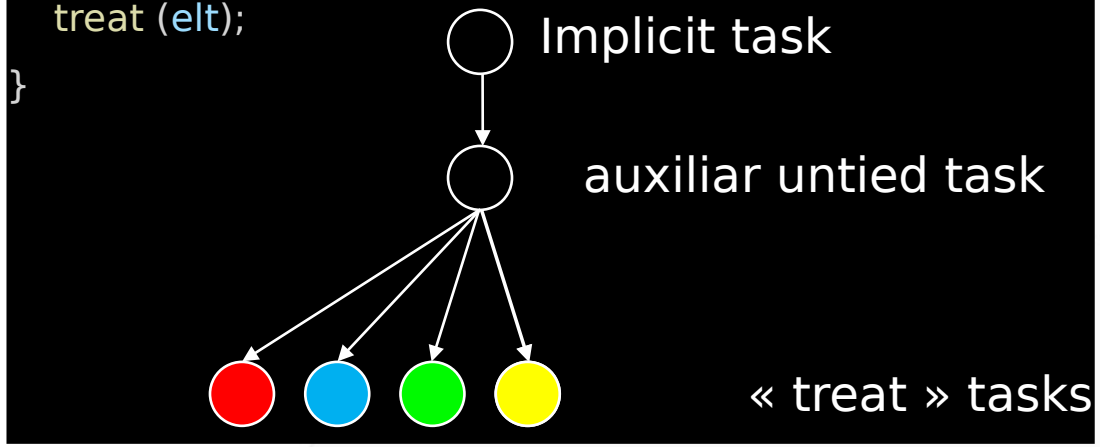
```
#pragma omp parallel
#pragma omp single
{
 element_t elt;
#pragma omp task untied
 while (elt = get_next ())
#pragma omp task firstprivate (elt)
   treat (elt);
}
```



OpenMP task – but can be untied !

- Tied to the 1st thread that starts its execution
- When a tied task is interrupted, no other thread can continue its execution…
- Codes using omp_get_thread_num are guaranteed to stick to the same thread

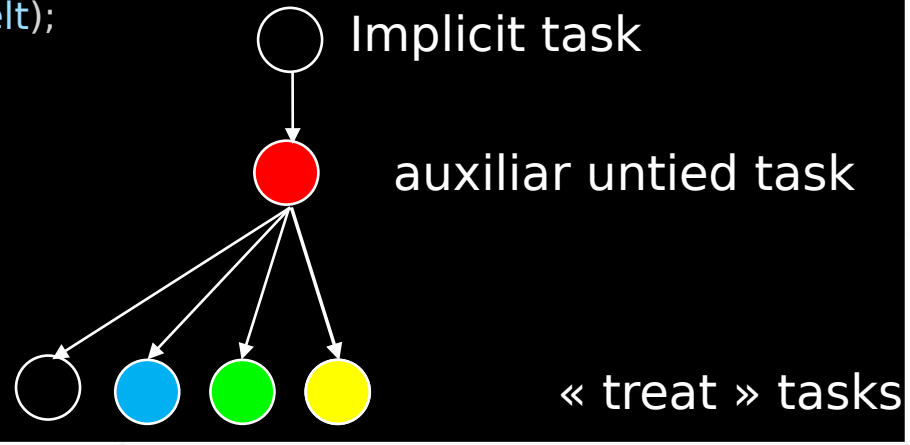
```
#pragma omp parallel
#pragma omp single
{
 element_t elt;
#pragma omp task untied
 while (elt = get_next ())
#pragma omp task firstprivate (elt)
   treat (elt);
}
```



OpenMP task – but can be untied !

- Tied to the 1st thread that starts its execution
- When a tied task is interrupted, no other thread can continue its execution…
- Codes using omp_get_thread_num are guaranteed to stick to the same thread

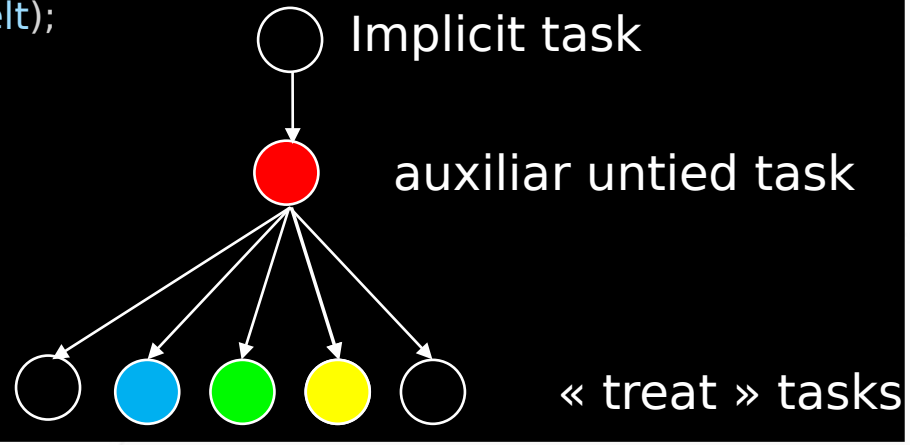
```
#pragma omp parallel
#pragma omp single
{
 element_t elt;
#pragma omp task untied
 while (elt = get_next ())
#pragma omp task firstprivate (elt)
   treat (elt);
}
```



OpenMP task – but can be untied !

- Tied to the 1st thread that starts its execution
- When a tied task is interrupted, no other thread can continue its execution…
- Codes using omp_get_thread_num are guaranteed to stick to the same thread

```
#pragma omp parallel
#pragma omp single
{
 element_t elt;
#pragma omp task untied
 while (elt = get_next ())
#pragma omp task firstprivate (elt)
   treat (elt);
}
```



OpenMP task - synchronization

- Fibonacci
  - Recursive parallelism

```
int fib_par (int n)
{
 if (n < 2)
   return n;

 int r1, r2;
#pragma omp task shared (r1)
 r1 = fib_par (n - 1);
#pragma omp task shared (r2)
 r2 = fib_par (n - 2);

 return r1 + r2;
}
```

- `#pragma omp parallel shared(r)`
  `#pragma omp single`
  `r = fib_par (n);`

See fib.c

OpenMP task - synchronization

- Fibonacci
  - Recursive parallelism
  - Task tree generated

```
int fib_par (int n)
{
 if (n < 2)
   return n;

 int r1, r2;
#pragma omp task shared (r1)
 r1 = fib_par (n - 1);
#pragma omp task shared (r2)
 r2 = fib_par (n - 2);

 return r1 + r2;
}
```

- `#pragma omp parallel shared(r)`
  `#pragma omp single`
  `r = fib_par (n);`

See fib.c

OpenMP task - synchronization

- Fibonacci
  - Recursive parallelism
  - Task tree generated
  - → **RACE CONDITION**

```
int fib_par (int n)
{
 if (n < 2)
   return n;

 int r1, r2;
#pragma omp task shared (r1)
 r1 = fib_par (n - 1);
#pragma omp task shared (r2)
 r2 = fib_par (n - 2);

 return r1 + r2;
}
```

- `#pragma omp parallel shared(r)`
  `#pragma omp single`
  `r = fib_par (n);`

See fib.c

OpenMP task - synchronization

- Taskwait directive
  - Waits completion of *child* tasks

```
int fib_par (int n)
{
 if (n < 2)
   return n;

 int r1, r2;
#pragma omp task shared (r1)
 r1 = fib_par (n - 1);
#pragma omp task shared (r2)
 r2 = fib_par (n - 2);
#pragma omp taskwait
 return r1 + r2;
}
```

- `#pragma omp parallel shared(r)`
  `#pragma omp single`
  `r = fib_par (n);`

See fib.c

OpenMP taskwait vs taskgroup

```
#pragma omp task
{
  #pragma omp task
  f ();
  #pragma omp task
  g ();
}
#pragma omp task
  h ();
#pragma omp taskwait
// Only h () is guaranteed to be completed
```

OpenMP taskwait vs taskgroup

```
#pragma omp task
{
  #pragma omp task
  f ();
  #pragma omp task
  g ();
}
#pragma omp task
  h ();
#pragma omp taskwait
// Only h () is guaranteed to be completed
```

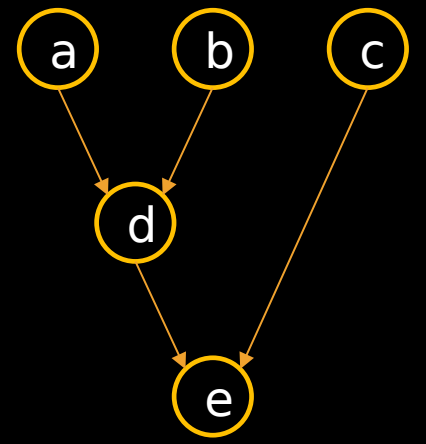
```
#pragma omp taskgroup
{
  #pragma omp task
  {
    #pragma omp task
    f ();
    #pragma omp task
    g ();
  }
  #pragma omp task
    h ();
}
// f(), g() and h () are guaranteed to be
// completed
```

OpenMP task dependencies

- Need a tighter control on synchronizations
- Say we want to *taskify* the following code
  - Where to insert taskwait directives?

```
{
 int a, b, c, d, e;
 {
  a = fa ();
  b = fb ();
  c = fc ();
  d = fadd (a, b);
  e = fmul (c, d);
 }

 printf ("result = %d\n", e);
}
```



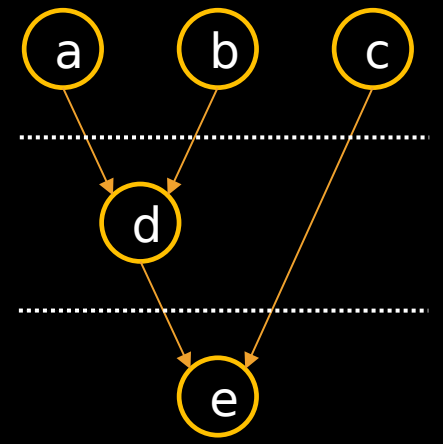
See flow.c

OpenMP task dependencies

- Need a tighter control on synchronizations
- Say we want to *taskify* the following code
  - Where to insert taskwait directives?

```
{
 int a, b, c, d, e;
 {
  a = fa ();
  b = fb ();
  c = fc ();
  d = fadd (a, b);
  e = fmul (c, d);
 }

 printf ("result = %d\n", e);
}
```



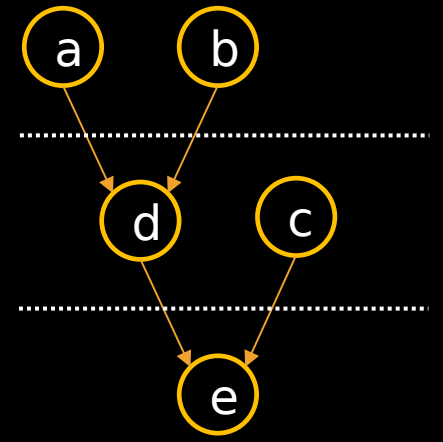
See flow.c

OpenMP task dependencies

- Need a tighter control on synchronizations
- Say we want to *taskify* the following code
  - Where to insert taskwait directives?

```
{
 int a, b, c, d, e;
 {
  a = fa ();
  b = fb ();
  c = fc ();
  d = fadd (a, b);
  e = fmul (c, d);
 }

 printf ("result = %d\n", e);
}
```



See flow.c

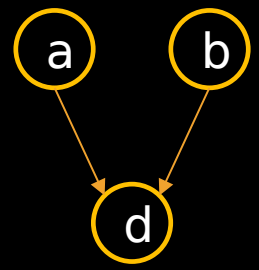
OpenMP task dependencies

- Task dependencies inferred by OpenMP runtime thanks to in/out/inout specified accesses to variables
- **depend** clause
  - depend (out: v)
    - The task modifies v
  - depend (in: v)
    - The task reads v
  - depend (mutexinoutset: v)
    - Only one task accessing v can run at a time, but no specific order is required

```
#pragma omp task shared (a) depend (out: a)
 a = fa ();

#pragma omp task shared (b) depend (out: b)
 b = fb ();

#pragma omp task shared (d) depend (out: d) depend (in: a, b)
 d = fadd (a, b);
```



See flow-depend.c

OpenMP task dependencies

- Dependencies only apply to tasks which have the same parent task
- Depend clauses only use the address of variables internally
  - OpenMP uses addresses as keys to match in/out/inout clauses
  - Variables are not accessed
- OpenMP drops depend(in: v) if no depend(out: v) was previously encountered…

Heading to the lab session !

Titre