

Tasking in OpenMP

Paolo Burgio

paolo.burgio@unimore.it



Outline

- › Expressing parallelism
 - Understanding parallel threads

- › Memory Data management
 - Data clauses

- › Synchronization
 - Barriers, locks, critical sections

- › Work partitioning
 - Loops, sections, single work, tasks...

- › Execution devices
 - Target



A history of OpenMP

- › 1997
 - OpenMP for Fortran 1.0

- › 1998
 - OpenMP for C/C++ 1.0

- › 2000
 - OpenMP for Fortran 2.0

- › 2002
 - OpenMP for C/C++ 2.5

- › 2008
 - OpenMP 3.0

- › 2011
 - OpenMP 3.1

- › 2014
 - OpenMP 4.5

Thread-centric

Regular, loop-based parallelism

Task-centric

Irregular, parallelism → tasking

Devices

Heterogeneous parallelism, *à la* GP-GPU



OpenMP programming patterns

- › "Traditional" OpenMP has a thread-centric execution model
 - Fork/join
 - Master-slave

- › Create a team of threads...
 - ..then partition the work among them
 - Using work-sharing constructs

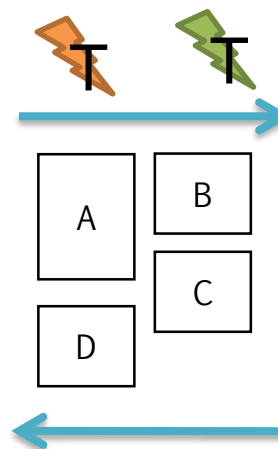
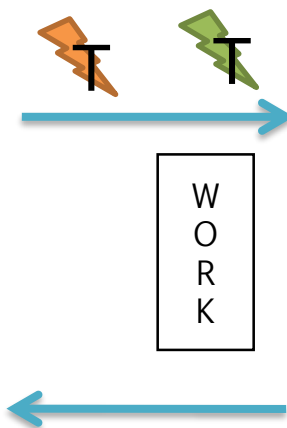
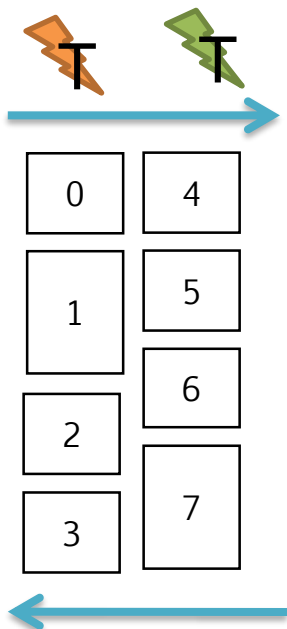


OpenMP programming patterns

```
#pragma omp for
for (int i=0; i<8; i++)
{
    // ...
}
```

```
#pragma omp single
{
    work();
}
```

```
#pragma omp sections
{
    #pragma omp section
    { A(); }
    #pragma omp section
    { B(); }
    #pragma omp section
    { C(); }
    #pragma omp section
    { D(); }
}
```



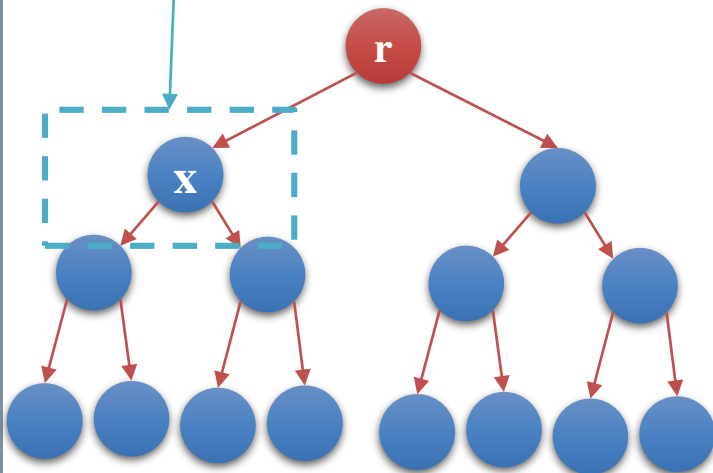


Exercise

Let's
code!

- › Traverse a tree
 - Perform the same operation on all elements
 - Download sample code

- › Recursive

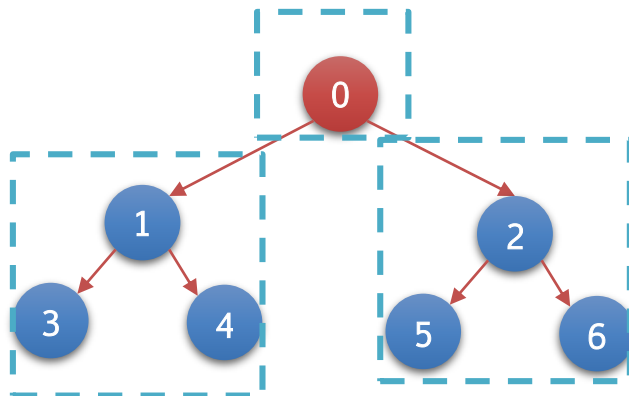




Exercise

Let's
code!

- › Now, parallelize it!
 - From the example



```
void traverse_tree(node_t *n)
{
    doYourWork(n);

    if(n->left)
        traverse_tree(n->left);

    if(n->right)
        traverse_tree(n->right);
}

...
traverse_tree(root);
```



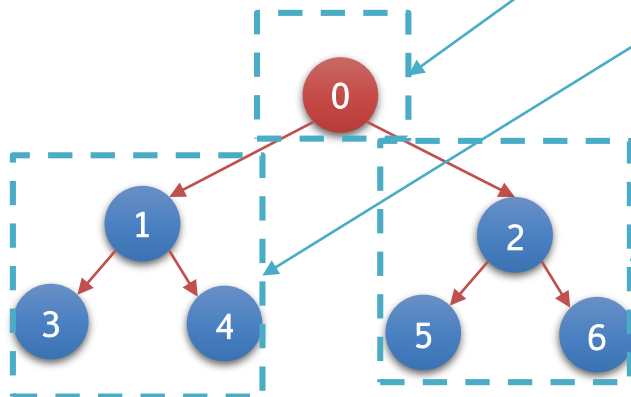
Solved: traversing a tree in parallel

> Recursive

- Parreg+section for each call
- Nested parallelism

> Assume the very first time we call `traverse_tree`

- Root node




```
void traverse_tree(node_t *n)
{
    #pragma omp parallel sections
    {
        #pragma omp section
        doYourWork(n);

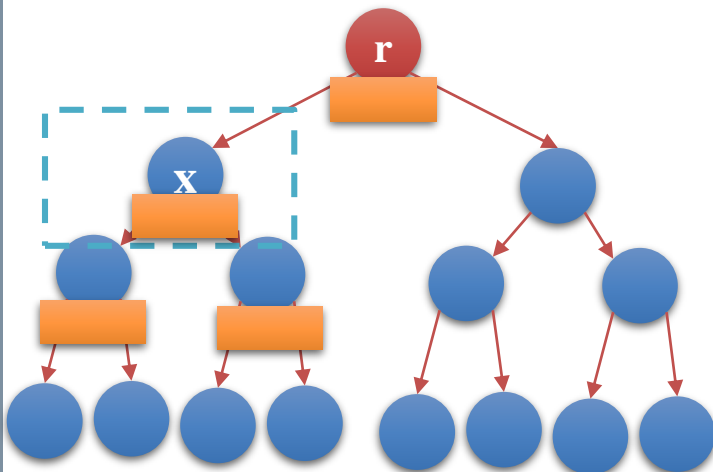
        #pragma omp section
        if(n->left)
            traverse_tree(n->left);
        #pragma omp section
        if(n->right)
            traverse_tree(n->right);
    }
}

...
traverse_tree(root);
```


Catches (1)

› Cannot nest worksharing constructs without an intervening parreg

- And its barrier... 
- Costly



```
void traverse_tree(node_t *n)
{
    doYourWork(n);

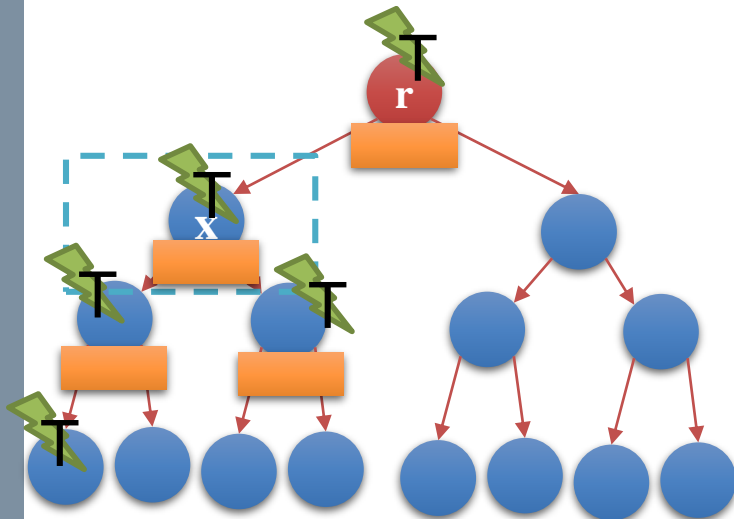
    [ #pragma omp parallel sections
    {
        #pragma omp section
        if(n->left)
        [ traverse_tree(n->left);
        #pragma omp section
        if(n->right)
        traverse_tree(n->right);
    } // Barrier
    } // Parreg barrier
```

```
...
traverse_tree(root);
```



Catches (2)

- › #threads grows exponentially
 - Harder to manage



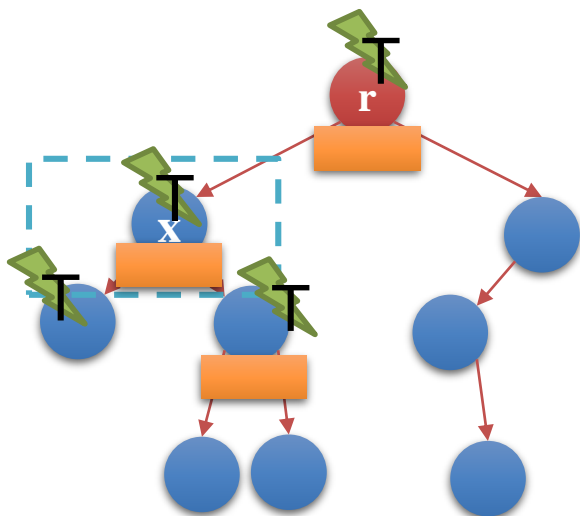
```
void traverse_tree(node_t *n)
{
    doYourWork(n);

    #pragma omp parallel sections
    {
        #pragma omp section
        if(n->left)
            traverse_tree(n->left);
        #pragma omp section
        if(n->right)
            traverse_tree(n->right);
    } // Barrier
} // Prrrg barrier

...
traverse_tree(root);
```

Catches (3)

- › Code is not easy to understand
- › Even harder to modify
 - What if I add a third child node?



```
void traverse_tree(node_t *n)
{
    doYourWork(n);

    #pragma omp parallel sections
    {
        #pragma omp section
        if(n->left)
            traverse_tree(n->left);
        #pragma omp section
        if(n->right)
            traverse_tree(n->right);
    } // Barrier
} // Prrrg barrier

...
traverse_tree(root);
```



Limitations of "traditional" WS

Cannot nest worksharing constructs without an intervening parreg

- › Parreg are traditionally costly
 - A lot of operations to create a team of threads
 - Barrier...

Parreg	Static loops prologue	Dyn loops start
30k cycles	10-150 cycles	5-6k cycles

- › The number of threads explodes and it's harder to manage
 - Parreg => create new threads

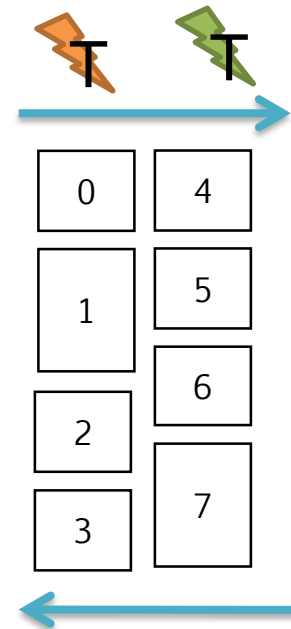


Limitations of "traditional" WS

It is cumbersome to create parallelism dynamically

- › In loops, sections
 - Work is statically determined!
 - Before entering the construct
 - Even in dynamic loops
- › "if <condition>, then create work"

```
#pragma omp for
for (int i=0; i<8; i++)
{
    // ...
}
```

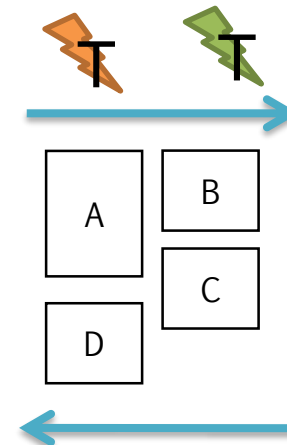


Limitations of "traditional" WS

Poor semantics for irregular workload

- › Sections-based parallelism that is anyway cumbersome to write
 - OpenMP was born for loop-based parallelism
- › Code not scalable
 - Even a small modifications causes you to re-think the strategy

```
#pragma omp sections
{
    #pragma omp section
    { A(); }
    #pragma omp section
    { B(); }
    #pragma omp section
    { C(); }
    #pragma omp section
    { D(); }
}
```

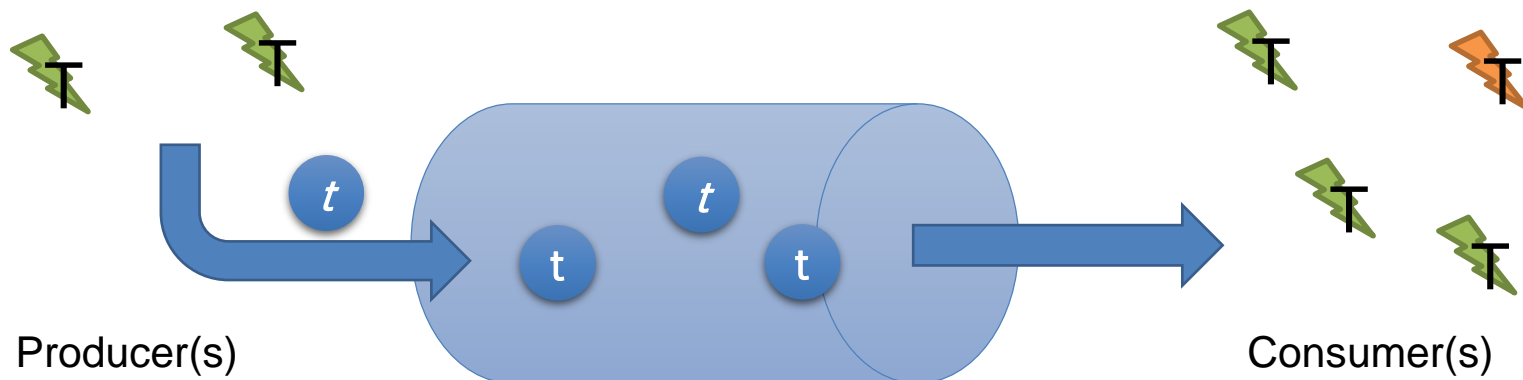




A different parallel paradigm

A **work-oriented** paradigm for partitioning workloads

- › Implements a **producer-consumer paradigm**
 - As opposite to OpenMP thread-centric model
- › Introduce the task pool
 - Where units of work (OpenMP tasks)
 - are pushed by threads
 - and pulled and executed by threads
- › E.g., **implemented** as a fifo queue (aka task queue)





The task directive

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

Where clauses can be:

```
if([ task : ]scalar-expression)  
final(scalar-expression)  
untied  
default(shared | none)  
mergeable  
private(list)  
firstprivate(list)  
shared(list)  
depend(dependence-type : list)  
priority(priority-value)
```

- › We will see only data sharing clauses
 - Same as parallel but...**DEFAULT IS NOT SHARED!!!!**

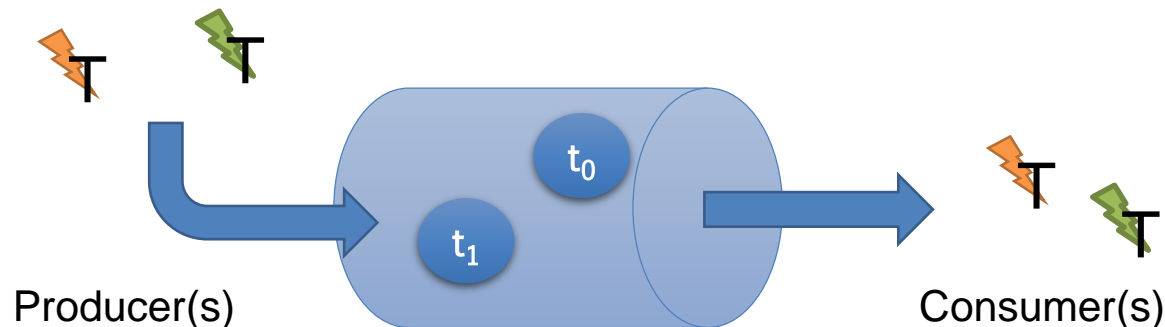
Two sides

- › Tasks are **produced**
- › Tasks are **consumed**

Let's
code!

- › Try this!
 - t_0 and t_1 are `printf`
 - Also, print who produces

```
/* Create threads */  
#pragma omp parallel num_threads(2)  
{  
    /* Push a task in the q */  
    #pragma omp task  
    {  
        t0();  
    }  
  
    /* Push another task in the q */  
    #pragma omp task  
    t1();  
} // Implicit barrier
```

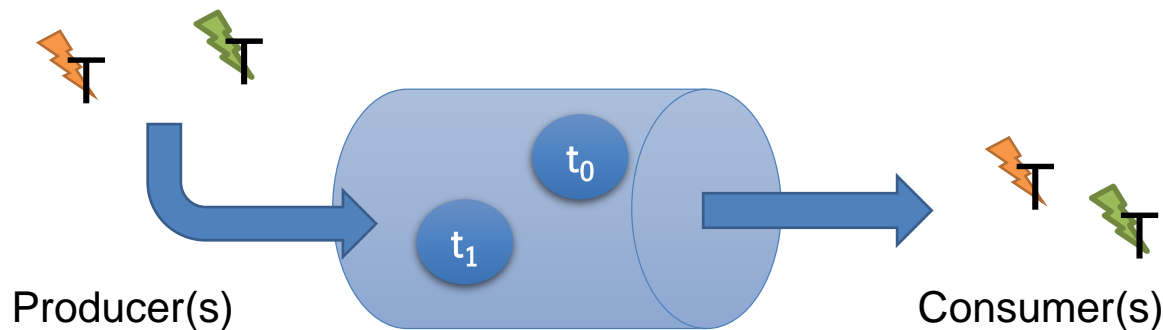




I cheated a bit

- › How many producers?
 - So, how many tasks?

```
/* Create threads */  
#pragma omp parallel num_threads(2)  
{  
    /* Push a task in the q */  
    #pragma omp task  
    {  
        t0();  
    }  
  
    /* Push another task in the q */  
    #pragma omp task  
    {  
        t1();  
    }  
} // Implicit barrier
```

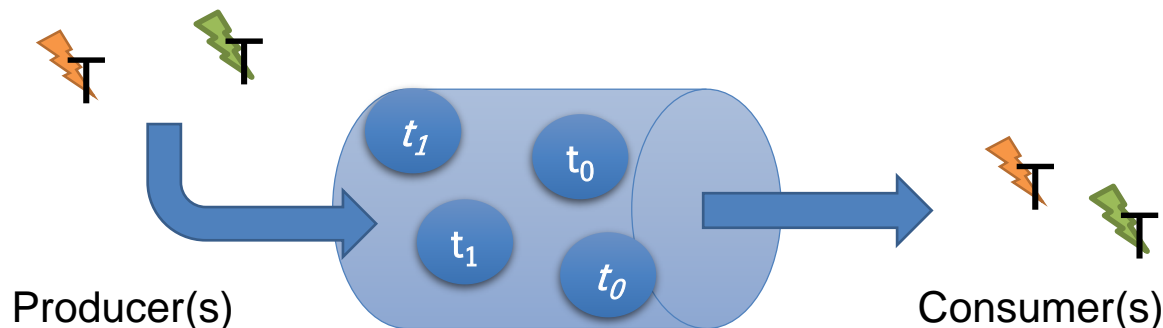




I cheated a bit

- › How many producers?
 - So, how many tasks?

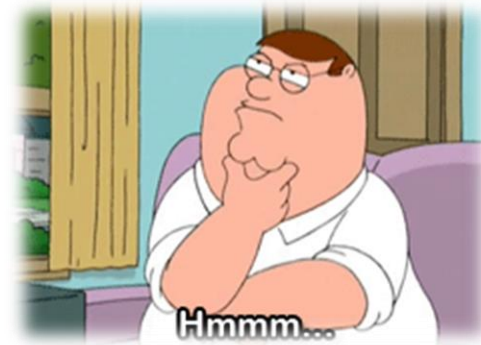
```
/* Create threads */  
#pragma omp parallel num_threads(2)  
{  
    /* Push a task in the q */  
    #pragma omp task  
    {  
        t0();  
    }  
  
    /* Push another task in the q */  
    #pragma omp task  
    {  
        t1();  
    }  
} // Implicit barrier
```





Let's make it simpler

- › Work is produced in parallel by threads
- › Work is consumed in parallel by threads
- › A lot of confusion!
 - Number of tasks grows
 - Hard to control producers
- › How to make this simpler?



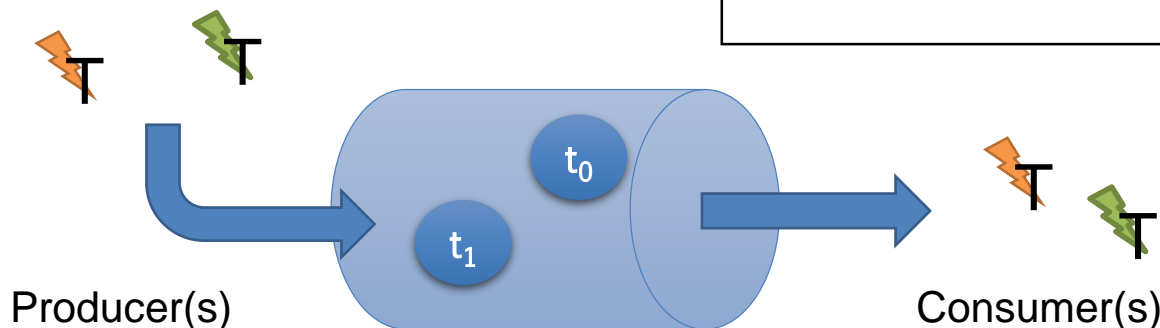
Single-producer, multiple consumers

› A paradigm! Typically preferred by programmers

- Code more understandable
- Simple
- More manageable

› How to do this?

```
/* Create threads */  
#pragma omp parallel num_threads(2)  
{  
  
    #pragma omp task  
    t0();  
  
    #pragma omp task  
    t1();  
  
} // Implicit barrier
```



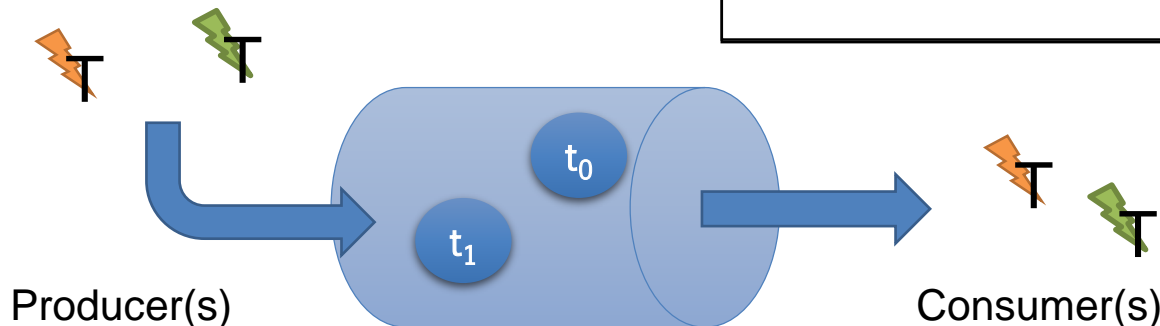
Single-producer, multiple consumers

› A paradigm! Typically preferred by programmers

- Code more understandable
- Simple
- More manageable

› How to do this?

```
/* Create threads */  
#pragma omp parallel num_threads(2)  
{  
    #pragma omp single  
    {  
        #pragma omp task  
        t0();  
  
        #pragma omp task  
        t1();  
    }  
} // Implicit barrier
```



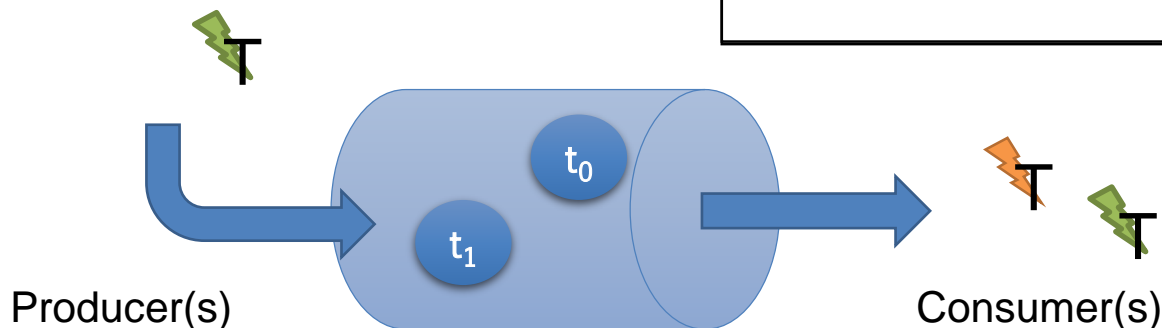
Single-producer, multiple consumers

› A paradigm! Typically preferred by programmers

- Code more understandable
- Simple
- More manageable

› How to do this?

```
/* Create threads */  
#pragma omp parallel num_threads(2)  
{  
    #pragma omp single  
    {  
        #pragma omp task  
        t0();  
  
        #pragma omp task  
        t1();  
    }  
} // Implicit barrier
```





The task directive

Can be used

> in a nested manner

- Before doing work, produce two other tasks
- Only need one parreg "outside"

> in an irregular manner

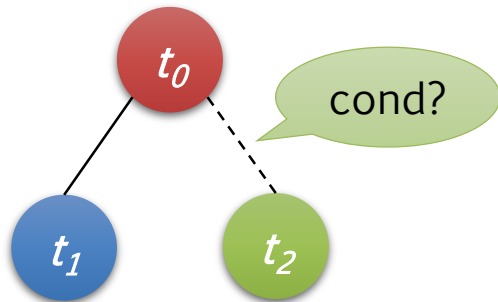
- See cond ?
- Barriers are not involved!
- Unlike parregs'

```
/* Create threads */
#pragma omp parallel num_treads(2)
{
    #pragma omp single
    {
        /* Push a task in the q */
        #pragma omp task
        {
            /* Push a (children) task in the q */
            #pragma omp task
            t1();

            /* Conditionally push task in the q */
            if(cond)
                #pragma omp task
                t2();

            /* After producing t1 and t2,
             * do some work */
            t0();
        }
    }
} // Implicit barrier
```


The task directive



- › A task graph
- › Edges are "father-son" relationships
- › Not timing/precedence!!!

```

/* Create threads */
#pragma omp parallel num_threads(2)
{
    #pragma omp single
    {
        /* Push a task in the q */
        #pragma omp task
        {
            /* Push a (children) task in the q */
            #pragma omp task
            t1();

            /* Conditionally push task in the q */
            if(cond)
                #pragma omp task
                t2();

            /* After producing t1 and t2,
             * do some work */
            t0();
        }
    }
} // Implicit barrier
    
```



It's a matter of time



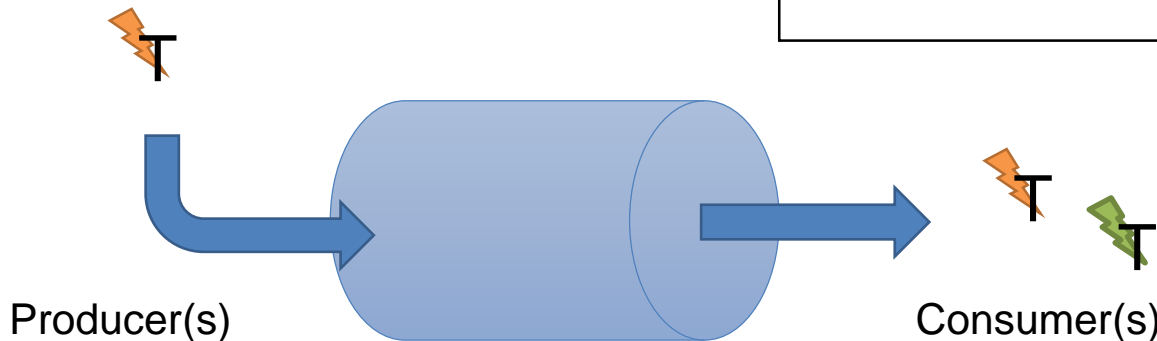
- › The `task` directive represents the **push** in the WQ
 - And the **pull**???
- › Not "where" it is in the code
 - But, **when**!
- › In OpenMP tasks, we separate **the moment in time**
 - when we produce work (push - `#pragma omp task`)
 - when we consume the work (pull - **????**)



Timing de-coupling

- › One thread produces
- › All of the thread consume
- › ..but, when???

```
/* Create threads */  
#pragma omp parallel num_threads(2)  
{  
    #pragma omp single  
    {  
        #pragma omp task  
        t0();  
  
        #pragma omp task  
        t1();  
    } // Implicit barrier  
} // Implicit barrier
```

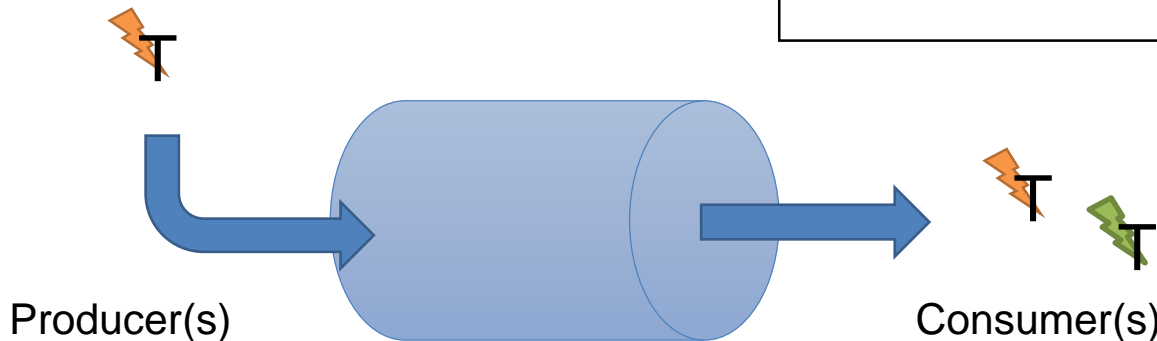




Timing de-coupling

- › One thread produces
- › All of the thread consume
- › ..but, when???

```
/* Create threads */  
#pragma omp parallel num_threads(2)  
{  
    #pragma omp single  
    {  
        ⌚ #pragma omp task  
          t0();  
  
        #pragma omp task  
          t1();  
    } // Implicit barrier  
} // Implicit barrier
```

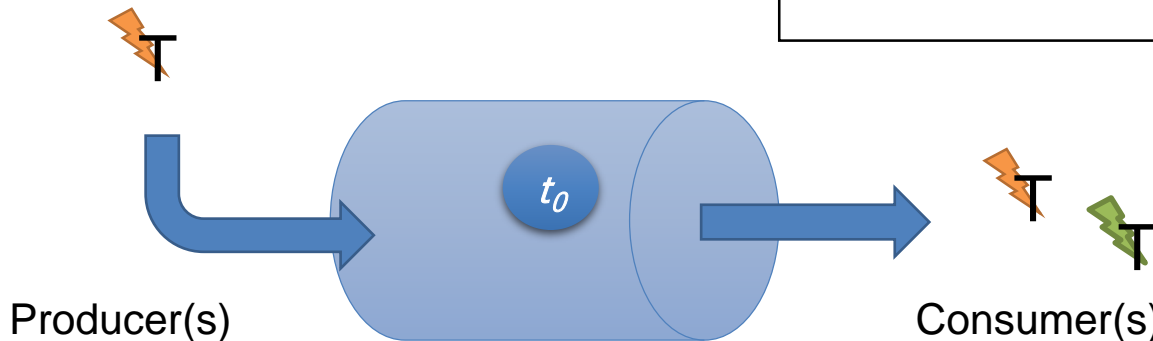




Timing de-coupling


- › One thread produces
- › All of the thread consume
- › ..but, when???

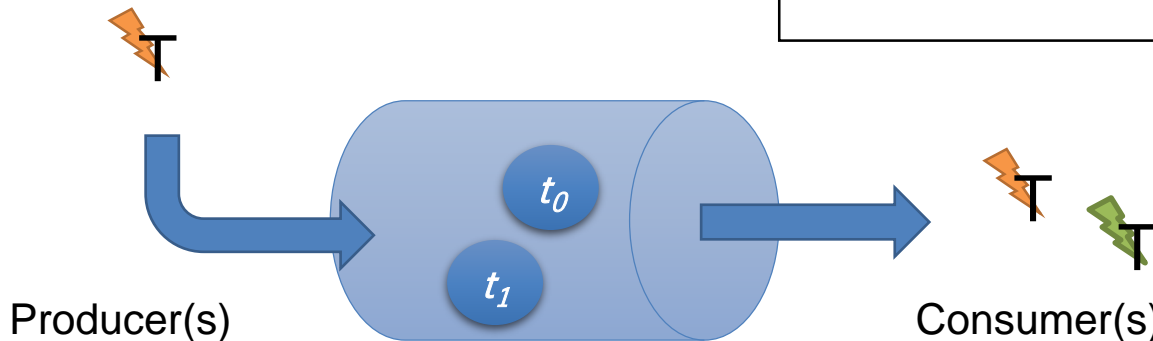
```
/* Create threads */  
#pragma omp parallel num_threads(2)  
{  
    #pragma omp single  
    {  
        ⌚ #pragma omp task  
          t0();  
  
        #pragma omp task  
          t1();  
    } // Implicit barrier  
} // Implicit barrier
```



Timing de-coupling

- › One thread produces
- › All of the thread consume
- › ..but, when???

```
/* Create threads */  
#pragma omp parallel num_threads(2)  
{  
    #pragma omp single  
    {  
        #pragma omp task  
        t0();  
  
         #pragma omp task  
        t1();  
    } // Implicit barrier  
} // Implicit barrier
```





Task Scheduling Points

- › The point when the executing thread can pull a task from the q

OMP specs

- a. the point immediately following the generation of an explicit task;
- b. after the point of completion of a task region;
- c. in a taskyield region;
- d. in a taskwait region;
- e. at the end of a taskgroup region;
- f. in an implicit and explicit barrier region;

```
/* Create threads */
#pragma omp parallel num_treads(2)
{
    #pragma omp single
    {
        #pragma omp task
        t0();

        #pragma omp task
        {
            #pragma omp task
            t2();

            t1();

            /* I just finished a task */
        }
        // I just pushed a task

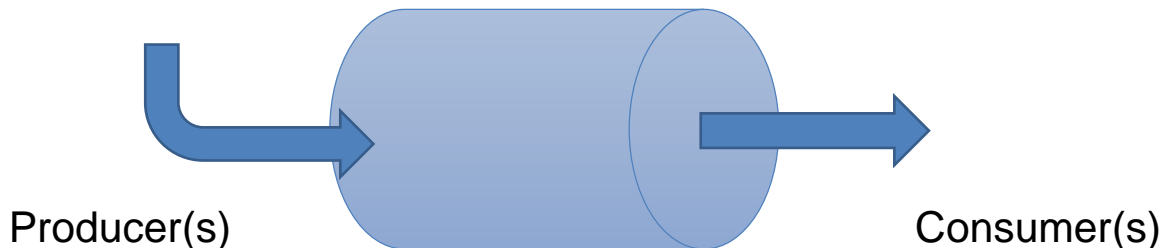
    } // Implicit barrier
} // Implicit barrier
```



Timing de-coupling

- › One thread produces
- › All of the thread consume



```
/* Create threads */  
#pragma omp parallel num_threads(2)  
{  
    #pragma omp single  
    {  
        #pragma omp task  
        t0();  
  
        #pragma omp task  
        t1();  
    } // Implicit barrier  
} // Implicit barrier
```

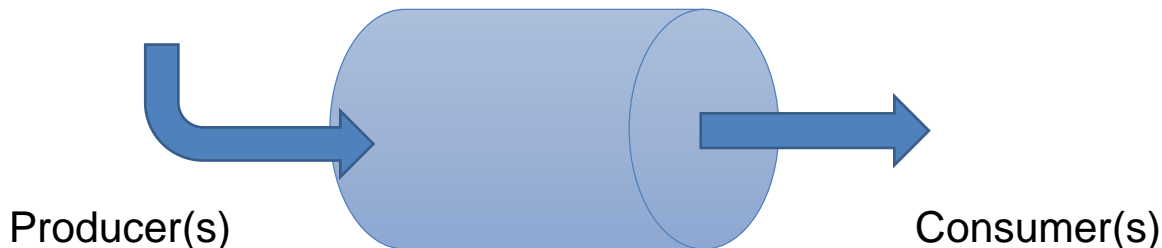




Timing de-coupling

- › One thread produces
- › All of the thread consume

```
/* Create threads */  
#pragma omp parallel num_threads(2)  
{  
    #pragma omp single  T  
    {  
        #pragma omp task  
        t0();  
  
        #pragma omp task  
        t1();  
    } // Implicit barrier   
} // Implicit barrier
```





Timing de-coupling

- › One thread produces
- › All of the thread consume

```
/* Create threads */  
#pragma omp parallel num_threads(2)  
{  
    #pragma omp single T  
    {  
        #pragma omp task  
        t0();  
  
        #pragma omp task  
        t1();  
    } // Implicit barrier T  
} // Implicit barrier f
```



Timing de-coupling

- › One thread produces
- › All of the thread consume

```
/* Create threads */  
#pragma omp parallel num_threads(2)  
{  
    #pragma omp single  
    {  
        #pragma omp task ⚡ T  
        t0();  
  
        #pragma omp task  
        t1();  
    } // Implicit barrier T  
} // Implicit barrier f
```

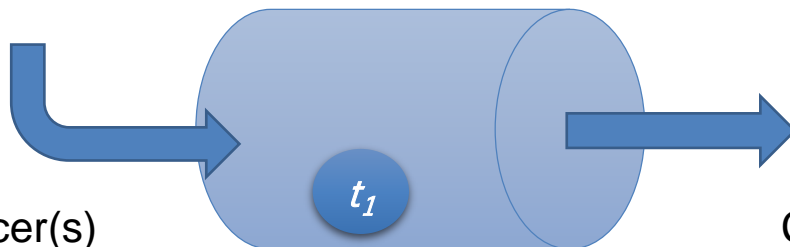




Timing de-coupling

- › One thread produces
- › All of the thread consume

```
/* Create threads */  
#pragma omp parallel num_threads(2)  
{  
    #pragma omp single  
    {  
        #pragma omp task  
        t0();  
  
        #pragma omp task  
        t1();  
    } // Implicit barrier  
} // Implicit barrier
```



Consumer(s)

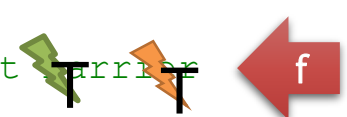
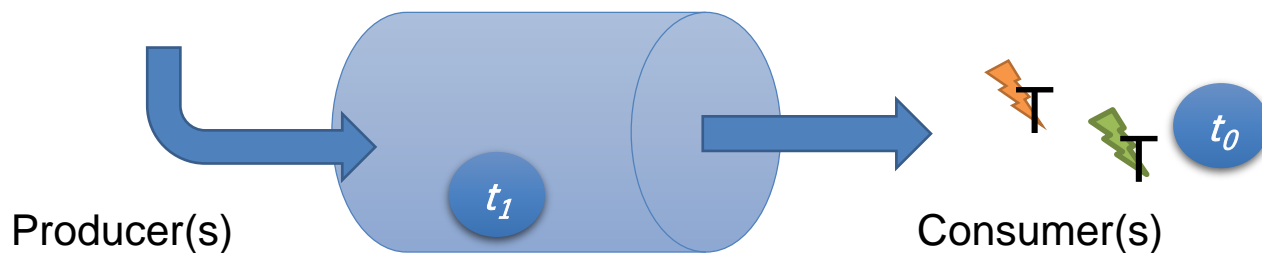
Timing de-coupling

- › One thread produces
- › All of the thread consume

```

/* Create threads */
#pragma omp parallel num_threads(2)
{
    #pragma omp single
    {
        #pragma omp task
        t0();

        #pragma omp task
        t1();
    } // Implicit barrier
} // Implicit barrier
  
```

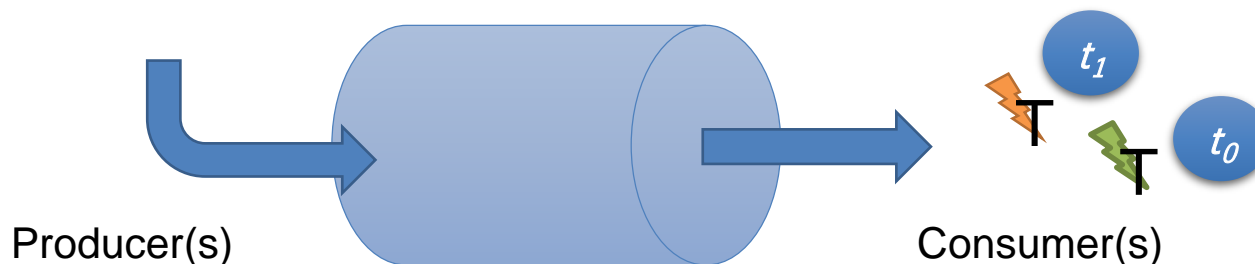
Timing de-coupling

- › One thread produces
- › All of the thread consume

```

/* Create threads */
#pragma omp parallel num_threads(2)
{
    #pragma omp single
    {
        #pragma omp task
        t0();

        #pragma omp task
        t1();
    } // Implicit barrier
} // Implicit barrier
  
```





Exercise

Let's
code!

- › Create an array of N elements
 - Put inside each array element its index, multiplied by '2'
 - `arr[0] = 0; arr[1] = 2; arr[2] = 4; ...and so on..`

- › Now, do it in parallel with a team of T threads
 - Using the `task` construct instead of `for`
 - Remember: if not specified, data sharing is unknown! (NOT SHARED)

- › Mimic `dynamic` loops semantic
 - `chunk = 1` → 1 iteration per thread
 - "Tasks made of 1 iteration"



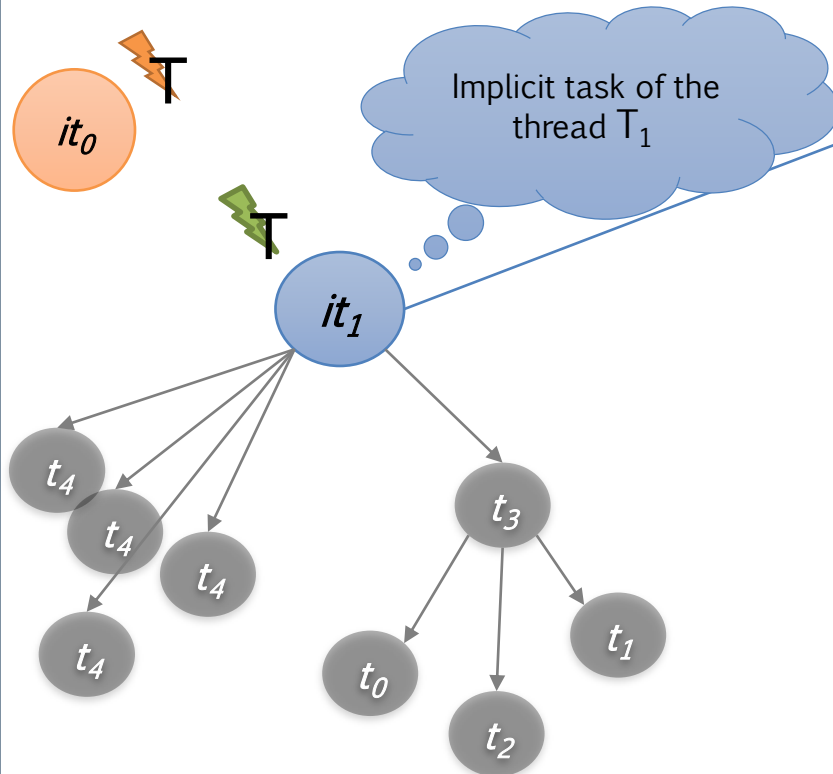
Exercise

Let's
code!

- › Create an array of N elements
 - Put inside each array element its index, multiplied by '2'
 - `arr[0] = 0; arr[1] = 2; arr[2] = 4; ...and so on..`
- › Mimic `dynamic` loops semantic
 - Now, find a way to increase chunking
 - Tasks made of `CHUNK = 1..2..4..5` iterations
 - (simple: `N = 20`)

Implicit task

- › In parregs, threads perform work
 - Called implicit task
 - One for each thread in parreg



```
#pragma omp parallel num_threads(2)
{
    #pragma omp single
    {
        for(i<10000)
            #pragma omp task
            t4_i();

        #pragma omp task
        {
            #pragma omp task
            t0();
            #pragma omp task
            t1();
            #pragma omp task
            t2();

            work();

        } // end of task
    } // end of single (bar)
} // parreg end
```



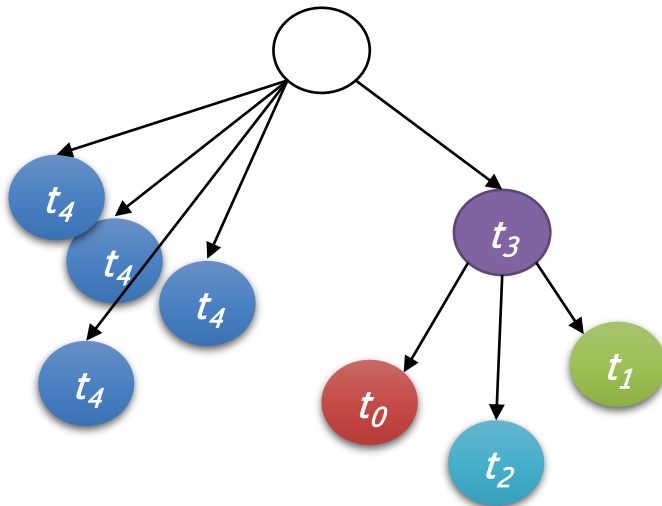
Task synchronization

- › Implicit or explicit barriers
 - Join all threads in a parreg
- › Need something lighter
 - That involves only tasks
 - That do not involve all tasks!

Wait them all?

Sometimes you don't need to..

- › t3 needs output from
 - t0
 - t1
 - t2
- › t3 doesn't need output from t4s



```

#pragma omp parallel
{
    #pragma omp single
    {
        for(i<10000)
            #pragma omp task
            t4i_work();

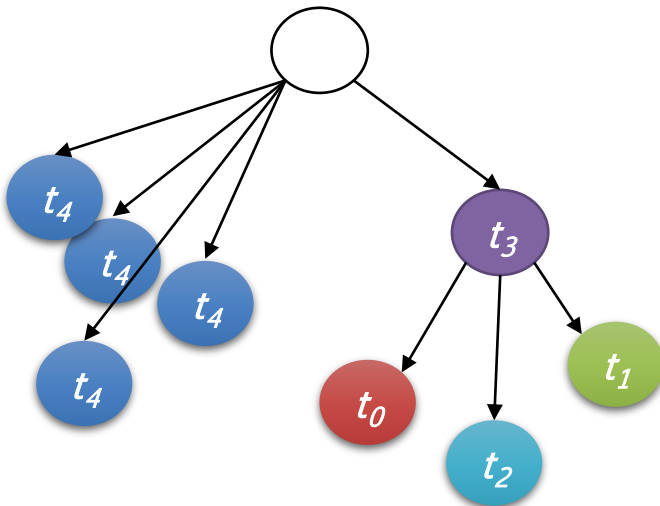
        #pragma omp task
        {
            #pragma omp task
            t0_work();
            #pragma omp task
            t1_work();
            #pragma omp task
            t2_work();

            // Requires the output of t0,
            //   t1, t2, but not of t4s
            t3_work();
        } // end of task t3
    } // bar
} // parreg end
    
```

Wait them all?

Sometimes you don't need to..

- › t3 needs output from
 - t0
 - t1
 - t2
- › t3 doesn't need output from t4s



```

#pragma omp parallel
{
  #pragma omp single
  {
    for(i<10000)
      #pragma omp task
      t4i_work();

    #pragma omp task
    {
      #pragma omp task
      t0_work();
      #pragma omp task
      t1_work();
      #pragma omp task
      t2_work();

      #pragma omp taskgroup
      {
        // Requires the output of t0,
        //   t1, t2, but not of t4s
        t3_work();
      }
    } // end of task t3
  } // bar
} // parreg end
  
```



The taskgroup directive

```
#pragma omp taskgroup
```

Standalone directive

› Wait on the completion of children tasks, and their descendants

› Implicit TSP

OMP specs

- a. the point immediately following the generation of an explicit task;
- b. after the point of completion of a task region;
- c. in a `taskyield` region;
- d. in a `taskwait` region;
- e. at the end of a `taskgroup` region;
- f. in an implicit and explicit barrier region;



The taskwait directive

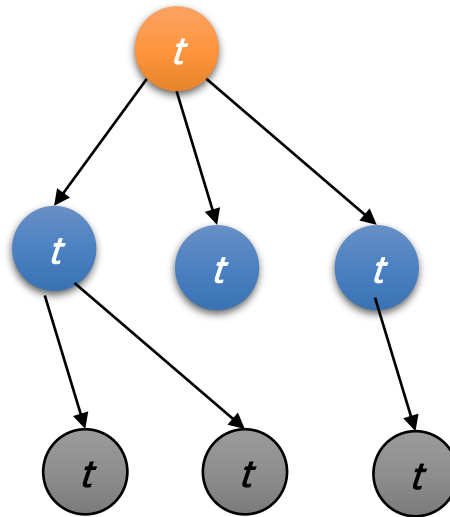
```
#pragma omp taskwait
```

Standalone directive

› Implicit TSP

› Strangely..

– Older than taskgroup



OMP specs

- a. the point immediately following the generation of an explicit task;
- b. after the point of completion of a task region;
- c. in a taskyield region;
- d. in a taskwait region;
- e. at the end of a taskgroup region;
- f. in an implicit and explicit barrier region;



The taskyield directive

```
#pragma omp taskyield
```

Standalone directive

› Explicit TSP

- Extracts (and exec) one task from the queue

OMP specs

- the point immediately following the generation of an explicit task;
- after the point of completion of a task region;
- in a `taskyield` region;
- in a `taskwait` region;
- at the end of a `taskgroup` region;
- in an implicit and explicit barrier region;



How to run the examples

Let's
code!

› Download the Code/ folder from the course website

› Compile

› `$ gcc -fopenmp code.c -o code`

› Run (Unix/Linux)

`$./code`

› Run (Win/Cygwin)

`$./code.exe`

References



- › "Calcolo parallelo" website
 - http://hipert.unimore.it/people/paolob/pub/Calcolo_Parallelo/

- › My contacts
 - paolo.burgio@unimore.it
 - <http://hipert.mat.unimore.it/people/paolob/>

- › Useful links
 - <http://www.openmp.org>
 - <http://www.google.com>