

# Tasking in OpenMP

---

---

Paolo Burgio

paolo.burgio@unimore.it



# Outline

---

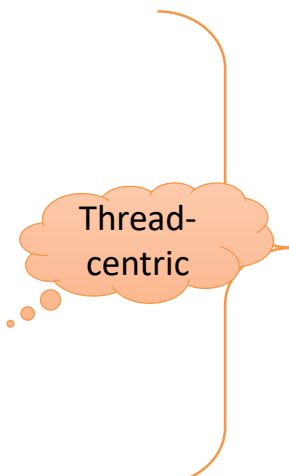
---

- › 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



Regular, loop-based parallelism



Irregular, parallelism → tasking



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

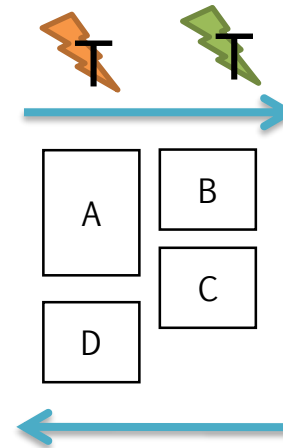
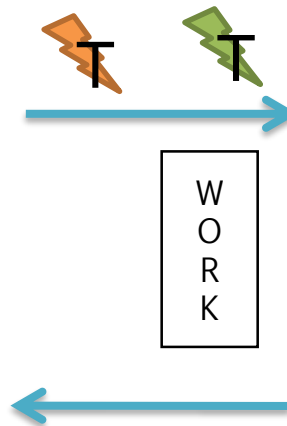
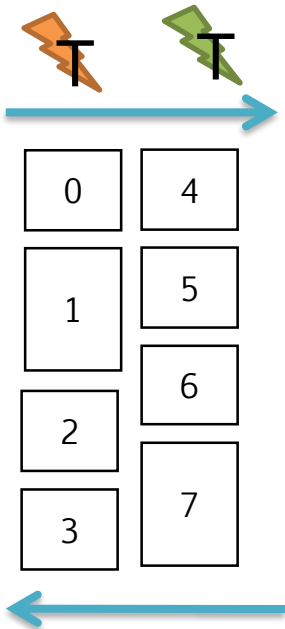


# OpenMP programming patterns

```
#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(); }
}
```





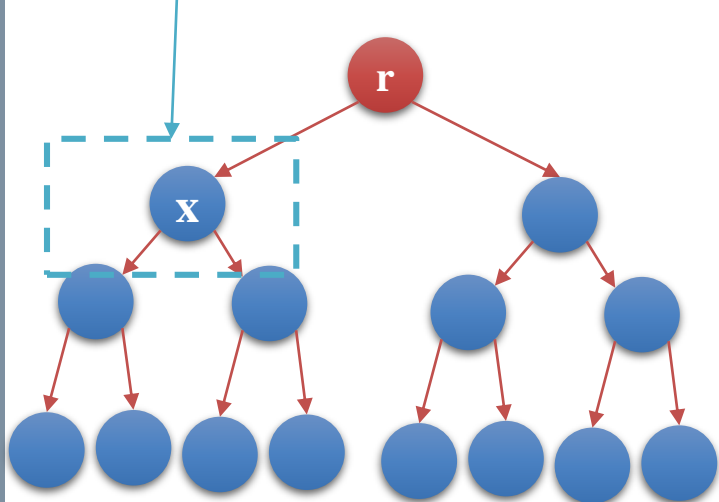
# Exercise

Let's  
code!

## > Traverse a tree

- Perform the same operation on all elements
- Download sample code

## > Recursive

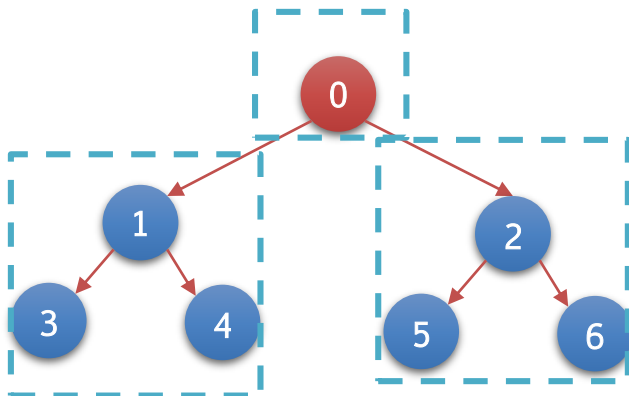




# Exercise

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);
```



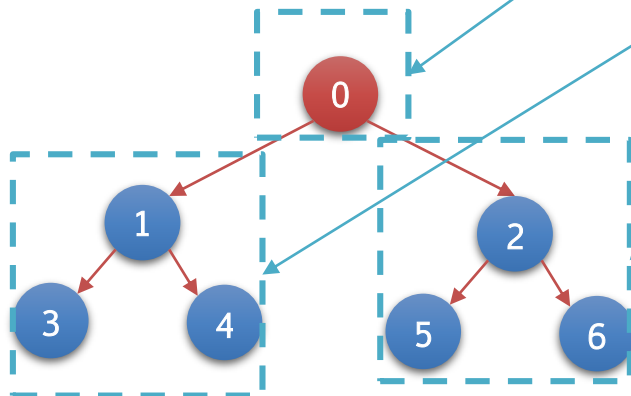
# Solved: traversing a tree in parallel

## > 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);
```

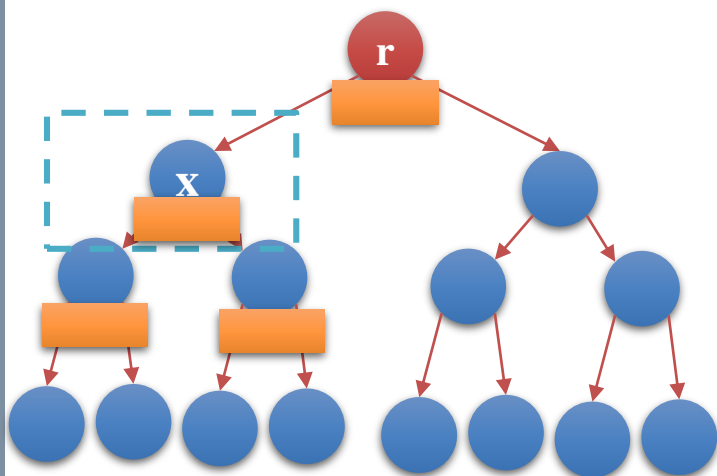


# Catches (1)

- > 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

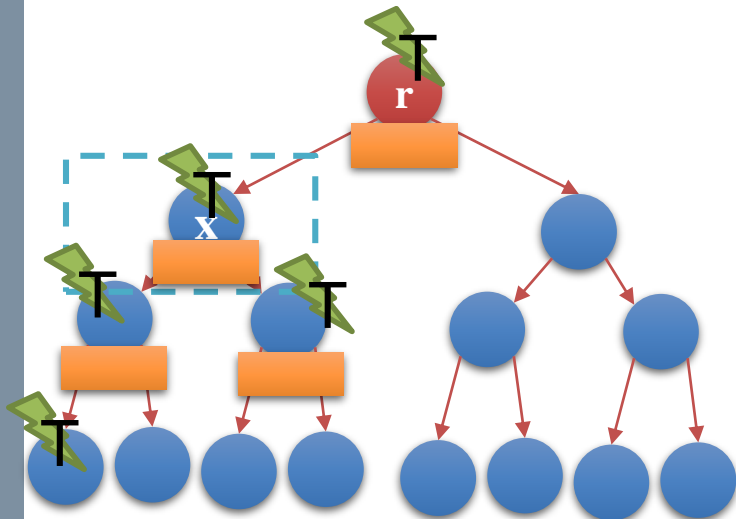
...
traverse_tree(root);
```





# Catches (2)

- > #threads grows exponentially
  - Harder to manage



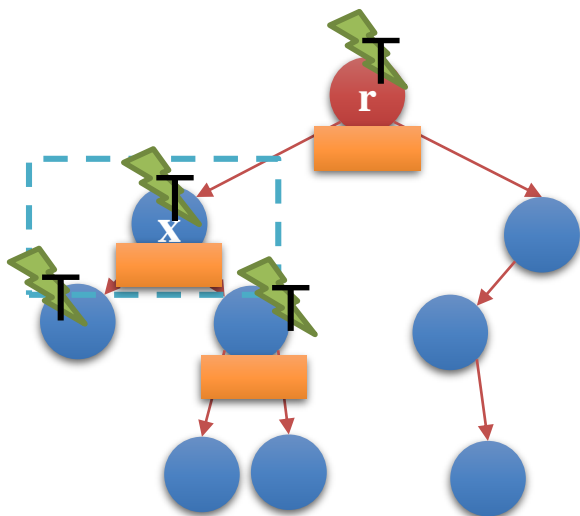
```
void traverse_tree(node_t *n)
{
    doYourWork(n);

    #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
} // Prrg 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

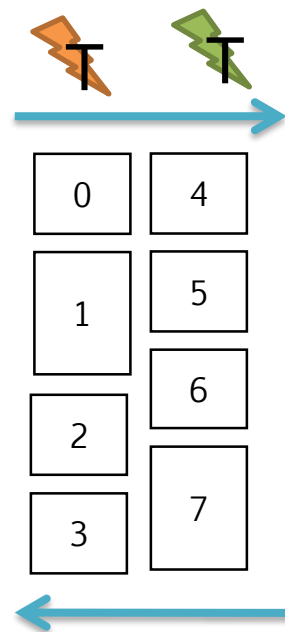
- › The number of threads explodes and it's harder to manage
  - Parreg => create new threads

# 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++)
{
    // ...
}
```

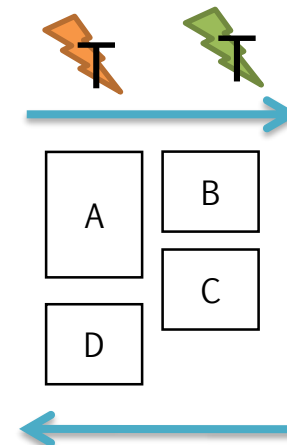


# Limitations of "traditional" WS

## 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

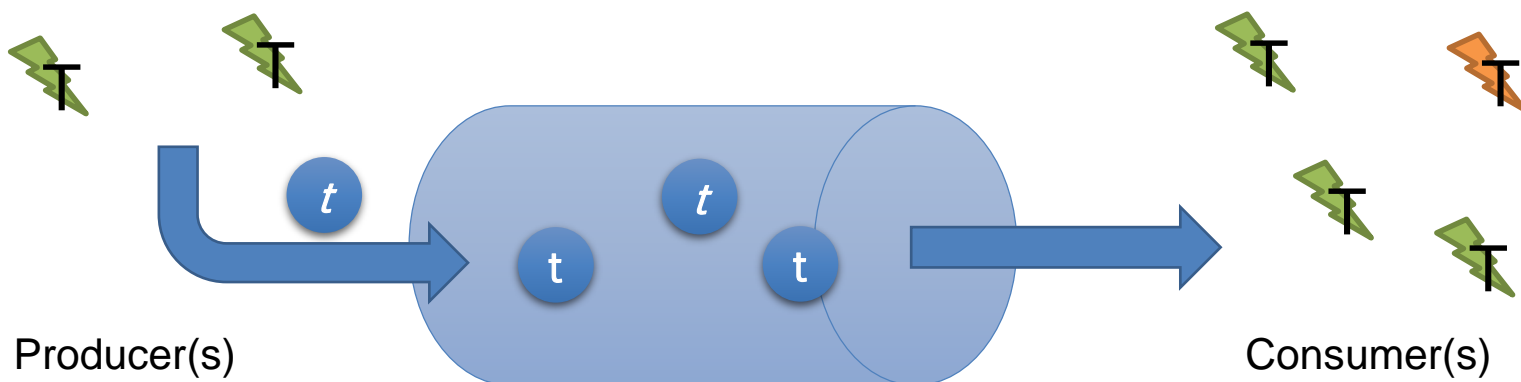
```
#pragma omp sections
{
  #pragma omp section
  { A(); }
  #pragma omp section
  { B(); }
  #pragma omp section
  { C(); }
  #pragma omp section
  { D(); }
}
```



# 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!!!!**



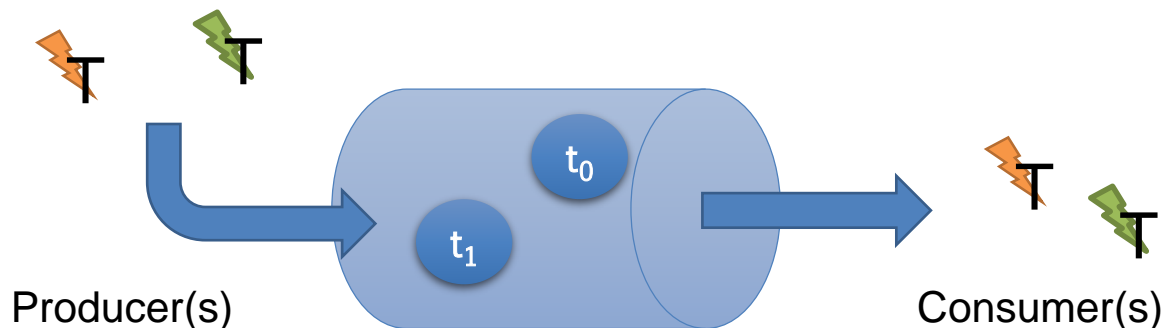
# Two sides

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

Let's  
code!

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

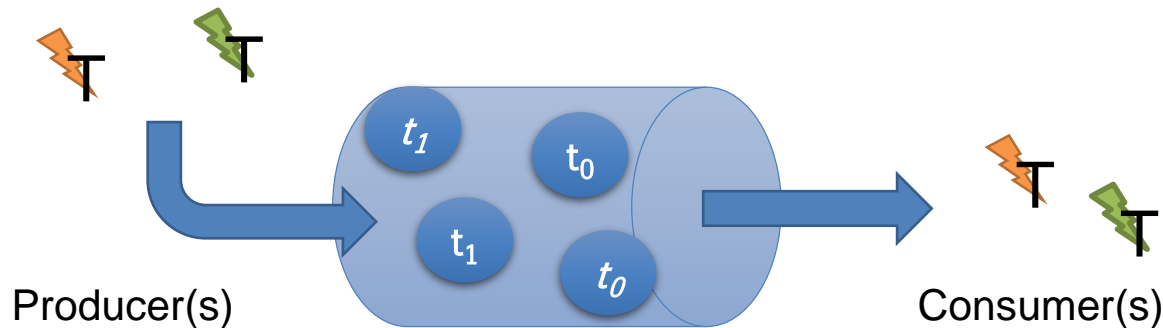
```
/* Create threads */  
#pragma omp parallel num_threads(2)  
{  
    /* Push a task in the q */  
    #pragma omp task  
    {  
        t0();  
    }  
  
    /* Push another task in the q */  
    #pragma omp task  
        t1();  
} // Implicit barrier
```



# I cheated a bit

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

```
/* Create threads */  
#pragma omp parallel num_treads(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?



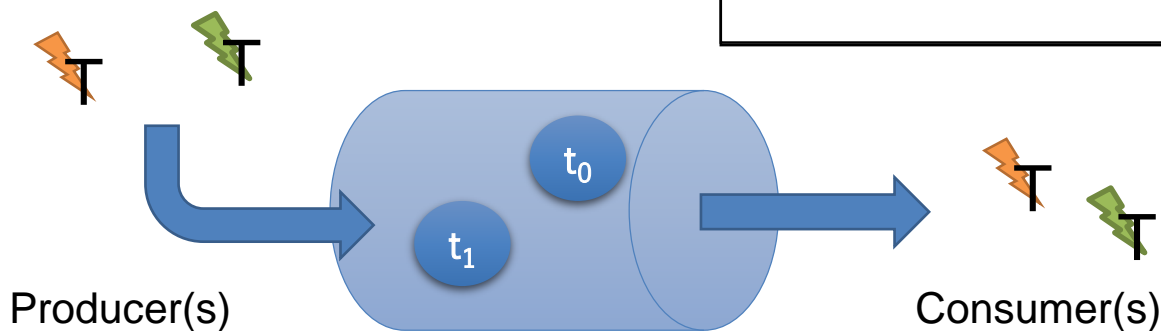
# Single-producer, multiple consumers

> 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 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

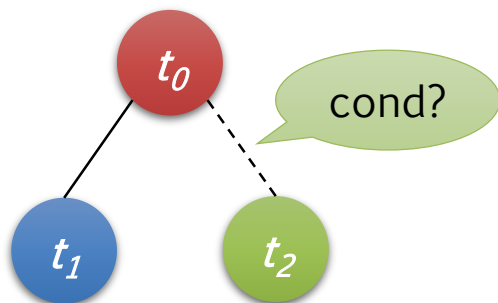
- See cond ?
- Barriers are not involved!
- Unlike parregs'

```
/* Create threads */
#pragma omp parallel num_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
```

# 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

---

---

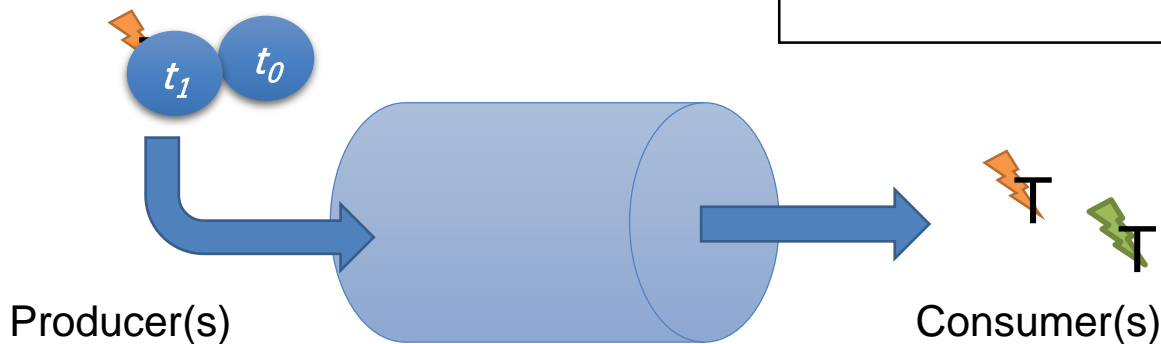


- › The `task` directive represents the **push** in the WQ
  - And the **pull**???
  
- › Not "where" it is in the code
  - But, **when**!
  
- › In OpenMP tasks, we separate **the moment in time**
  - when we produce work (push - `#pragma omp task`)
  - when we consume the work (pull - **????**)

# 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
```







# Task Scheduling Points

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

## 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;

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

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

            t1();

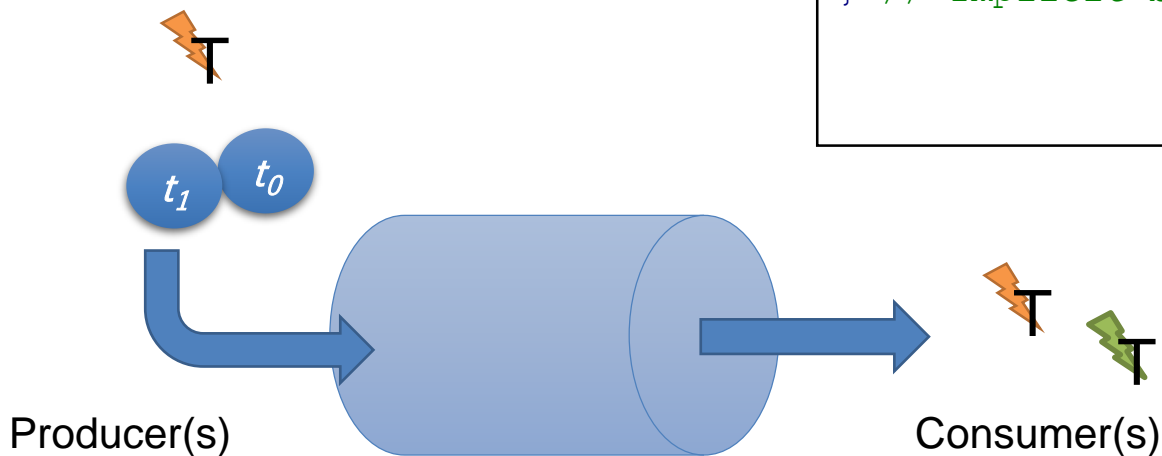
            /* I just finished a task */
        }
        // I just pushed a task

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

# Timing de-coupling

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

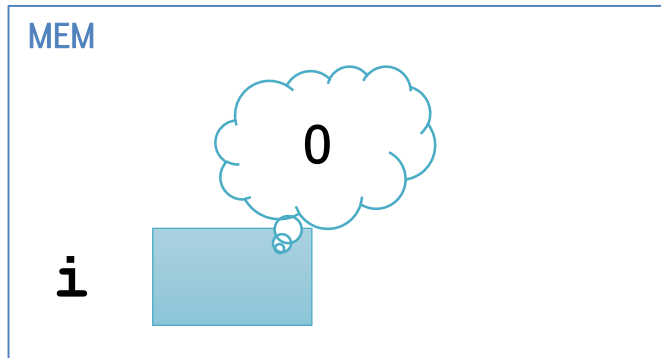
```
/* Create threads */  
#pragma omp parallel num_threads(2)  
{  
  #pragma omp single ⚡ T  
  {  
    #pragma omp task  
    t0();  
  
    #pragma omp task  
    t1();  
  } // Implicit barrier ⚡ T ← f  
} // Implicit barrier
```






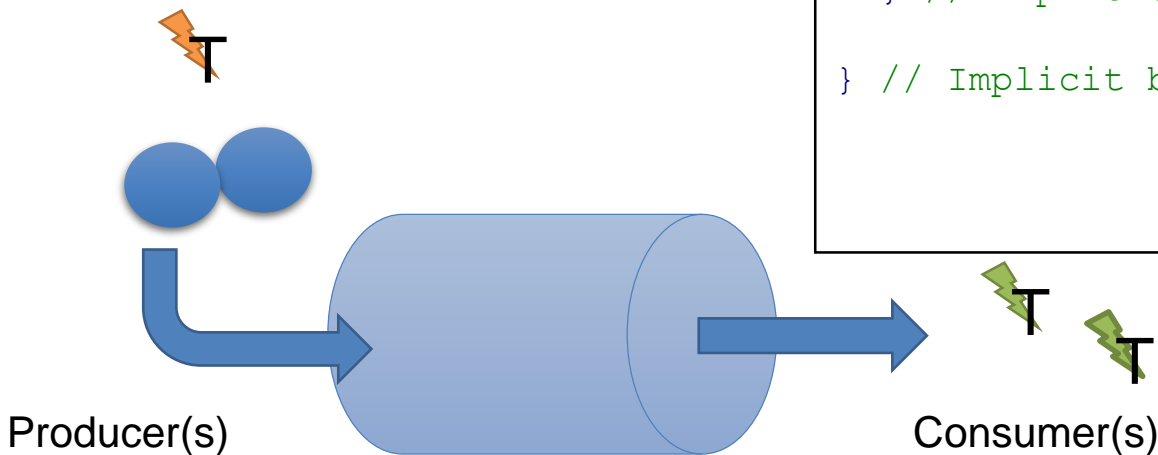


# Data changes over time!

> If  $i$  were shared



```
/* Create threads */  
#pragma omp parallel  
{  
  #pragma omp single  T  
  {  
    for(i<10000)  
      #pragma omp task shared(i)  
        work_on(i);  
  } // Implicit  barrier  f  
} // 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

---

---

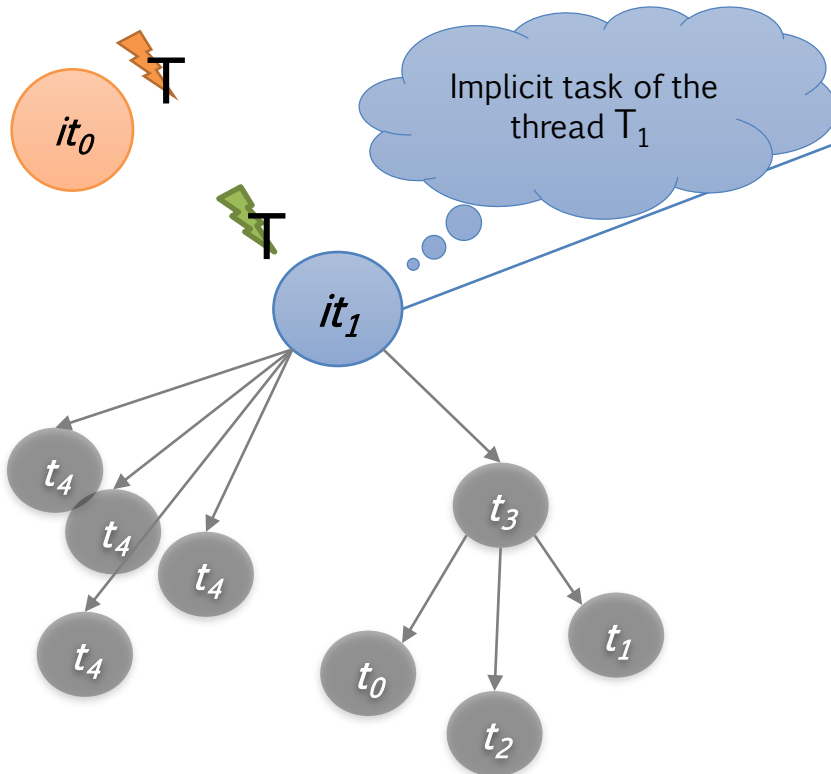
Let's  
code!

- › 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();

      t3();
    } // end of task
  } // end_of_single (bar)
} // parreg end
```



# Task synchronization

---

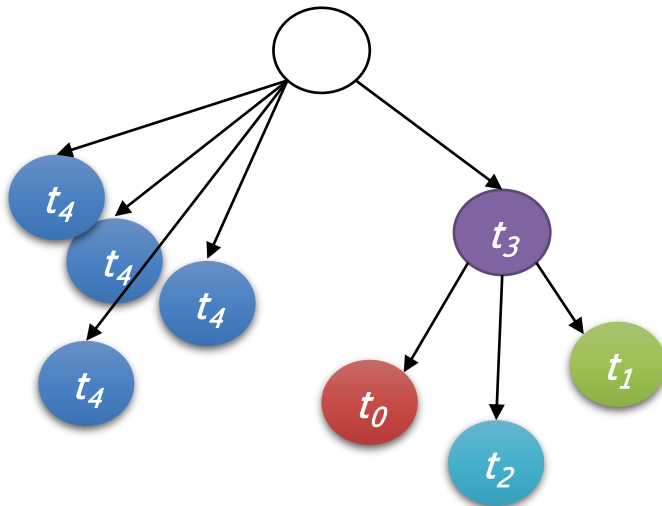
---

- › Implicit or explicit barriers
  - Join all threads in a parrreg
  
- › 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();

      #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

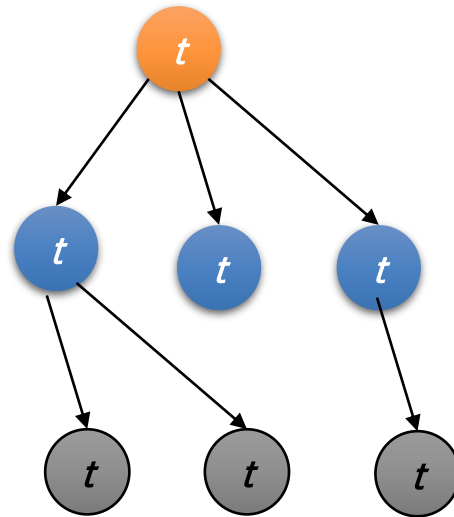
```
#pragma omp taskwait
```

Standalone directive

› Implicit TSP

› Strangely..

- Older than `taskgroup`



## OMP specs

- 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;



# 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;



# Tied vs. untied tasks

`taskyield` → interrupt a task

- > ...and what about resuming?
- > «Pre-empt»

Two possibilities

- > Must resume on the same thread
  - Data context preserved
  - No migration, faster
  - **tied** tasks (default)
- > Can resume on another thread
  - Must migrate context
  - Better work balancing
  - **untied** tasks (explicit clause)

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

        #pragma omp task [tied] | untied
        {
            #pragma omp task
                t0_work();
            #pragma omp task
                t1_work();
            #pragma omp task
                t2_work();

            // Pause t3, and pull/exec
            // another task from pool
            #pragma omp taskyield

            // Resume t3.
            t3();

        } // end of task t3
    } // bar
} // parreg end
```

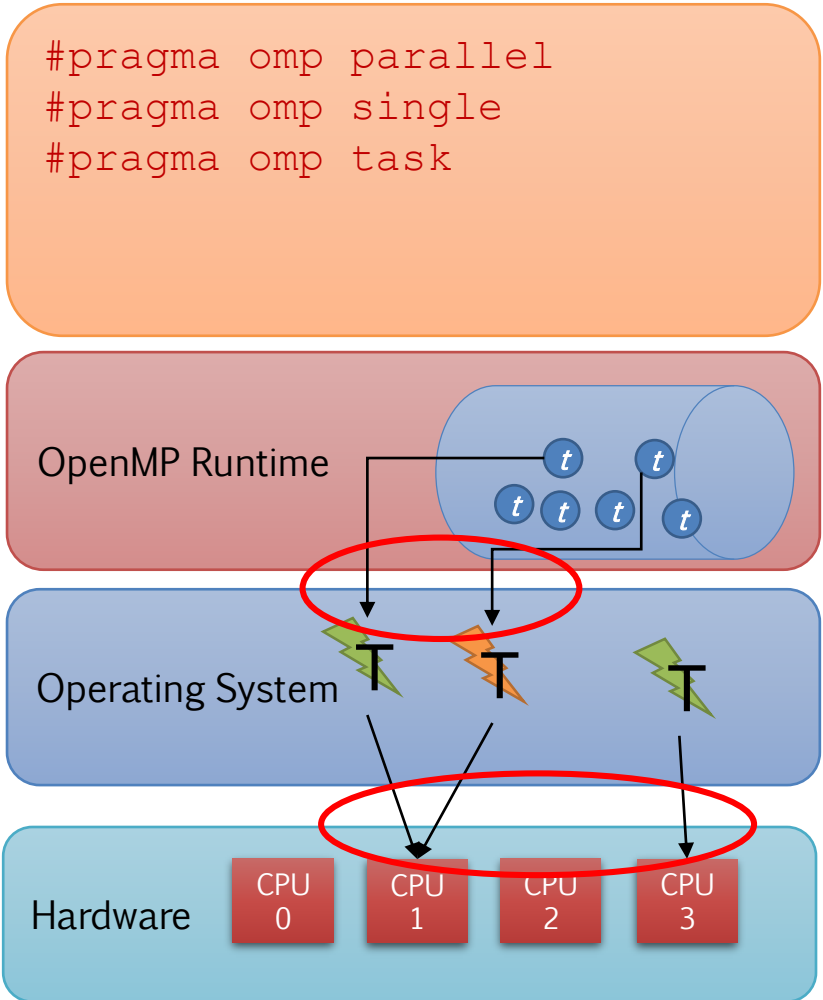
# Generic parallel stack

› For instance, OpenMP lets us specifying

- Threads in parregs
- Tasks in a parreg

› There are two scheduling levels

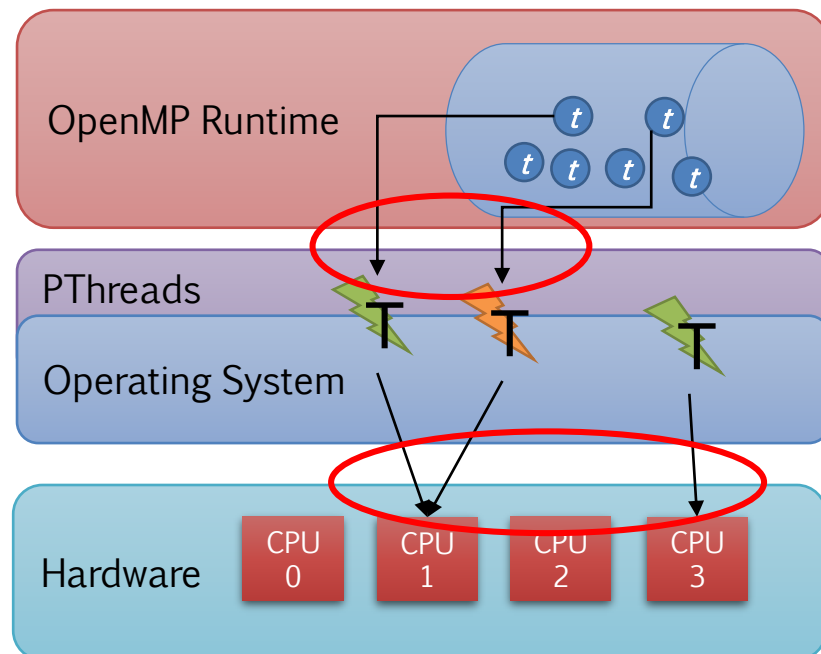
- Tasks -> Threads
- Threads -> Cores



# Example #1: OMP + GNU/Linux

- › Example of parallel stack
  - OMP runtime (e.g., GCC-OMP)
  - GNU/Linux w/Pthreads
  - 4 cores

```
#pragma omp parallel  
#pragma omp single  
#pragma omp task
```





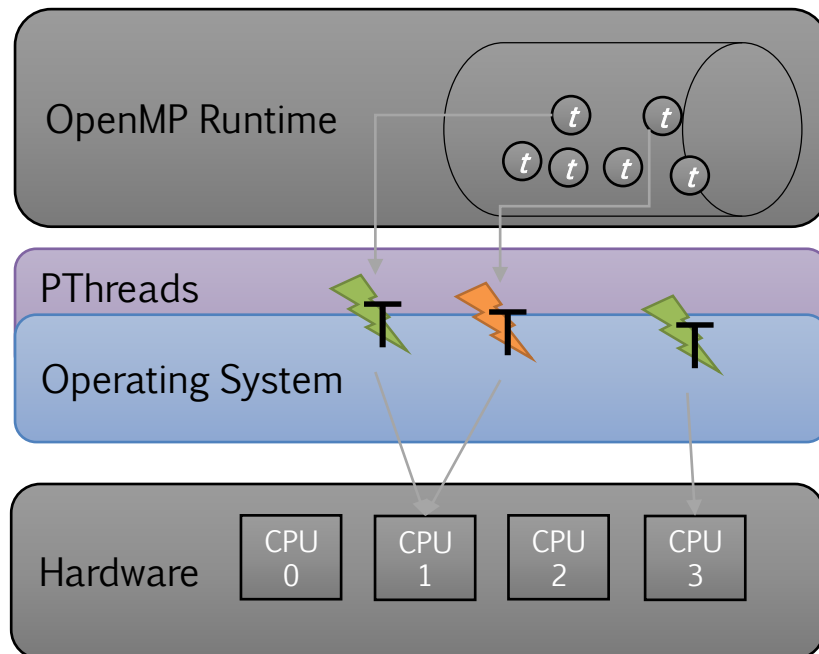
# OpenMP-to-OS threading

## > OpenMP runtime creates "OMP threads"

- Leveraging on OS threads
- E.g., Pthreads
- OMP threads <-> Pthread

Let's  
see  
this!

```
#pragma omp parallel  
#pragma omp single  
#pragma omp task
```

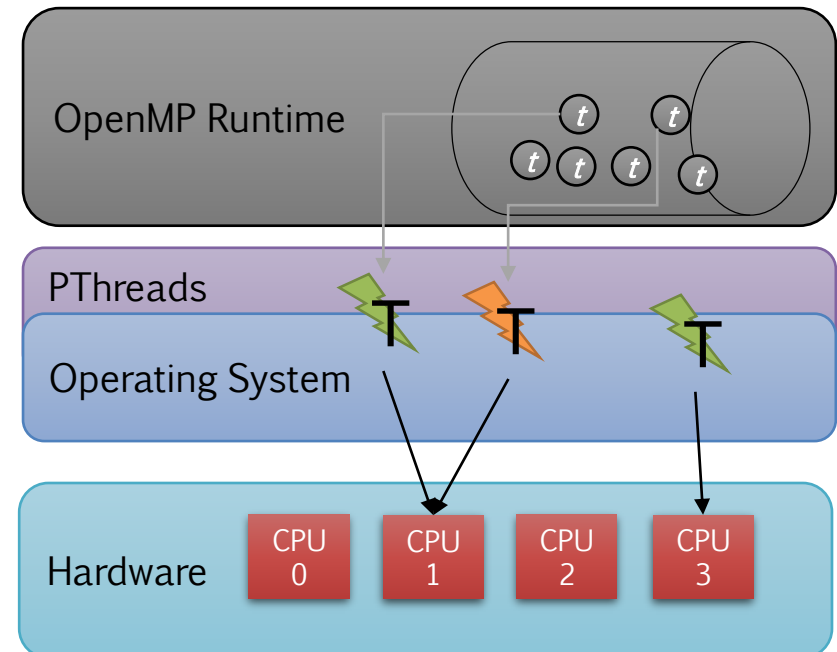




# Thread scheduling

- › OMP threads become OS threads
  - E.g., Pthreads
- › Linux has his own thread scheduling policies
- › Pthreads layer adds its own!
- › Runtime+App have poor control!
  - Only `proc_bind` clause
  - NO notion of **priority**

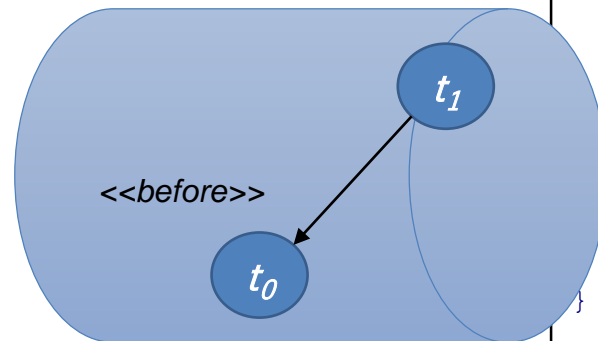
```
#pragma omp parallel  
#pragma omp single  
#pragma omp task
```





# A problem

- > A system with two tasks
  - $t_0$  created before  $t_1$
  - $t_1$  must execute before  $t_0$
  - Also on single-core system
- > ...sounds familiar?



```

/* Lock var */
omp_lock_t lock;

#pragma omp parallel
{
  #pragma omp single
  {
    /* Init as "locked" */
    omp_init_lock(&lock);
    omp_set_lock(&lock);

    #pragma omp task
    {
      /* Immediately wait */
      omp_set_lock(&lock);

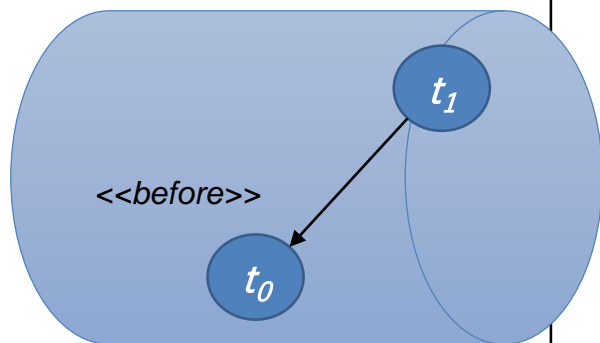
      t0();
    }

    #pragma omp task
    {
      t1();
      /* Done: release */
      omp_unset_lock(&lock);
    }
  } // bar&TSP
} // parreg end

```

# A further problem

- > A system with two tasks
  - $t_0$  created before  $t_1$
  - $t_1$  must execute before  $t_0$
  - Also on single-core system
  
- > Thread blocked on  $t_0$  will never work on  $t_1$ 
  - Not enough parallelism in the machine



```

/* Lock var */
omp_lock_t lock;

#pragma omp parallel
{
  #pragma omp single
  {
    /* Init as "locked" */
    omp_init_lock(&lock);
    omp_set_lock(&lock);

    #pragma omp task
    {
      /* Immediately wait */
      omp_set_lock(&lock);

      t0();
    }

    #pragma omp task
    {
      t1();
      /* Done: release */
      omp_unset_lock(&lock);
    }
  } // bar&TSP
// parreg end

```



# A further problem

- > `omp_set_lock` is blocking, but
  - There can be thread switch
  - There is NO task switch
- > Thread is stuck
- > In single-thread systems, dead

```

/* Lock var */
omp_lock_t lock;

#pragma omp parallel
{
    /* single
    ...
    s "locked" */
    lock(&lock);
    lock(&lock);
}

#pragma omp task
/* Immediately wait */
omp_set_lock(&lock);

t0();
}

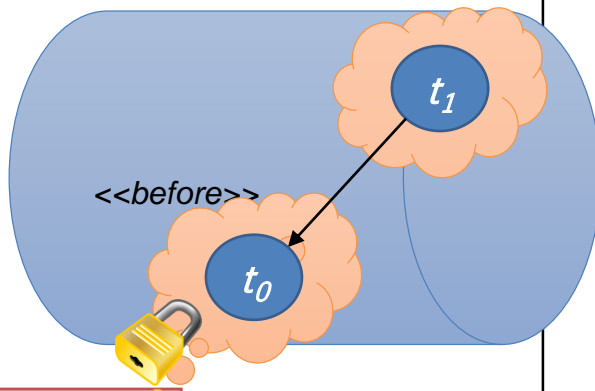
#pragma omp task
{
    t1();
    /* Done: release */
    omp_unset_lock(&lock);
}
} // bar&TSP
} // parreg end

```

```

/* Wait + TSP */
while(!omp_test_lock(&lock))
{
    #pragma omp taskyield
}

```

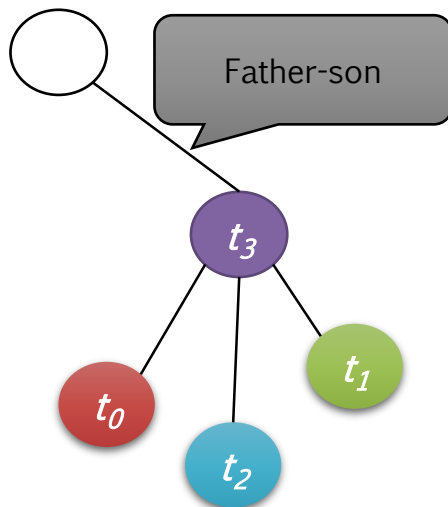




# ..but we want more!

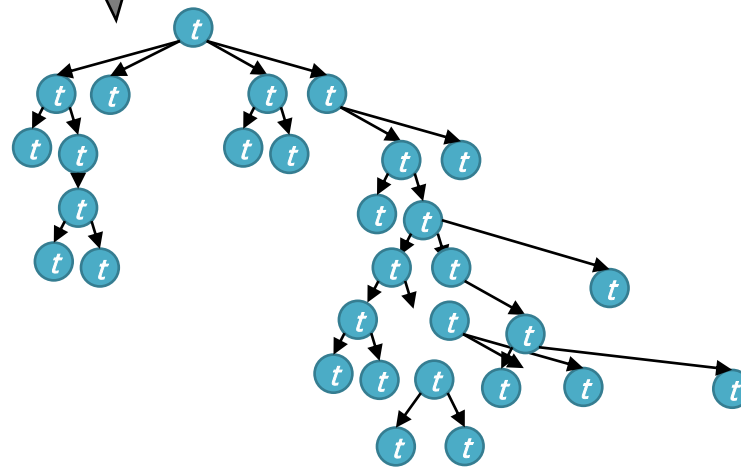
```
#pragma omp parallel
#pragma omp single
{
    #pragma omp task
    {
        #pragma omp task
        t0();
        #pragma omp task
        t1();
        #pragma omp task
        t2();
        t3();
    } // end of task
} // parreg end
```

› Instead of this...



Timing precedence

› Want to express this





# The depend clause

---

---

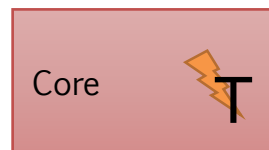
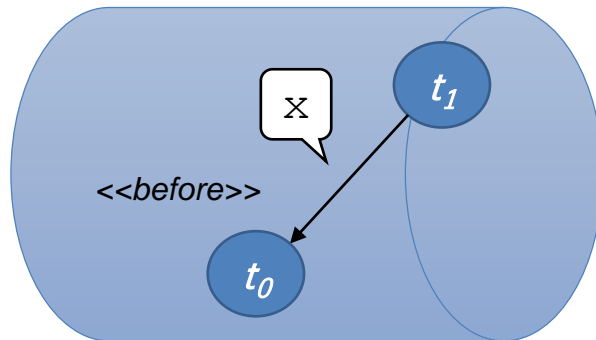
```
#pragma omp task [clause [,] clause]... new-line  
structured-block
```

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)
```

# The depend clause

- › Set a variable to act as **placeholder** for the dependency



```
#pragma omp parallel
{
  #pragma omp single
  {
    // Dependency is represented as a var
    int x = 0;

    #pragma omp task depend(in:x)
    {
      t0();
    }

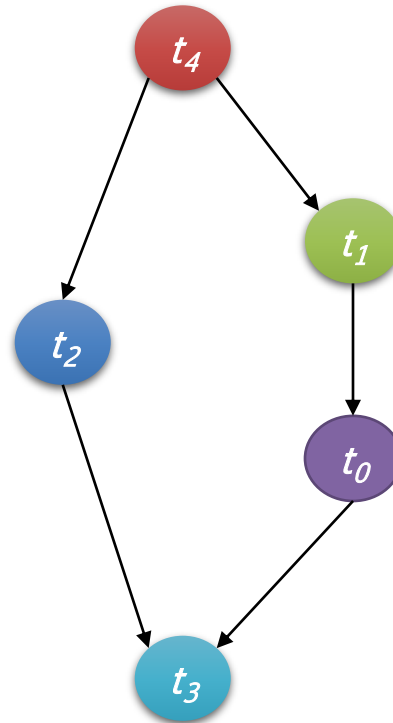
    #pragma omp task depend(out:x)
    {
      t1();
    }
  } // bar&TSO
} // parreg end
```

# Exercise

Let's  
code!

## > Implement this DAG using Task dependencies

- Tasks must be created in order of index
- 0, 1, 2, 3, 4



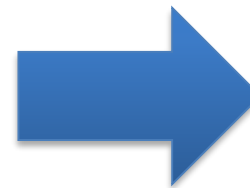


# OpenMP is just a language

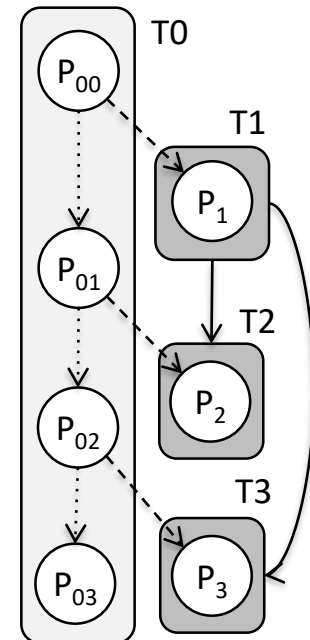
- › At the end, OpenMP can be seen as a **mechanism!**
  - Still, good, but let's move on!
- › Use OpenMP to express complex graphs
- › For the sake of completeness, i cheated a bit
  - Also an OpenMP task can be decomposed

```
#pragma omp parallel
#pragma omp single // task T0
P00 (x=0; y=0;)
// task T1
#pragma omp task depend(out:x,y) { P1 }
P01
// task T2
#pragma omp task depend(in:x) { P2 }
P02
// task T3
#pragma omp task depend(in:y) { P3 }
P03
```

*task-parts*



**OpenMP-  
DAG**





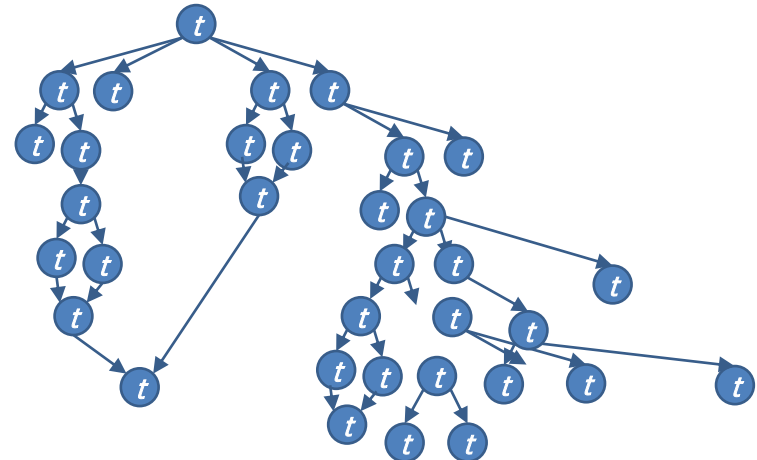
# Example: DAGs

---

---

## Directed Acyclic Graphs

- › Nodes are in a parent-children relationship, no cycles (back arcs)
  - No loops!
- › As opposite to, program as a "full" graph
  - Basic blocks in compiler internals
  - In some situations, preferable for representing the flow of a program





# 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://algo.ing.unimo.it/people/andrea/Didattica/HPC/>
  
- › My contacts
  - [paolo.burgio@unimore.it](mailto:paolo.burgio@unimore.it)
  - <http://hipert.mat.unimore.it/people/paolob/>
  
- › Useful links
  - <http://www.openmp.org>
  - <http://www.google.com>