

OpenMP dynamic loops

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



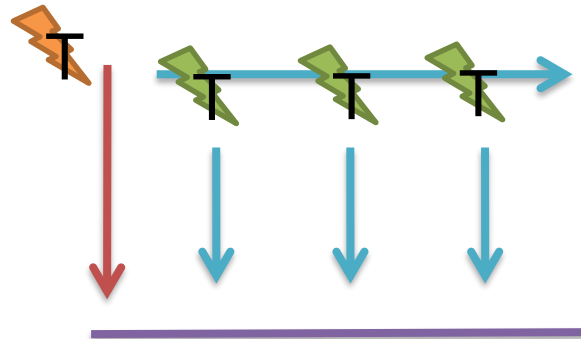
Let's talk about performance

- › We already saw how parallelism \neq performance
 - Example: a loop
 - If one thread is delayed, it prevents other threads to do useful work!!

```
#pragma omp parallel num_threads(4)
{
    #pragma omp for
    for(int i=0; i<N; i++)
    {
        ...

    } // (implicit) barrier

    // USEFUL WORK!!
} // (implicit) barrier
```





Let's talk about performance

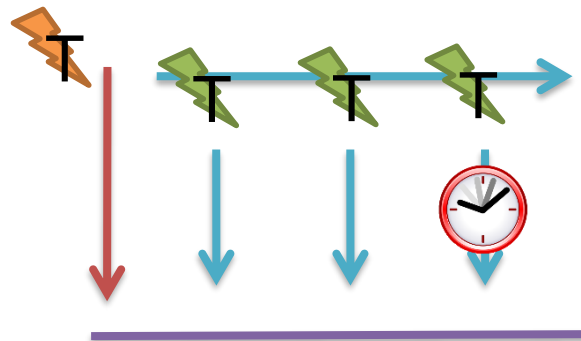
- › We already saw how parallelism \neq performance
 - Example: a loop
 - If one thread is delayed, it prevents other threads to do useful work!!

```
#pragma omp parallel num_threads(4)
{
    #pragma omp for
    for(int i=0; i<N; i++)
    {
        ...

    } // (implicit) barrier

    // USEFUL WORK!!

} // (implicit) barrier
```





Let's talk about performance

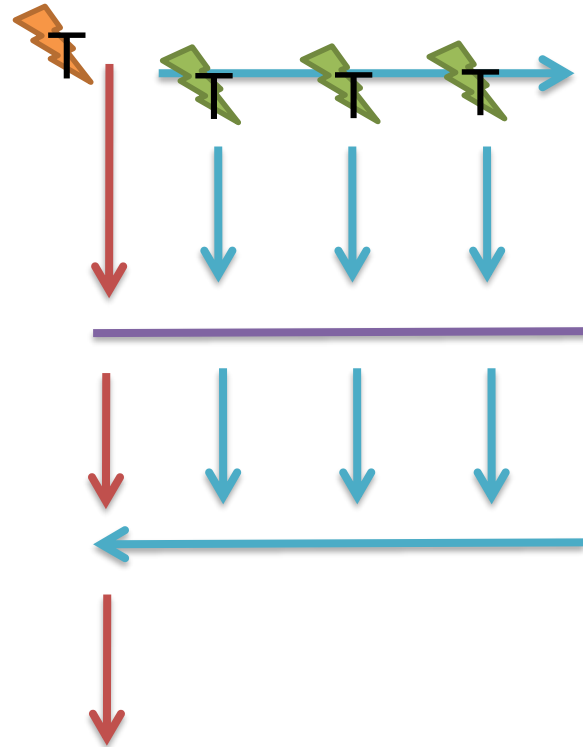
- › We already saw how parallelism \neq performance
 - Example: a loop
 - If one thread is delayed, it prevents other threads to do useful work!!

```
#pragma omp parallel num_threads(4)
{
    #pragma omp for
    for(int i=0; i<N; i++)
    {
        ...

    } // (implicit) barrier

    // USEFUL WORK!!

} // (implicit) barrier
```





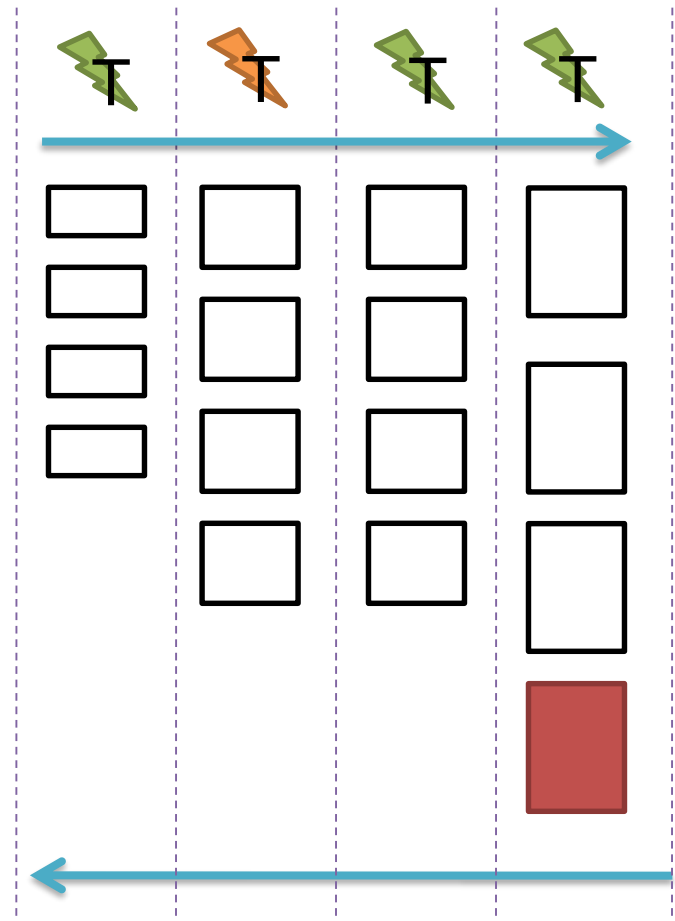
Unbalanced loop partitioning

- › Iterations are statically assigned before entering the loop
 - Might not be effective nor efficient

```
#pragma omp parallel for num_threads (4)
for (int i=0; i<16; i++)
{

    /* UNBALANCED LOOP CODE */

} /* (implicit) Barrier */
```





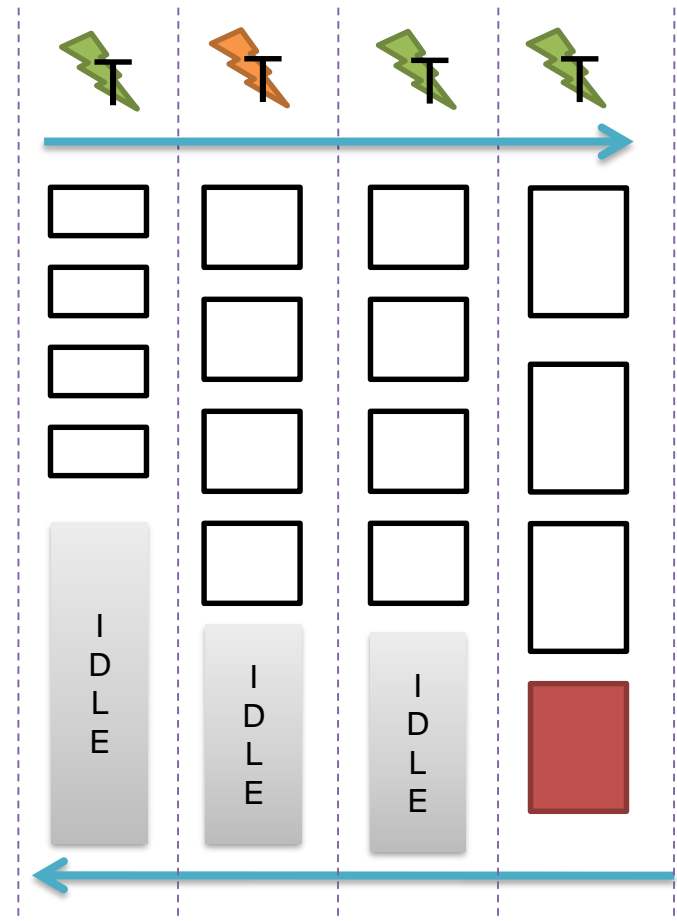
Unbalanced loop partitioning

- › Iterations are statically assigned before entering the loop
 - Might not be effective nor efficient

```
#pragma omp parallel for num_threads (4)
for (int i=0; i<16; i++)
{

    /* UNBALANCED LOOP CODE */

} /* (implicit) Barrier */
```





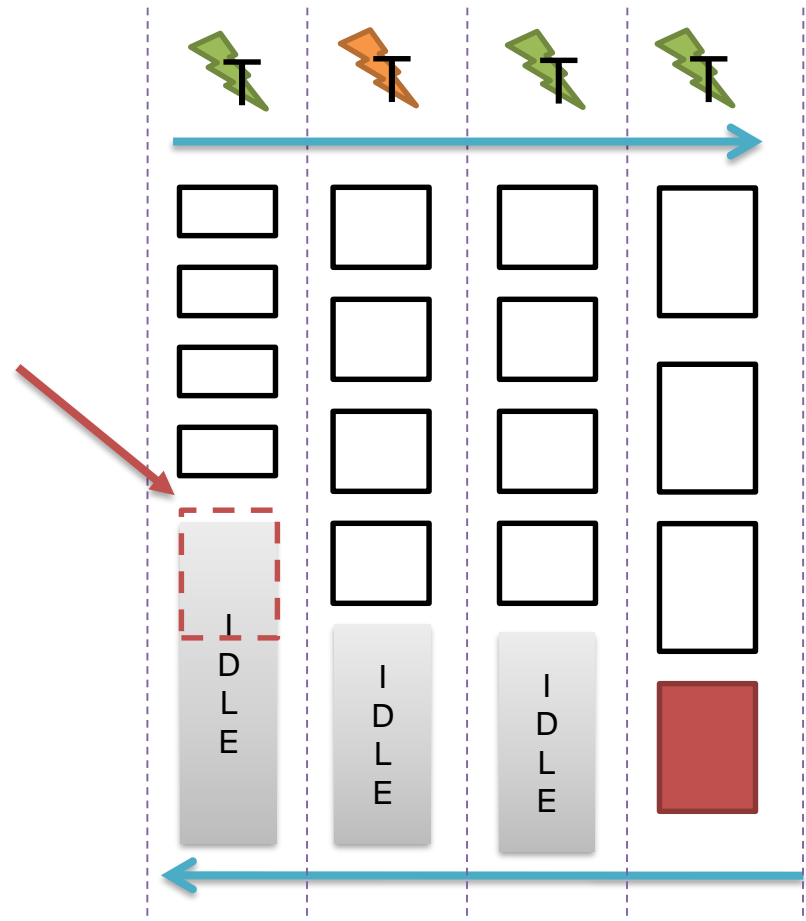
Unbalanced loop partitioning

- › Iterations are statically assigned before entering the loop
 - Might not be effective nor efficient

```
#pragma omp parallel for num_threads (4)
for (int i=0; i<16; i++)
{
```

```
/* UNBALANCED LOOP CODE */
```

```
} /* (implicit) Barrier */
```





Dynamic loops

- › Assign iterations to threads in a dynamic manner
 - At runtime!!

- › Static semantic
 - "Partition the loop in N_{threads} parts threads and assign them to the team"
 - Naive and passive

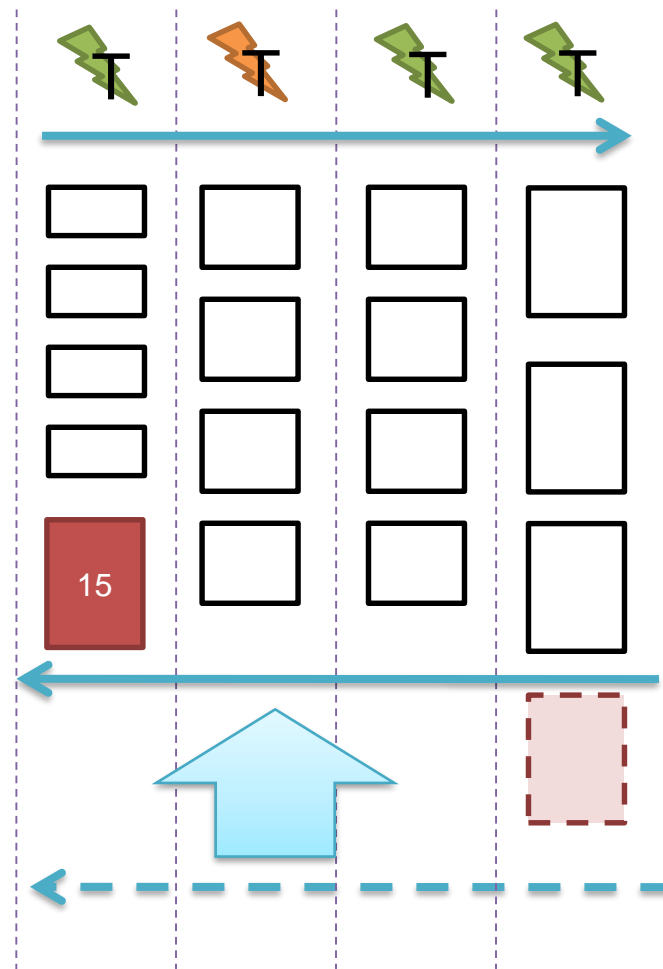
- › Dynamic semantic
 - "Each thread in the team fetches an iteration (or a block of) when he's idle"
 - Proactive
 - Work-conservative



Dynamic loops

› Activated using the `schedule` clause

```
#pragma omp parallel for num_threads (4) \  
    schedule(dynamic)  
  
for (int i=0; i<16; i++)  
{  
  
    /* UNBALANCED LOOP CODE */  
  
}  
/* (implicit) Barrier */
```





The schedule clause

```
#pragma omp for [clause [[,] clause]...] new-line  
for-loops
```

Where clauses can be:

```
private(list)  
firstprivate(list)  
lastprivate(list)  
linear(list[ : linear-step])  
reduction(reduction-identifier : list)  
schedule([modifier [, modifier]:]kind[, chunk_size])  
collapse(n)  
ordered[(n)]  
nowait
```

- › The iteration space is divided according to the schedule clause
 - kind can be: {static | dynamic | guided | auto | runtime }



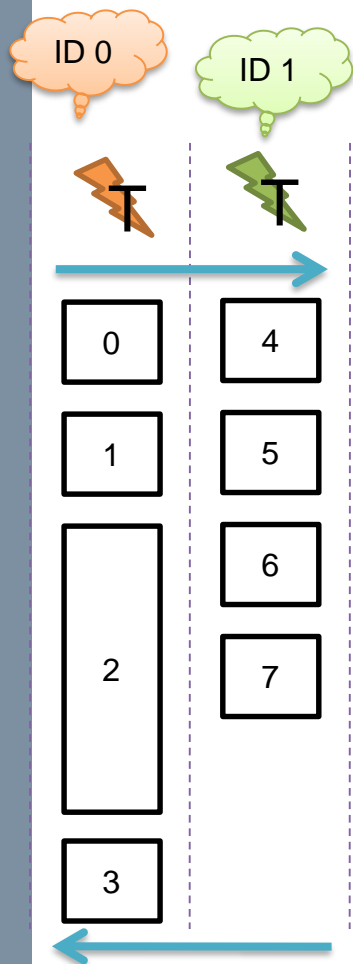
OMP loop schedule policies

- › `schedule(static[, chunk_size])`
 - Iterations are divided into **chunks** of `chunk_size`, and chunks are assigned to threads **before entering the loop**
 - If `chunk_size` unspecified, = `NITER/NTHREADS` (with some adjustment...)

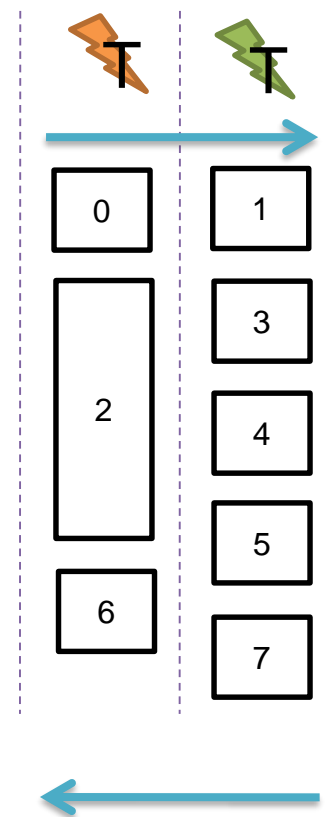
- › `schedule(dynamic[, chunk_size])`
 - Iterations are divided into chunks of `chunk_size`
 - **At runtime**, each thread requests for a new chunk after finishing one
 - If `chunk_size` unspecified, then = 1



Static vs. Dynamic



```
#pragma omp parallel for num_threads (2) \  
    schedule ( ... )  
  
for (int i=0; i<8; i++)  
{  
  
    // ...  
  
} /* (implicit) Barrier */
```





OMP loop schedule policies (cont'd)

- › `schedule(guided[, chunk_size])`
 - A mix of static and dynamic
 - `chunk_size` determined statically, assignment done dynamically

- › `schedule(auto)`
 - Programmer let compiler and/or runtime decide
 - Chunk size, thread mapping..
 - "I wash my hands"

- › `schedule(runtime)`
 - Only runtime decides according to `run-sched-var` ICV
 - If `run-sched-var = auto`, then implementation defined



Loops chunking

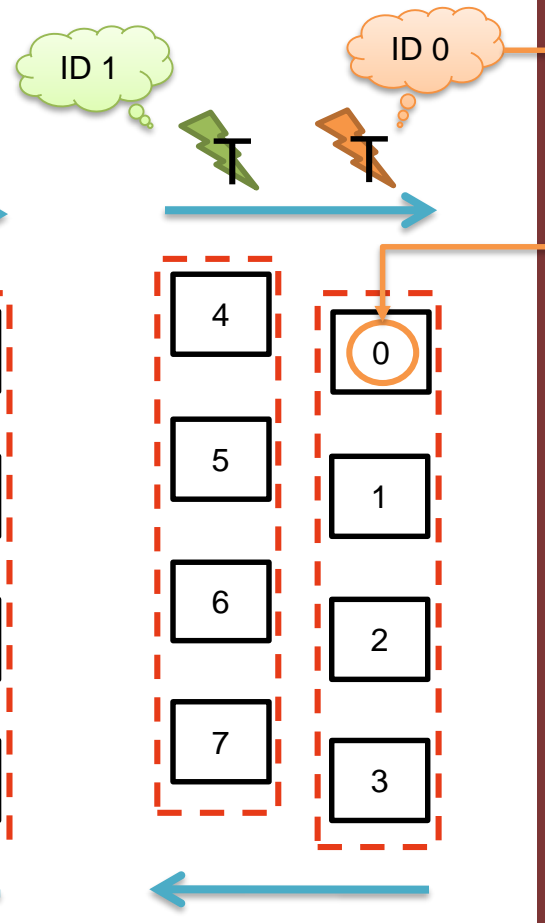
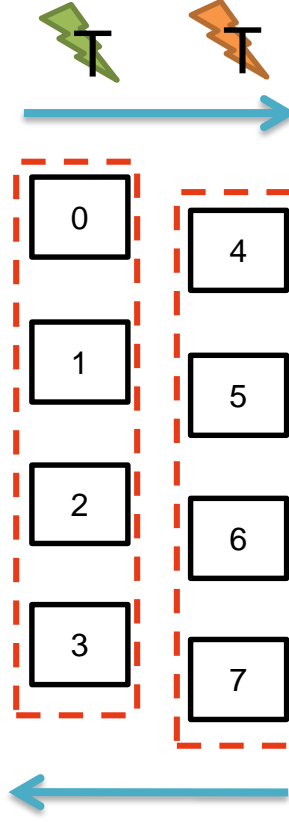
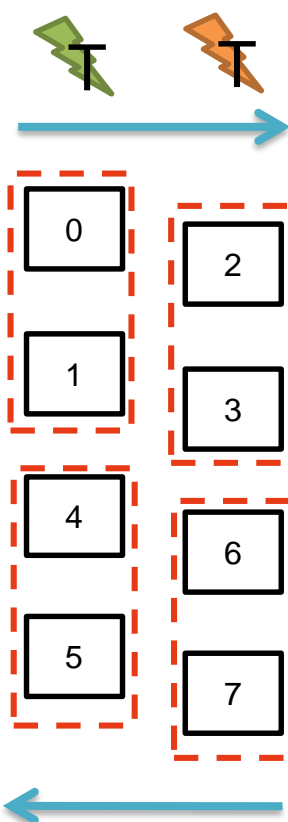
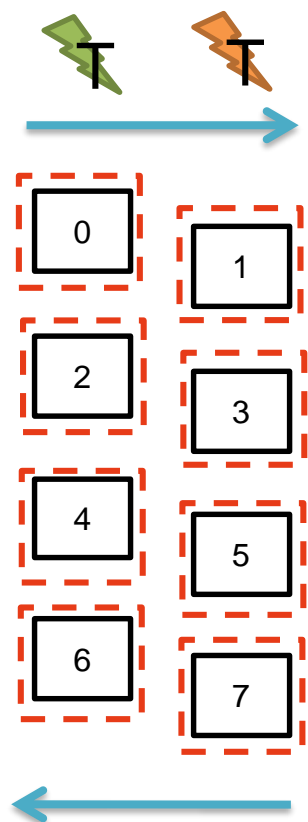
`schedule(dynamic, 1)`

`Schedule(dynamic)`

`schedule(dynamic, NITER/NTRHD)`

`schedule(static)`

`schedule(dynamic, 2)`





Modifiers, collapsed and ordered

```
#pragma omp for [clause [[,] clause]...] new-line  
for-loops
```

Where clauses can be:

```
private(list)  
firstprivate(list)  
lastprivate(list)  
linear(list[ : linear-step])  
reduction(reduction-identifier : list)  
schedule([modifier [, modifier]:]kind[, chunk_size])  
collapse(n)  
ordered[(n)]  
nowait
```

› These we won't see

- E.g., modifier can be : {monothonic | nonmonothonic | simd }
- Let you tune the loop and give more information to the OMP stack
- To maximize performance



Static vs. dynamic loops

- › So, why not always dynamic?
 - For unbalanced workloads, they are more flexible
 - "For balanced workload, in the worst case, they behave like static loops!"

Not always true!

- › Static loops have a (light) cost only before the loop
 - Actually, the lighter way you can distribute work in OpenMP!!
 - Often a performance reference..
- › Dynamic loops have a cost:
 - For initializing the loop
 - For fetching a(nother) chunk of work
 - At the end of the loop





OpenMP loops overhead

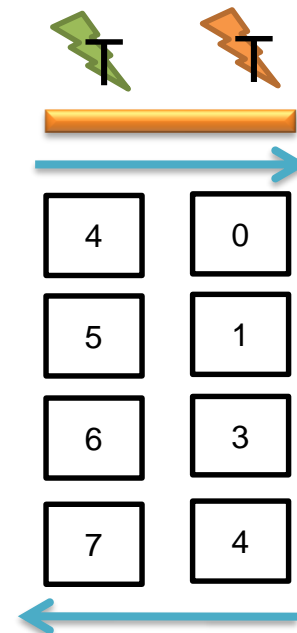
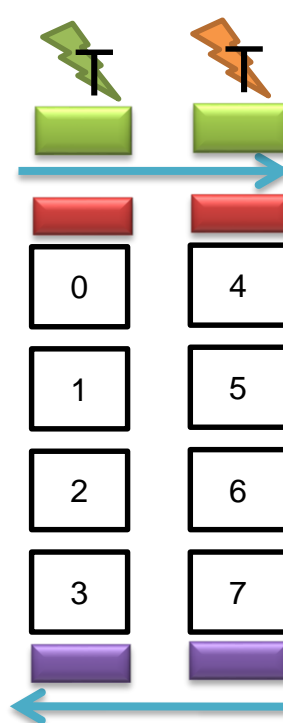
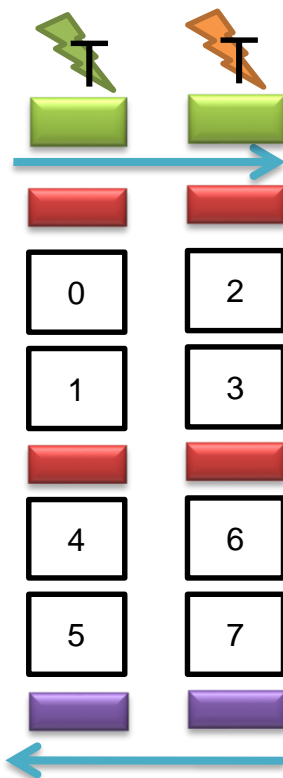
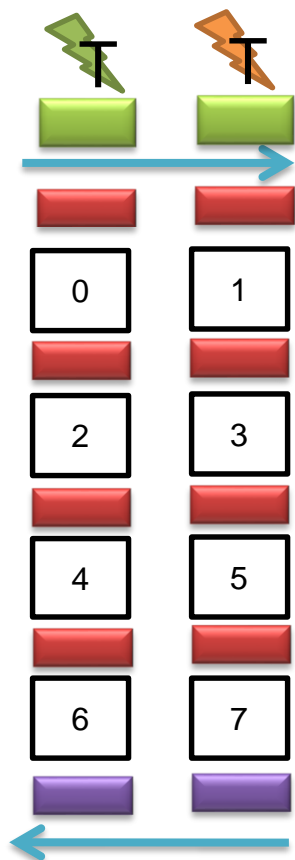
`schedule(dynamic, 1)`

`schedule(dynamic)`

`schedule(dynamic, NITER/NTHRD)`

`schedule(dynamic, 2)`

`schedule(static)`





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, simulate unbalanced workload
 - Use both static and dynamic loops
 - Each thread prints iteration index `i`
 - What do you (should) see?

```
#pragma omp parallel for schedule(...)
for (int i=0; i<NUM; i++)
{
    // ...

    // Simulate iteration-dependant work
    volatile long a = i * 1000000L;

    while(a--)
        ;
}
```



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>
 - <http://gcc.gnu.org>