

OpenMP threading

Paolo Burgio
paolo.burgio@unimore.it





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



Regular, loop-based parallelism

- > 2008
 - OpenMP 3.0
- > 2011
 - OpenMP 3.1



Irregular, parallelism → tasking

- > 2014
 - OpenMP 4.5



Heterogeneous parallelism, *à la* GP-GPU



What is OpenMP?



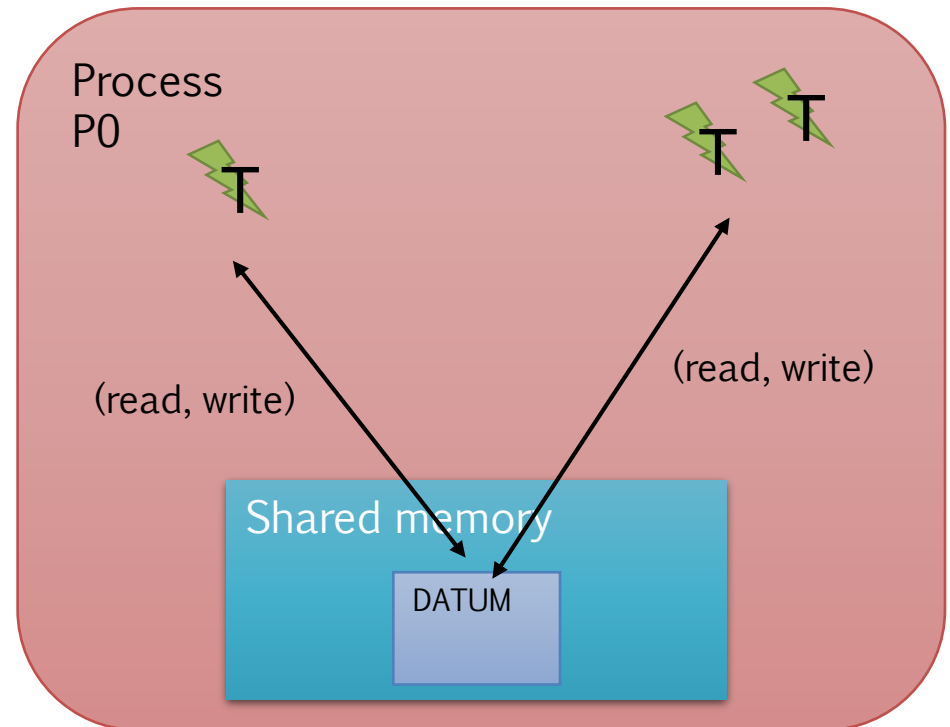
[eng.wikipedia.org](https://en.cppreference.com/w/cpp/parallel/openmp)

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, on most platforms, processor architectures and operating systems, including Solaris, AIX, HP-UX, Linux, OS X, and Windows. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior



Shared memory

- › Coherence problem
 - Memory consistency issue
 - Data races
- › Can share data ptrs
 - Ease-to-use
- › Several paradigm ("flavours")
 - Symmetric Multiprocessing
 - Distributed Shared memory
 - Partitioned Global Access Space





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



pragma omp parallel construct

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

Where clauses can be:

```
if([parallel :] scalar-expression)  
num_threads (integer-expression)  
default(shared | none)  
firstprivate (list)  
private (list)  
shared (list)  
copyin (list)  
reduction(reduction-identifier : list)  
proc_bind(master | close | spread)
```

Creating a parreg

› Master-slave, fork-join execution model

- Master thread spawns a team of Slave threads
- They all perform computation in parallel
- At the end of the parallel region, implicit barrier

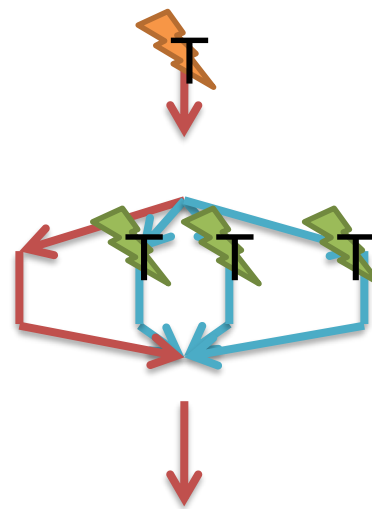
```
int main()
{
    /* Sequential code */

    #pragma omp parallel num_threads(4)
    {

        /* Parallel code */

    } // Parreg end: (implicit) barrier

    /* (More) sequential code */
}
```





Thread control

- › OpenMP provides ways to
 - Retrieve thread ID
 - Retrieve number of threads
 - Set the number of threads
 - Specify threads-to-cores affinity (we won't see this)



Get thread ID

omp.h

```
/*  
 * The omp_get_thread_num routine returns  
 * the thread number, within the current team,  
 * of the calling thread.  
 */  
int omp_get_thread_num(void);
```

> Function call

- Returns an integer
- Can be used **everywhere** where inside your code
 - > Also in sequential parts

> Don't forget to **#include <omp.h>!!**

> Master thread (typically) has ID #0 



Get the number of threads

omp.h

```
/*  
 * The omp_get_num_threads routine returns  
 * the number of threads in the current team.  
 */  
int omp_get_num_threads(void);
```

> Function call

- Returns an integer
- Can be used **everywhere** where inside your code
 - > Also in sequential parts
- Don't forget to **#include <omp.h>!!**

> BTW

- ...thread ID from `omp_get_thread_num` is always < this value..



Set the number of threads (2)

omp.h

```
/*  
 * The omp_set_num_threads routine affects the number of threads  
 * to be used for subsequent parallel regions that do not specify  
 * a num_threads clause, by setting the value of the first  
 * element of the nthreads-var ICV of the current task.  
 */  
void omp_set_num_threads(int num_threads);
```

- › Function call
 - Accepts an integer
 - Can be used **everywhere** where inside your code
 - › Also in sequential parts
- › Don't forget to **#include <omp.h>!!**
- › Overrides value from `OMP_NUM_THREADS`
 - Affects all of the subsequent parallel regions





Set the number of threads (1)

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

Where clauses can be:

```
if([parallel :] scalar-expression)  
num_threads (integer-expression)  
default(shared | none)  
firstprivate (list)  
private (list)  
shared (list)  
copyin (list)  
reduction(reduction-identifier : list)  
proc_bind(master | close | spread)
```



Exercise

Let's
code!

- › Spawn a team of parallel (OMP) Threads
 - Each printing "Hello Parallel World. I am thread #<tid> out of <num>"
 - Also, print "Hello Sequential World. I am thread #<tid> out of <num>" before and after parreg
 - What do you see?

- › Don't forget the `-fopenmp` switch
 - Compiler-dependant!

Compiler	Compiler Options
GNU (gcc, g++, gfortran)	<code>-fopenmp</code>
Intel (icc ifort)	<code>-openmp</code>
Portland Group (pgcc,pgCC,pgf77,pgf90)	<code>-mp</code>



The `if` clause

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

Where clauses can be:

```
if([parallel :] scalar-expression)  
num_threads (integer-expression)  
default(shared | none)  
firstprivate (list)  
private (list)  
shared (list)  
copyin (list)  
reduction(reduction-identifier : list)  
proc_bind(master | close | spread)
```

- › If `scalar-expression` is false, then spawn a single-thread region
- › We will see it also in other constructs...
 - "Can be used in combined constructs, in this case programmer must specify which one it refers to (in this case, with the `parallel` specifier)"



Nested parallel regions

- › One can create a parallel region **within** a parallel region
 - A **new** team of thread is created
- › Enabled-disabled via environmental var, or library call
- › Easy to lose control..
 - Too many threads!
 - Their number explodes
 - Be ready to debug..



Dynamic # threads adjustment

- › The OpenMP implementation might decide to dynamically adjust the number of thread within a parreg
 - Aka the team size
 - Under heavy load might be reduced

- › Also this can be disabled



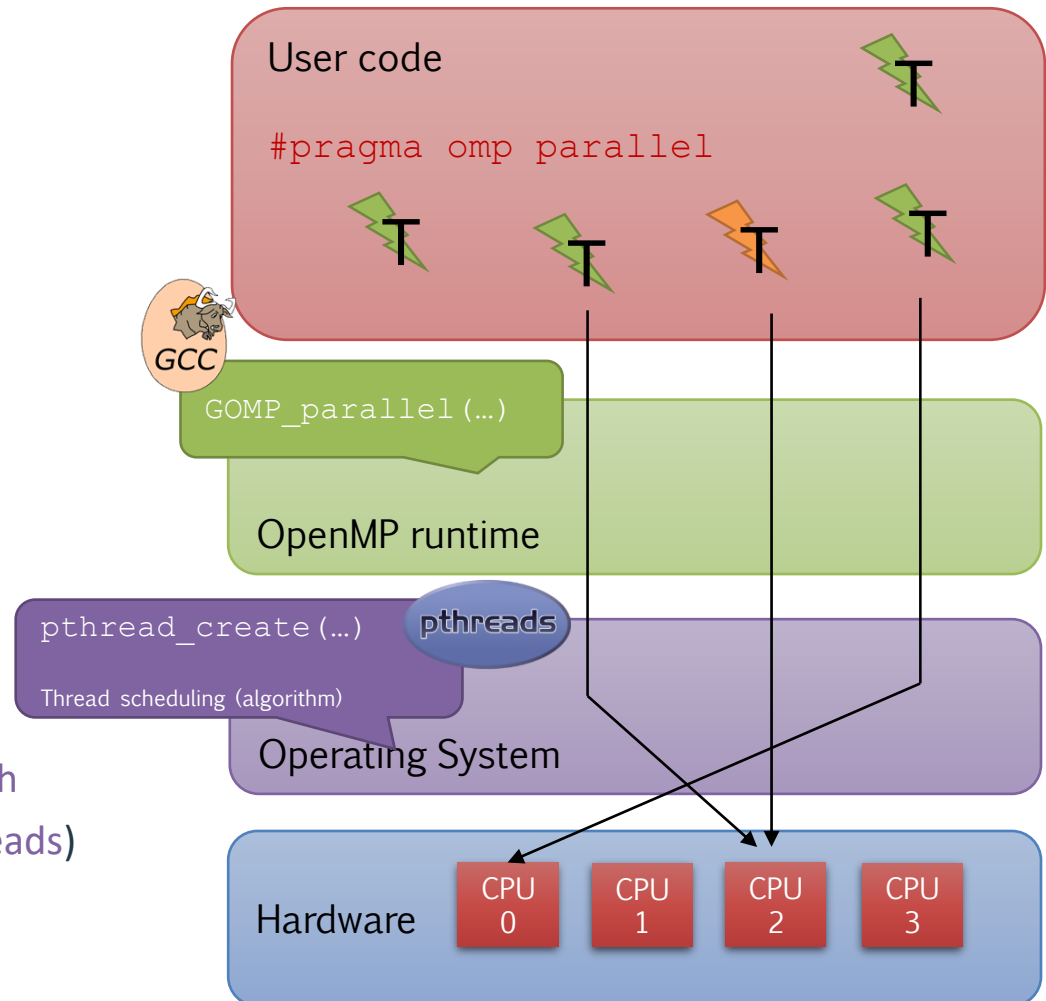
Under the hood

- › You have control on # threads
 - Partly
- › You have partial control on where the threads are scheduled
 - Affinity
- › You have no control on the actual scheduling!
 - Demanded to OS + runtime
- › ..."OS and runtime"?

OpenMP software stack

Multi-layer stack, engineered for portability

- › Application code
 - Compliant to OMP standard
- › Runtime (e.g., GCC-OpenMP)
 - Provides services for parallelism
 - Compiler replaces pragma with runtime-specific function calls
- › OS (e.g., Linux)
 - Provides basic services
 - Threading, memory mgmt, synch
 - Can be standardized (e.g., PThreads)





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



Exercise

Let's
code!

- › Declare and initialize a variable outside the parallel region

- › Spawn a team of parallel Threads
 - Each printing the value of the variable

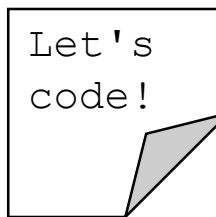
- › What do you see?



shared variables

- › The variable is `shared` among the parallel threads
 - If one thread modifies it, then all threads see the new value

- › Let's see this!
 - Let (only) Thread 0 modify the value of the variable



- › What's happening?
 - (probably | might be that) Thread 0 modifies the value **after** the other threads read it
 - The more thread you have, the more probably you see this...



As opposite to... `private` variables

- › Threads might want to own a `private` copy of a datum
 - Other threads cannot modify it

- › Two ways
 - They can declare it inside the parallel region
 - Or, they can use *data sharing attribute clauses*

- › `private` | `firstprivate`
 - Create a storage for the specified datum (variable or param) in each thread's stack



Data sharing clauses in parregs

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

Where clauses can be:

```
if([parallel :] scalar-expression)  
num_threads (integer-expression)  
default(shared | none)  
firstprivate (list)  
private (list)  
shared (list)  
copyin (list)  
reduction(reduction-identifier : list)  
proc_bind(master | close | spread)
```



Initial value for (`first`) `private` data

- › How is the private data initialized?
 - `firstprivate` initializes it with the value of the enclosing context
 - `private` does not initialize it / initializes it with 0



Exercise

Let's
code!

- › Declare and initialize a variable outside the parallel region
- › Spawn a team of parallel Threads
 - Mark the variable as `private` using data sharing clause
 - Each printing the value of the variable
 - Let (only) Thread 0 modify the value of the variable
- › What do you see?
 - Now, mark the variable as `firstprivate`



shared data-sharing clause

- › All variables specified are shared among all threads
- › Programmer is in charge of consistency!
 - OpenMP philosophy..



Multiple variables in a single clause

- › Do not need to repeat the clause always
 - If you don't want..
- › Separated by commas

```
int a = 11, b = 1, c;  
  
#pragma omp parallel num_threads(16) \  
    private(a, b)  
private(c)  
{  
    ...  
}
```



private vs. parreg-local variables

- › Find the difference between...

```
#pragma omp parallel num_threads(4)
{
    int a = ...
}
```

```
int a = ...;
#pragma omp parallel private(a) \
    num_threads(4)
{
    a = ...
}
```

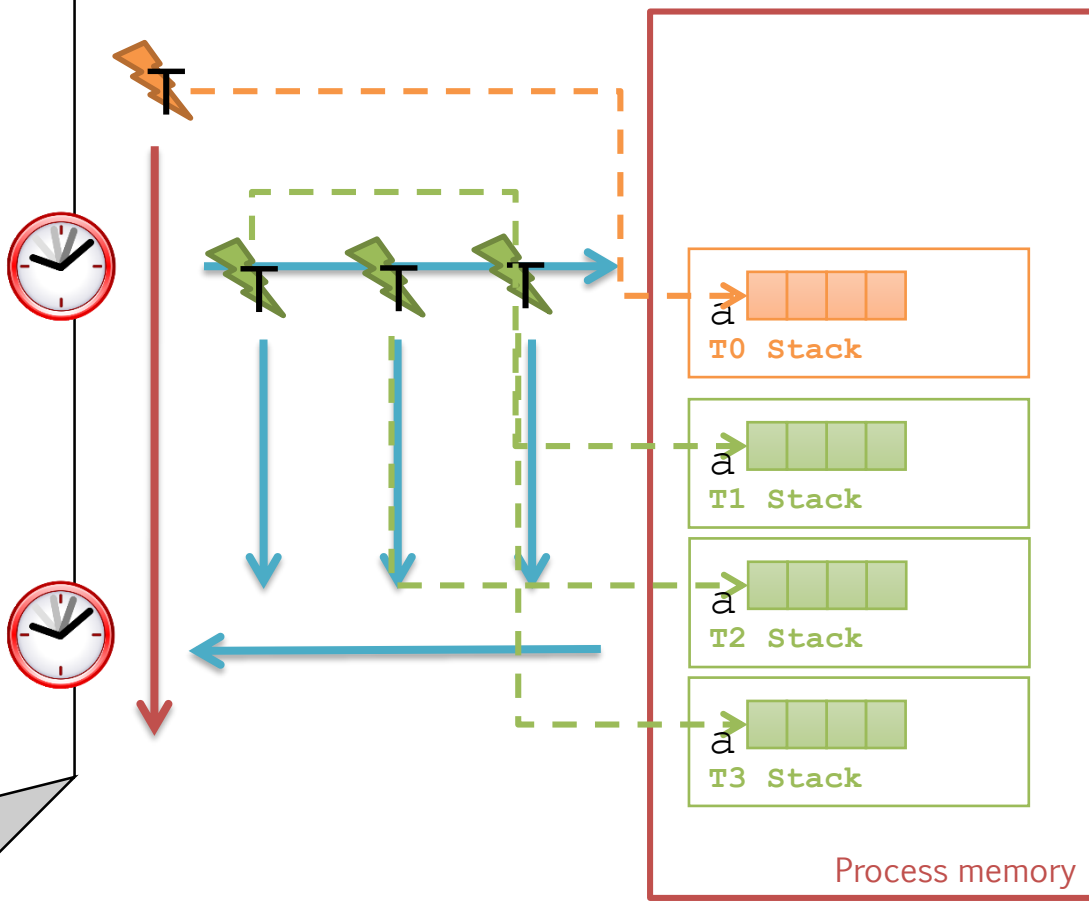
- › *"A new storage is created as we enter the region, and destroyed after"*
- › On the right (`private`)
 - There is also a storage that exists before and after parreg



Variables and memory (1)

> "The traditional way"

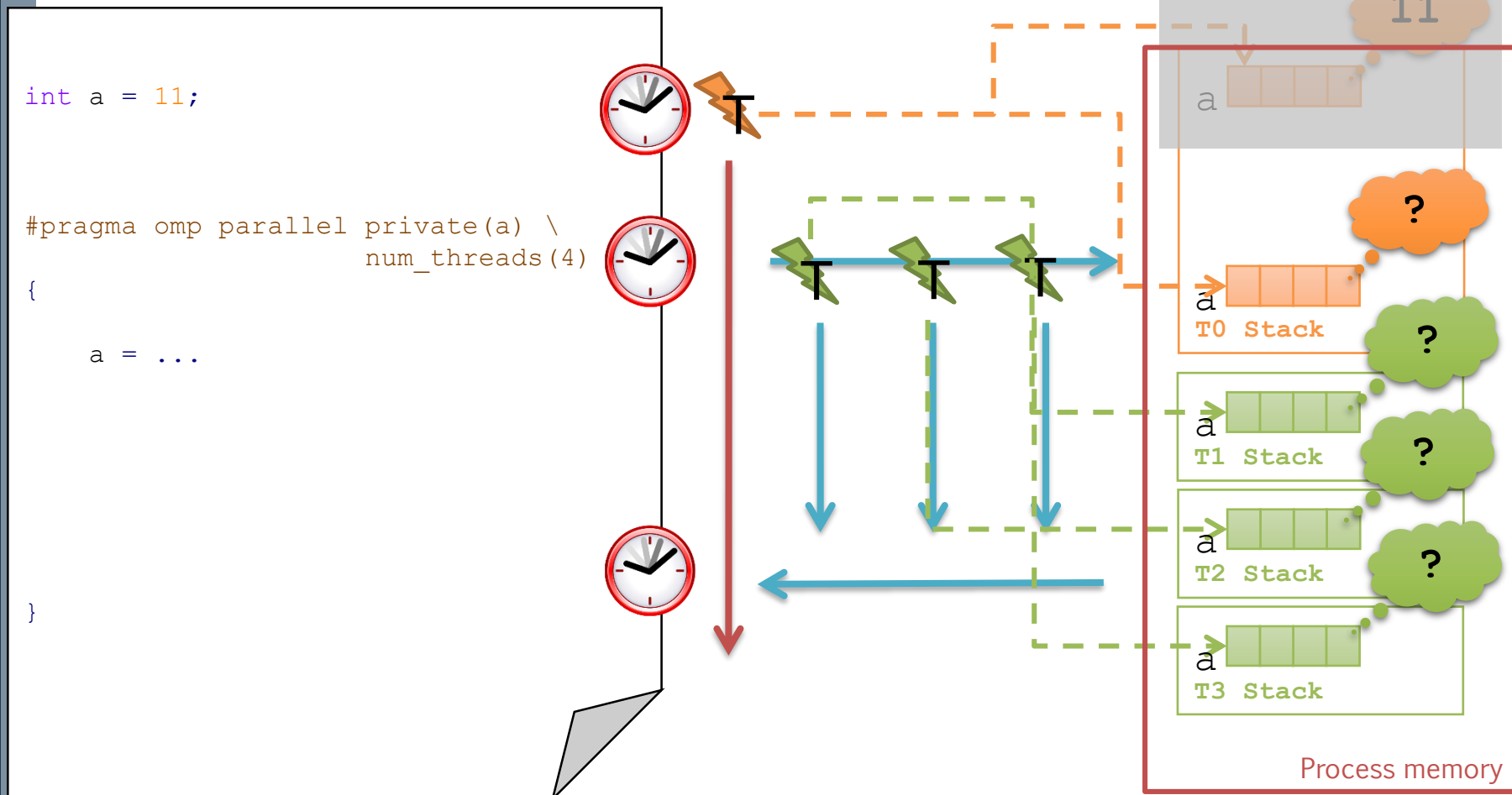
```
#pragma omp parallel num_threads(4)
{
    int a = ...
}
```





Variables and memory (2)

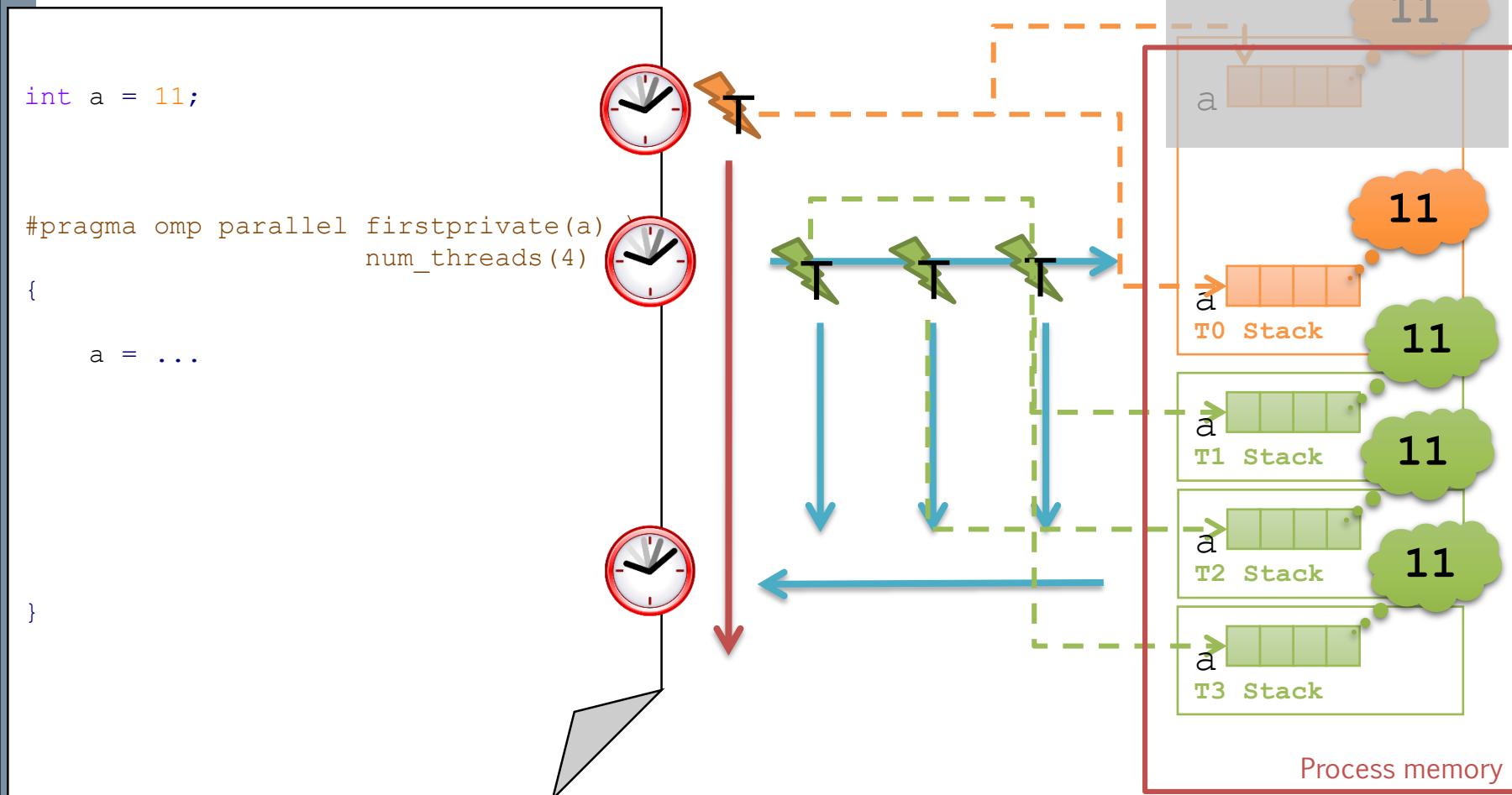
- › Create a new storage for the variables, local to threads





Variables and memory (3)

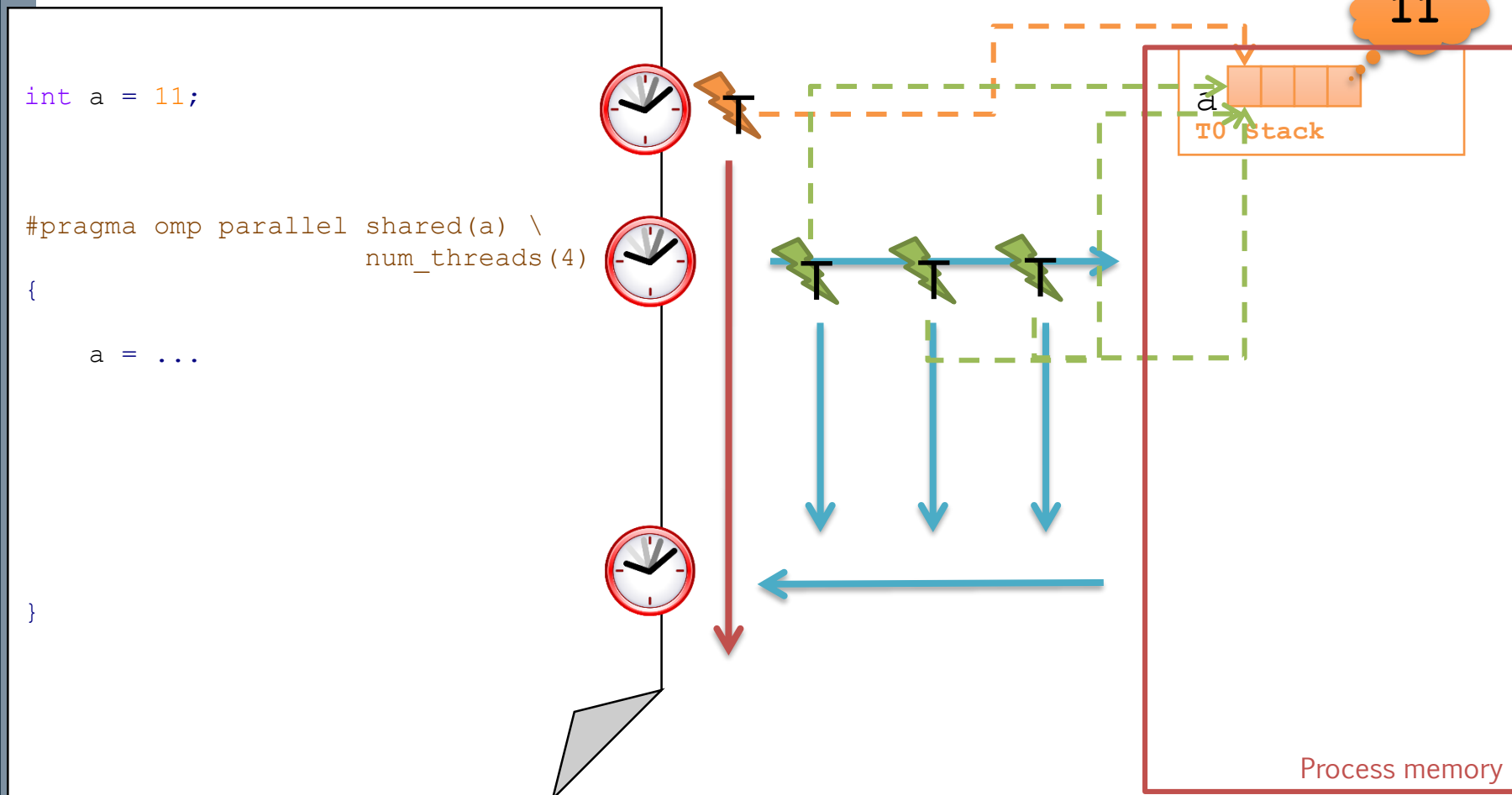
- > Create a new storage for the variables, local to threads, and initialize





Variables and memory (4)

- › Every slave Thread refers to master's storage





Data sharing clauses in parregs

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

Where clauses can be:

```
if([parallel :] scalar-expression)  
num_threads (integer-expression)  
default(shared | none)  
firstprivate (list)  
private (list)  
shared (list)  
copyin (list)  
reduction(reduction-identifier : list)  
proc_bind(master | close | spread)
```



default data sharing clause

OpenMP specifications

The `default` clause explicitly determines the data-sharing attributes of variables that are referenced in a `parallel`, `teams`, or task generating construct and would otherwise be implicitly determined (see Section 2.15.1.1 on page 179).

> Can be

- `shared`: all variables referenced in the construct that are not present in a data sharing clause are shared
- `none`: each variable that is referenced in the construct, *and that does not have a predetermined data-sharing attribute*, **must** have its data-sharing attribute explicitly determined using a data-sharing clause

> (Yes, we can have predetermined attributes)

- We won't see this



Exercise

Let's
code!

- › Declare and initialize a variable outside the parallel region
- › Spawn a team of parallel Threads
 - Use the `default (none)` using data sharing clause
 - Do not use any other data sharing clause
 - Each thread prints the value of the variable
- › What do you see?

Watch out!



- › We haven't seen everything..
 - Rules determining default sharing attributes are complex
 - For instance, `automatic` variables within a `parreg` are implicitly private
 - `static` variables within a `parallel` are **implicitly shared!!**

- › Stay on the safe side:
 - Use the `default` clause for variables you care about!!
 - Use `shared` clauses
 - If you can, declare variables inside `parreg`, instead of marking them as `private`

- › ...informatics is the art science of managing data



reduction clause in parregs

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

Where clauses can be:

```
if([parallel :] scalar-expression)  
num_threads (integer-expression)  
default(shared | none)  
firstprivate (list)  
private (list)  
shared (list)  
copyin (list)  
reduction(reduction-identifier : list)  
proc_bind(master | close | spread)
```



Reduction

OpenMP specifications

*The reduction clause can be used to perform some forms of **recurrence** calculations (involving **mathematically associative and commutative operators**) in parallel. For parallel [...], a **private** copy of each list item is created, one for each implicit task, as if the private clause had been used. [...] The private copy is then initialized as specified above. At the end of the region for which the reduction clause was specified, the original list item is updated by **combining its original value with the final value of each of the private copies**, using the combiner of the specified reduction-identifier.*

› In a nutshell

- For each variable specified, create a `private` storage
- At the end of the region, update master thread's value according to `reduction-identifier`
- The variable must be **qualified** for that operation



Reduction identifiers

+ - * & | ^ && || max min

› Mathematical/logical identifiers

- Each has a default initializer, and a combiner
- Minus (-) is more or less the same as plus (+)

OpenMP
specifications

Identifier	Initializer	Combiner
+	<code>omp_priv = 0</code>	<code>omp_out += omp_in</code>
*	<code>omp_priv = 1</code>	<code>omp_out *= omp_in</code>
-	<code>omp_priv = 0</code>	<code>omp_out += omp_in</code>
&	<code>omp_priv = 0</code>	<code>omp_out &= omp_in</code>
	<code>omp_priv = 0</code>	<code>omp_out = omp_in</code>
^	<code>omp_priv = 0</code>	<code>omp_out ^= omp_in</code>
&&	<code>omp_priv = 1</code>	<code>omp_out = omp_in && omp_out</code>
	<code>omp_priv = 0</code>	<code>omp_out = omp_in omp_out</code>
max	<code>omp_priv = <i>Least representable number in the reduction list item type</i></code>	<code>omp_out = omp_in > omp_out ? omp_in : omp_out</code>
min	<code>omp_priv = <i>Largest representable number in the reduction list item type</i></code>	<code>omp_out = omp_in < omp_out ? omp_in : omp_out</code>



Exercise

Let's
code!

› Declare and initialize a variable outside the parallel region

- `int a = 11`

› Spawn a team of parallel Threads

- Mark the variable as `reduction(+:a)`

- Increment variable `a`

- Print the value of the variable **before**, **inside**, and **after** the parreg

› What do you see?

- (at home) repeat with other reduction-identifiers



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/index.html>

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

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

- › A "small blog"
 - <http://www.google.com>