PARALLEL PROGRAMMING WITH OPENMP

Caspar van Leewen
Carlos Teijeiro Barjas

Goals

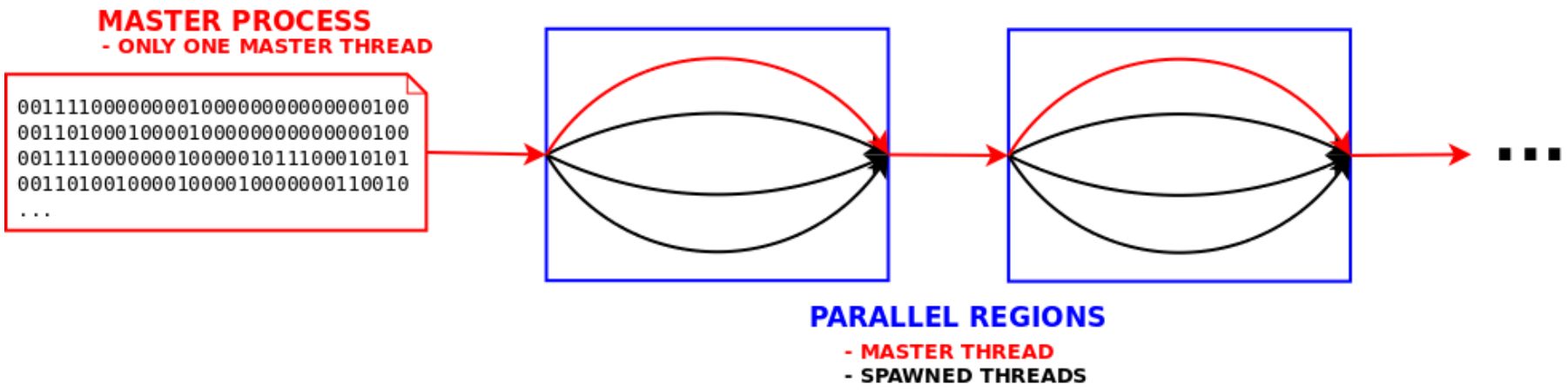
- The goal is to understand:
  - How multithreading can be used for parallel computation
  - The structure of the OpenMP framework (library, directives)
  - How OpenMP can be used to distribute work

Content

- Components of OpenMP
- Shared memory parallelization with OpenMP constructs (with hands-on)
- Special hands-on with automatic parallelization: Parallelware Trainer

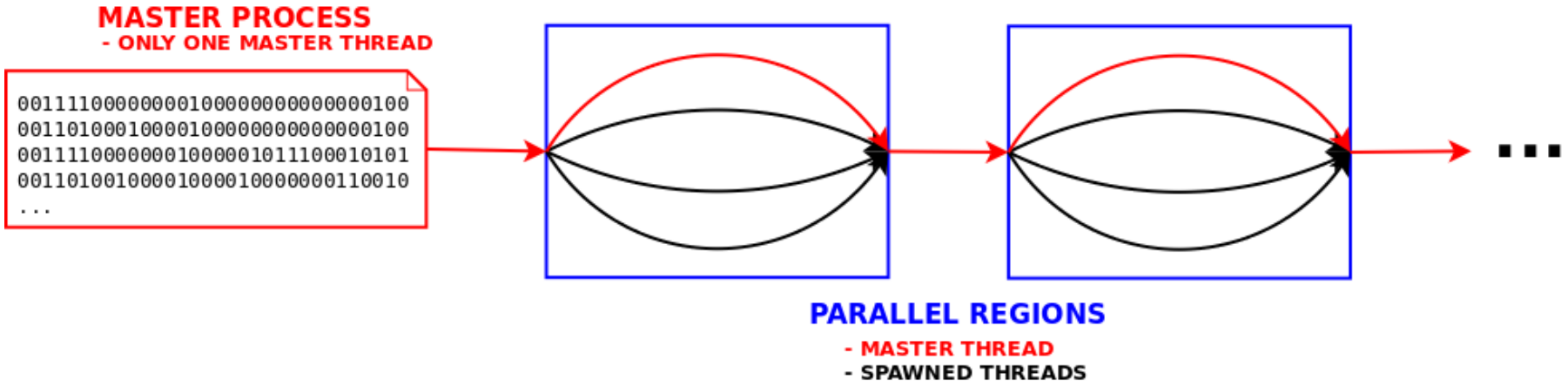
OpenMP (Open Multi-Processing)

- Standard programming model for shared memory
- Very simple approach to parallel programming with many threads
- Minimal changes introduced in the original code
- Bindings for C and Fortran
- Latest version: 5.0 (November 2018)
  - This course will focus on basic constructs (up to v3.1), with reference to some new features



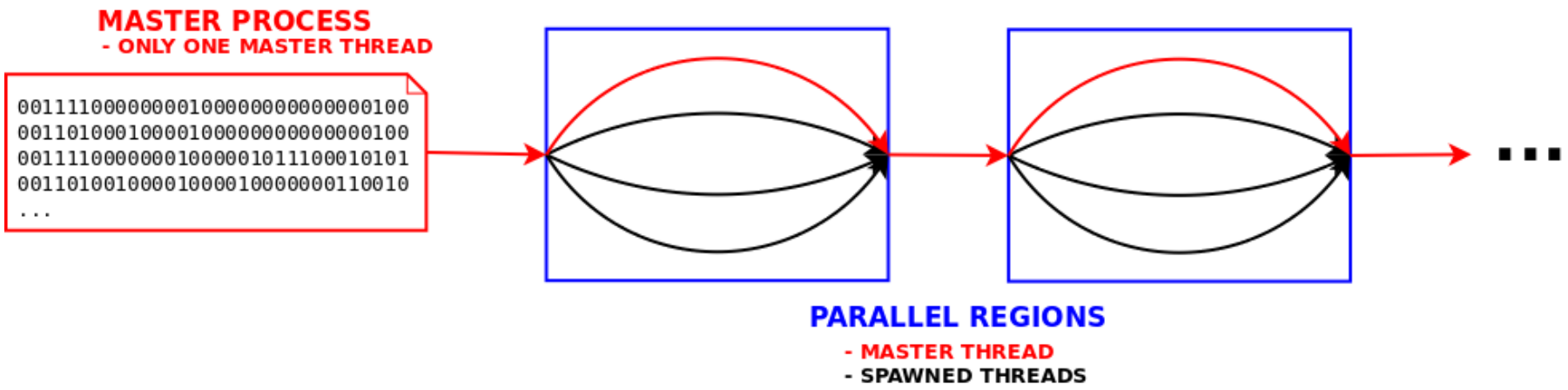
OpenMP (Open Multi-Processing)

- It defines standard constructs: implementations may vary depending on compiler vendors
- Threads *may* be scheduled on different CPU cores, exploiting multi-core hardware in a natural way
- No guarantee of optimal performance: tuning may be necessary
- No specific control on “side-effects” derived from the parallelization of code executions (e.g. variable dependencies, parallel I/O)



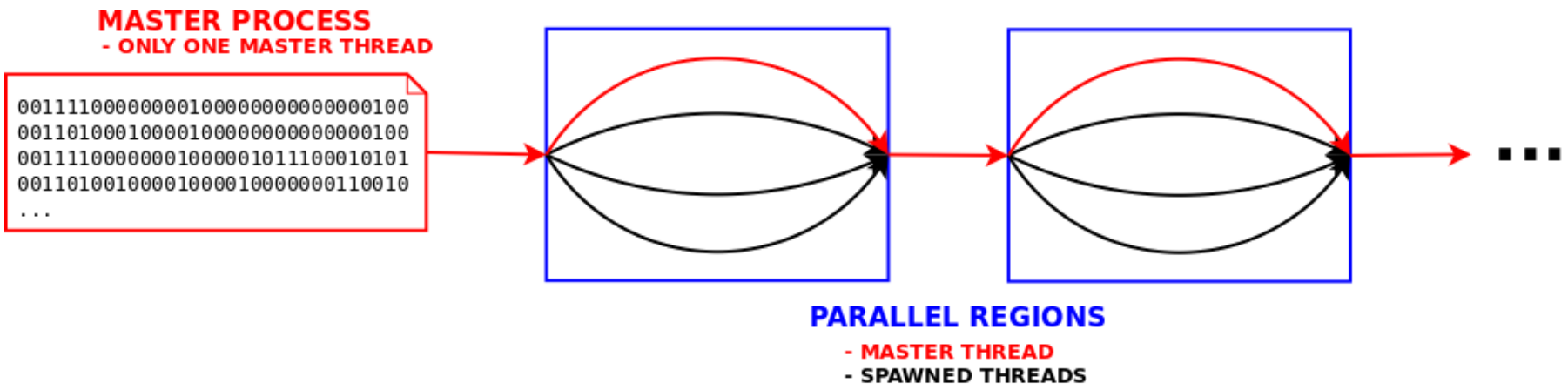
OpenMP (Open Multi-Processing)

- Full reference: standard document
- Condensed information
  - OpenMP Syntax Reference Guide
  - OpenMP Syntax Quick Reference Card for C/C++ or Fortran



OpenMP (Open Multi-Processing)

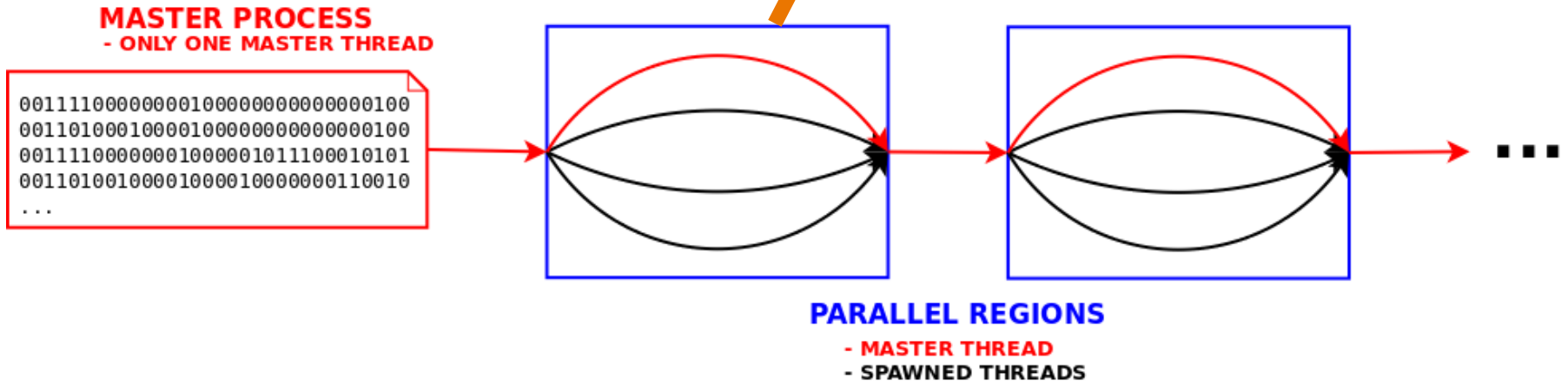
- Our task as programmers
  - Create multiple threads
  - Distribute work over threads
  - Make sure threads can execute independently (!)



OpenMP (Open Multi-Processing)

- OpenMP programs are multi-thread in a single process:

```
top - 11:10:36 up 41 days, 15:20,  1 user,  load average: 0,50, 0,35, 0,22
Tasks: 472 total,   2 running, 469 sleeping,   1 stopped,   0 zombie
%Cpu(s):  0,0 us,  0,1 sy, 16,7 ni, 83,2 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
KiB Mem : 49284800 total, 35080392 free,   774420 used, 13429988 buff/cache
KiB Swap: 25165820 total, 25136612 free,    29208 used. 46501188 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
31648 casparl   39  19  228816   1656   1180 R 400,0  0,0   0:22.37 my_omp_prog
   84 root      rt   0       0      0      0 S  0,3  0,0   0:08.41 migration/15
  477 root      20   0       0      0      0 S  0,3  0,0  59:26.57 xfsaild/sda3
```



Execution model: fork/join

- Programs are executed in the typical way (e.g. ./a.out)
- Threads are created (fork), and then many threads execute the same code
- Threads are destroyed (join), and then a serial execution continues
- Basic elements
  - Directives: lines inserted in the code with information for parallelization
    - Comment format: only interpreted when compiled with the correct flags
    - Some directives may contain different specifiers (clauses)
  - Library routines: small set of functions associated to OpenMP
  - Environment variables: specific parameters for parallel execution (e.g. number of threads)

Directives and clauses

- Structure of a directive:
  - <sentinel> <directive_name> [<clause> [, <clause>]*]
- Sentinel: identifier for a directive
  - C/C++ : #pragma
  - Fortran: !$OMP
- Clauses are arguments that may be added to some directives to modify their behavior
- OpenMP is able to provide significant flexibility for parallelism using just a few directives with some additional clauses

SURF SARA

Library routines

- Different subroutines/functions that provide specific information about the parallelization
- The most common ones are related to thread identification and parallel control
- The library header (C/C++) or the module (Fortran) may need to be included in the code for compilation
  - C/C++: #include <omp.h>
  - Fortran: use omp_lib
- Conditional compilation is also possible
  - C/C++: #ifdef _OPENMP
  - Fortran: !$

SURF SARA

Library routines

- Some common and useful library routines:
  - omp_get_num_threads / omp_set_num_threads
  - omp_get_thread_num
  - omp_get_num_procs
  - omp_in_parallel
  - omp_get_wtime

Environment variables

- Predefined variables that control the execution of OpenMP codes
- Definition depends on the shell that is used:
  - [csh,tcsh]: setenv VARIABLE value
  - [sh,ksh,bash]: export VARIABLE=value
- Most common variables
  - OMP_NUM_THREADS
  - OMP_SCHEDULE
  - OMP_NESTED

Let’s begin!

- Keep the quick reference cards for C/C++ or Fortran open
- Exercises and all additional information about the hands-on sessions can be found in the course GitHub repository:

  https://github.com/sara-nl/PRACE-MPI-OpenMP

- Additional MPI/OpenMP codes and more can be found in the PRACE CodeVault:

  https://repository.prace-ri.eu/git/CodeVault/training-material

SURF SARA

Directive PARALLEL

- Basis of any parallel implementation in OpenMP
  - C/C++ :
    - #pragma omp parallel [clause [, clause] ...]
  - Fortran:
    - !$OMP PARALLEL [clause [, clause] ...]

Directive PARALLEL

- C/C++ :

```
#pragma omp parallel
{
   printf("I am a thread");
}
```

- Fortran:

```
!$OMP PARALLEL
print *,"I am a thread"
!$OMP END PARALLEL
```

Directive PARALLEL

- Parallel regions can be defined to include any part of your code
- They may include many variables (also declarations), loops, function/subroutine calls, etc.
- Clauses help to provide specific features. These are the most common:
  - PRIVATE (list_of_variables)
  - SHARED (list_of_variables)
  - FIRSTPRIVATE (list_of_variables)
  - DEFAULT (PRIVATE | SHARED | FIRSTPRIVATE | NONE)
  - NUM_THREADS (scalar_integer_expression)

Directive PARALLEL

- PRIVATE clause
  - The indicated variables are private to each thread inside the parallel region
  - Every thread gets a private copy of the variables that is only accessed by the local thread
- SHARED clause
  - The indicated variables are accessible to all threads
  - Threads may read / write the same memory address any time
- FIRSTPRIVATE clause
  - Private variables are created, and their previous values before the parallel region is copied (variables are initialized)

Directive PARALLEL

- The number of threads of a parallel region can be defined in many ways
- Order of precedence in the definition:
  - Setting a value in the NUM_THREADS clause
  - Using the library routine omp_set_num_threads()
  - Setting the OMP_NUM_THREADS environment variable
  - Implementation default (e.g. number of CPUs available on a node)

Ready for a first hands-on?

- Try to parallelize the “Hello World” codes
  - C/C++: hello.c
  - Fortran: hello.f
- Check the use of omp_get_thread_num
  - C/C++: omp_threadnum.c
  - Fortran: omp_threadnum.f
- Understand how private/shared/firstprivate variables work
  - C/C++: omp_private.c, omp_firstprivate.c
  - Fortran: omp_private.f, omp_firstprivate.f
- What could you see after executing the parallel codes?

Quick example for Hello World

```
#include <omp.h>
#include <stdio.h>

int main (int argc, char *argv[])
{
  printf("Hello World from thread = %d\n", omp_get_thread_num());
}
```

Defines that this is the main program

Quick example for Hello World

```
#include <omp.h>
#include <stdio.h>

int main (int argc, char *argv[])
{
  printf("Hello World from thread = %d\n", omp_get_thread_num());
}
```

Prints formatted string with the number returned by omp_get_thread_num()

Quick example for Hello World

```
#include <omp.h>
#include <stdio.h>

int main (int argc, char *argv[])
{
/* Fork a team of threads */
#pragma omp parallel
  {
  printf("Hello World from thread = %d\n", omp_get_thread_num());
  }  /* All threads join master thread and disband */
}
```

Basic ideas / pitfalls

- The same code is executed by all the threads
- No specific order is enforced
- No control of access to the variables: this is left to the programmer (!!!)
- Main source of problems: data dependencies
  - Reading an old value of a variable that has not been written yet
  - Overwriting a correct value on a shared variable

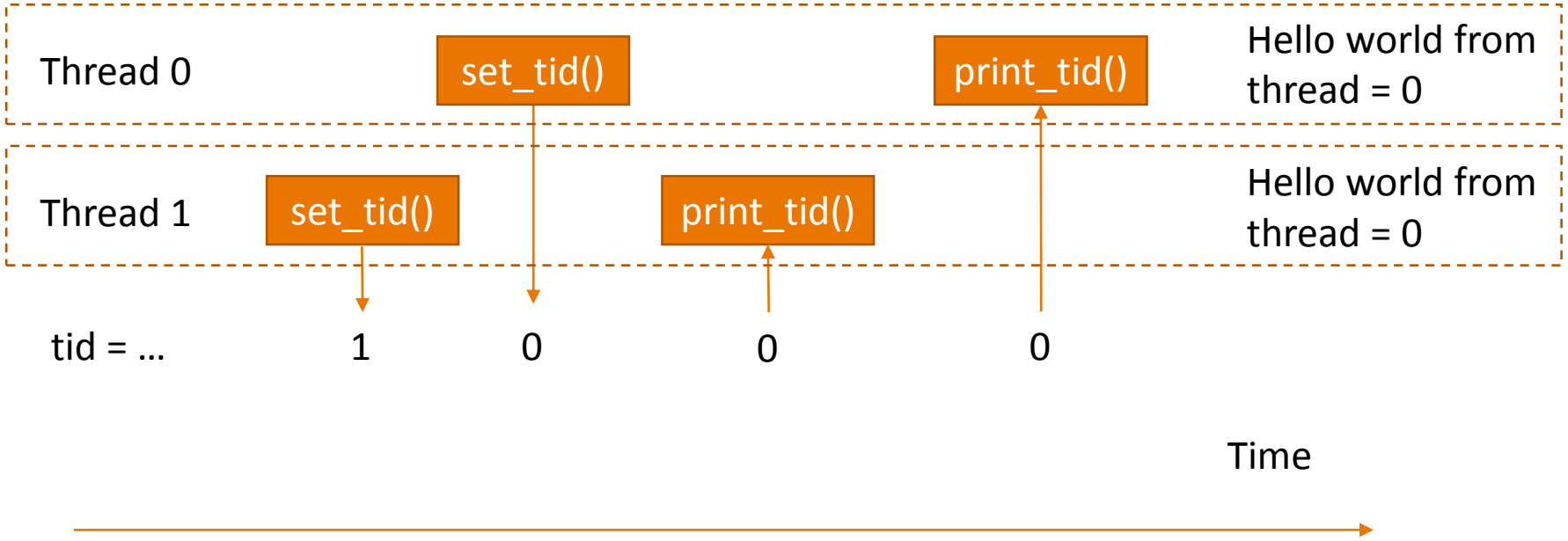
Basic ideas / pitfalls

- This code uses an additional variable “tid” inside the parallel region
- What can be the effect?

```
#include <omp.h>
#include <stdio.h>

int main (int argc, char *argv[])
{

int tid, nthreads;

/* Fork a team of threads */
#pragma omp parallel
  {
  tid = omp_get_thread_num();
  nthreads = omp_get_num_threads();
  printf("Hello World from thread = %d out of %d\n", tid, nthreads);
  }  /* All threads join master thread and disband */
}
```

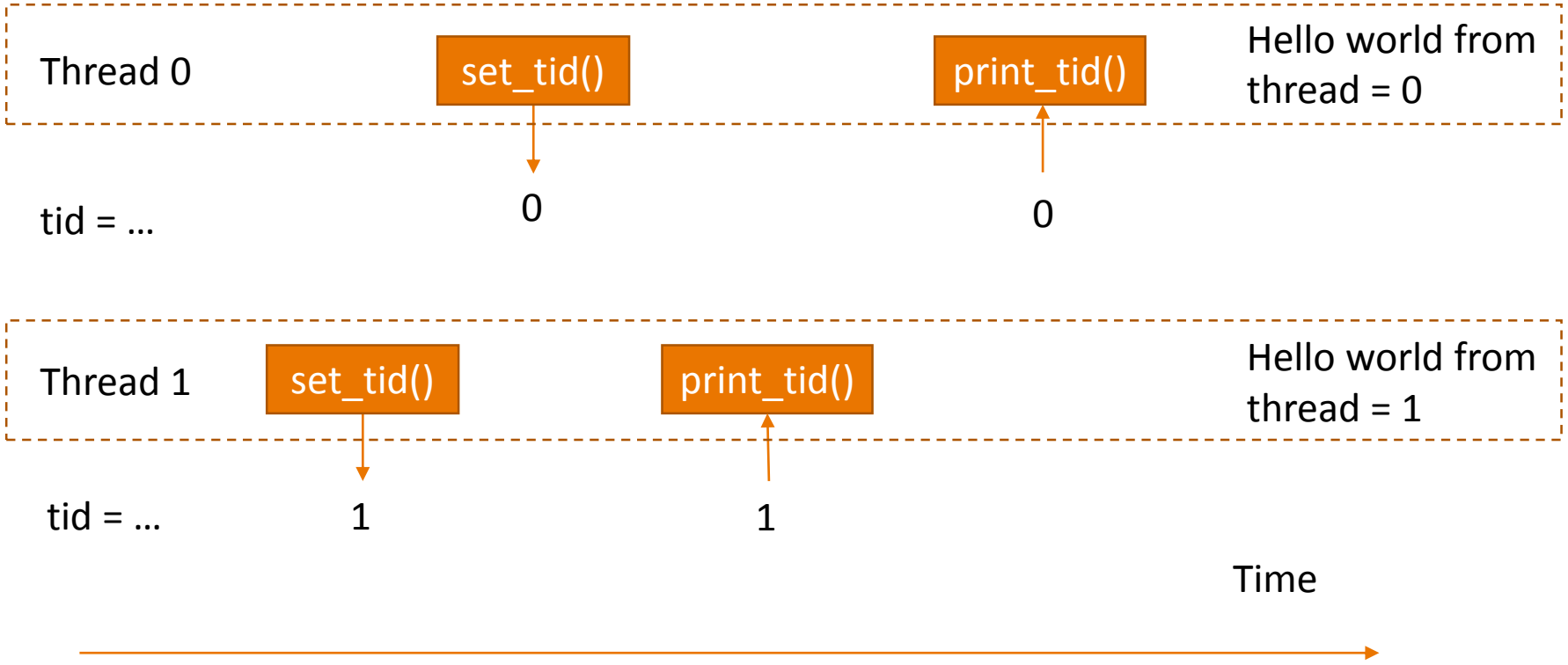
Basic ideas / pitfalls

- Race condition: shared variable “tid” is overwritten



Basic ideas / pitfalls

- Solution: define “tid” as a private variable



Work sharing constructs

- Not all threads should be doing the same work all the time…
- OpenMP has specific constructs to organize the parallel execution
- Work sharing is coupled to the use of PARALLEL
- The threads may share the work in different ways using different directives:
  - Loop parallelism: directive FOR (C/C++) / DO (Fortran)
  - Task parallelism: directive SECTIONS, SINGLE (+ construct TASK)

Loop parallelism (FOR/DO)

- The iterations of a (parallelizable!!!) loop can be distributed between threads
- This distribution can be merged with the PARALLEL directive
- C/C++ example

```
#pragma omp parallel
{
#pragma omp for
for(i=0; i<100; i++) {
  ...
}
}
```

Loop parallelism (FOR/DO)

- The iterations of a (parallelizable!!!) loop can be distributed between threads
- This distribution can be merged with the PARALLEL directive
- C/C++ example

```
#pragma omp parallel for
for(i=0; i<100; i++) {
  …
}
```

Loop parallelism (FOR/DO)

- The iterations of a (parallelizable!!!) loop can be distributed between threads
- This distribution can be merged with the PARALLEL directive
- Fortran example

```
!$OMP PARALLEL
!$OMP DO
do i=1,100
  …
enddo
[ !$OMP END DO ]
!$OMP END PARALLEL
```

Loop parallelism (FOR/DO)

- The iterations of a (parallelizable!!!) loop can be distributed between threads
- This distribution can be merged with the PARALLEL directive
- Fortran example

```
!$OMP PARALLEL DO
do i=1,100
  …
enddo
[ !$OMP END PARALLEL DO ]
```

Loop parallelism (FOR/DO)

- SCHEDULE clause: different options to control the work distribution
- Syntax: SCHEDULE (type [, blocksize])
- Type may be…
  - STATIC: iterations are divided in equal blocks of size *chunk* and statically assigned to a given thread
  - DYNAMIC: iterations are divided in equal blocks of size *chunk* and dynamically assigned whenever a thread is available
  - GUIDED: blocks are dynamically assigned, and their size is reduced after each assignment to a thread. This reduction is implementation-dependent.
  - RUNTIME: scheduling is deferred to environment variable OMP_SCHEDULE
  - AUTO: scheduling decision is left to the compiler or runtime environment

Loop parallelism (FOR/DO)

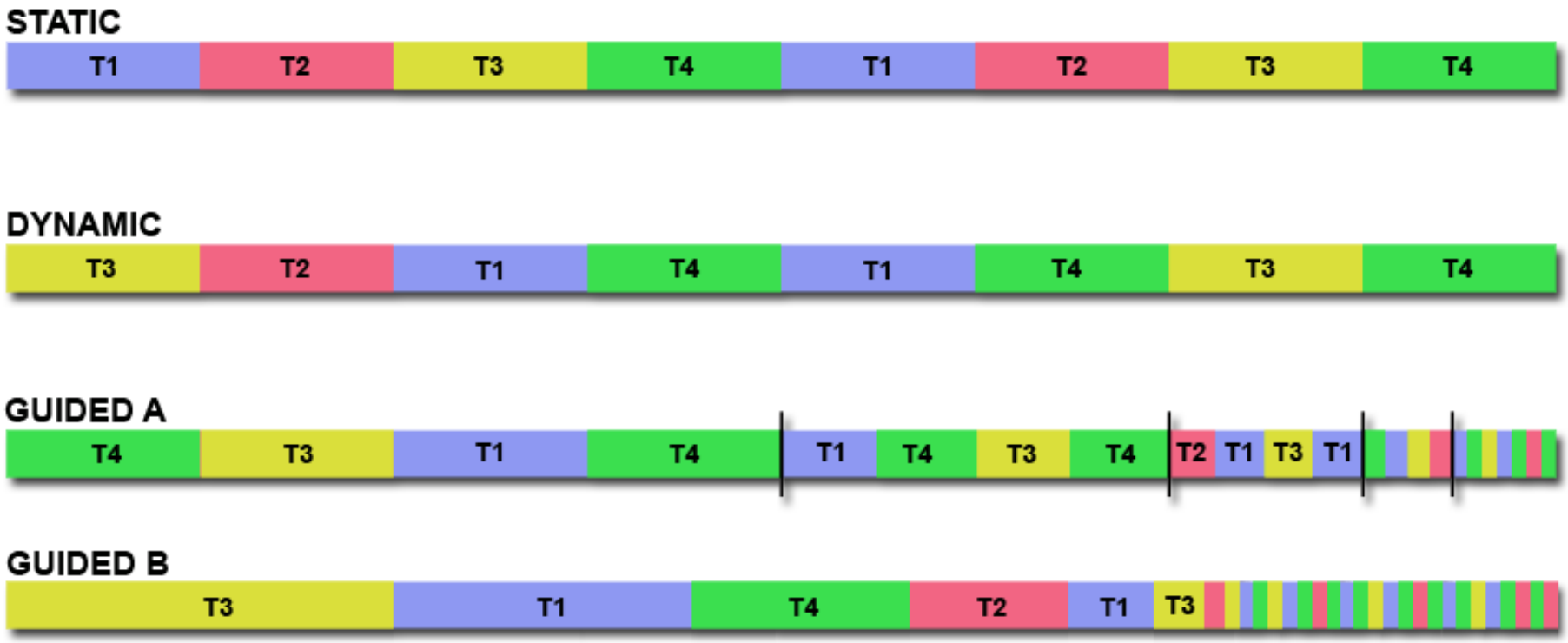


Image source: https://computing.llnl.gov/tutorials/openMP/

Loop parallelism (FOR/DO)

- Additional clauses to synchronize and/or control the execution
  - ORDERED: all iterations are executed in the same order as in the original loop
  - NOWAIT: avoid a synchronization between threads at the end of the loop
    - C/C++: stated in the directive (e.g. #pragma omp do nowait)
    - Fortran: stated at the **end** of the loop (e.g. !$OMP END DO NOWAIT)

Loop parallelism (FOR/DO)

- Most useful classes to deal with data dependencies
  - CRITICAL: definition of a section of code that only one thread can execute at a time
  - ATOMIC: similar to CRITICAL, but applying to specific single statements (e.g. assignments, basic arithmetical operations…)
  - REDUCTION: a list of variables are set as private, and their values are combined with an operator at the end of the associated directive

Loop parallelism (FOR/DO)

- Examples of different clauses: CRITICAL

```
#pragma omp parallel for
for (i=0; i<100; i++)
{
    code executed simultaneously by many threads
    #pragma omp critical
    {
        code executed by only one thread at a time
    }
}
```

Loop parallelism (FOR/DO)

- Examples of different clauses: CRITICAL

```
!$OMP PARALLEL DO
do i=1,100
   code executed simultaneously by many threads
!$OMP CRITICAL
      code executed by only one thread at a time
!$OMP END CRITICAL
enddo
```

Loop parallelism (FOR/DO)

- Examples of different clauses: ATOMIC

```
#pragma omp parallel for
for (i=0; i<100; i++)
{
    code executed simultaneously by many threads
    #pragma omp atomic
    a = a + b;
}
```

Loop parallelism (FOR/DO)

- Examples of different clauses: ATOMIC

```
!$OMP PARALLEL DO
do i=1,100
   code executed simultaneously by many threads
!$OMP ATOMIC
      a = a + b
enddo
```

Loop parallelism (FOR/DO)

- Examples of different clauses: REDUCTION (operator:variable [,operator:variable])

```
#pragma omp parallel for reduction(+:a)
for (i=0; i<100; i++)
{
   code executed simultaneously by many threads
   // variable “a” is modified locally per thread, and then added globally
   a = a + b;
}
```

Loop parallelism (FOR/DO)

- Examples of different clauses: REDUCTION (operator:variable [,operator:variable])

```
!$OMP PARALLEL DO REDUCTION(+:a)
do i=1,100
   code executed simultaneously by many threads
   // variable “a” is modified locally per thread, and then added globally
   a = a + b
enddo
```

Hands-on: coding part

- Try the examples of vector addition and pi
  - C/C++: omp_vectoradd.c, omp_pi.c
  - Fortran: omp_vectoradd.f, omp_pi.c
- Understand how they work and try to use different solutions for dependencies
  - CRITICAL
  - ATOMIC
  - REDUCTION
- Execute the codes with a different number of threads (1, 2, 4, 8)

Task parallelism: SECTIONS

- Simplest way of subdividing non-iterative independent work
- C/C++

```
#pragma omp sections
{
#pragma omp section
{ var_1 = … ; var_2 = … ; }
#pragma omp section
{ var_3 = … ; var_4 = … ; }
}
```

Task parallelism: SECTIONS

- Simplest way of subdividing non-iterative independent work
- Fortran

```
!$OMP SECTIONS
!$OMP SECTION
var_1 = … ; var_2 = …
!$OMP SECTION
var_3 = … ; var_4 = … ;
!$OMP END SECTIONS
```

Directive SECTIONS

- Several clauses can be used here
  - PRIVATE, FIRSTPRIVATE, REDUCTION, NOWAIT…
- All threads are synchronized at the end of the SECTIONS directive
  - … unless NOWAIT is stated
- It is possible to define as many sections inside SECTIONS as desired
  - numthreads >= numsections: one section is executed by one thread, and possibly other threads are idle
  - numthreads < numsections: the OpenMP implementation decides how to schedule the different sections (no specific control)
- Combination with PARALLEL:
  - #pragma omp parallel sections
  - !$OMP PARALLEL SECTIONS

Directive SINGLE

- Sets a code region to be executed by only one thread
- C/C++

```
#pragma omp parallel
{
code executed by all threads
#pragma omp single [nowait]
{ code executed by only one thread }
code executed by all threads
}
```

Directive SINGLE

- Sets a code region to be executed by only one thread
- Fortran

```
!$OMP PARALLEL
code executed by all threads
!$OMP SINGLE
code executed by only one thread
!$OMP END SINGLE [NOWAIT]
code executed by all threads
!$OMP END PARALLEL
```

Combination of SINGLE + construct TASK (TASKLOOP)

- TASK defines an explicit task that may be executed by the originating thread or another in the team
- Possible combination with SINGLE in order to define tasks that will be executed in parallel
- Tasks are generated, and then executed by a thread in the team
- TASKLOOP in OpenMP v. 4.5
- Scheduling dependent on the OpenMP implementation

```
#pragma omp parallel
{
   #pragma omp single
   {
      int i;
      for (i=0; i<LARGE_NUMBER; i++)
         #pragma omp task
         process(item[i]);
   }
}
```

Combination of SINGLE + construct TASK (TASKLOOP)

- TASK defines an explicit task that may be executed by the originating thread or another in the team
- Possible combination with SINGLE in order to define tasks that will be executed in parallel
- Tasks are generated, and then executed by a thread in the team
- TASKLOOP in OpenMP v. 4.5
- Scheduling dependent on the OpenMP implementation

```
!$OMP PARALLEL
!$OMP SINGLE
do i=1,10000000
!$OMP TASK
! i is firstprivate, item is shared
   call process(item(i))
!$OMP END TASK
end do
!$OMP END SINGLE
!$OMP END PARALLEL
```

Other OpenMP constructs

- Further synchronization directives
  - BARRIER: force a general synchronization point among threads
  - TASKWAIT: synchronize all child tasks generated by a given task
  - FLUSH: force a consistent view of memory
- Possible optimization for loop parallelism using FOR/DO
  - COLLAPSE: useful with nested loops to merge iterations for scheduling

```
#pragma omp for collapse(2) private(k, j, i)
for (k=0; k<100; k++)
   for (j=0; j<200; j++)
      for (i=0; i<300; i++)
         process(i,j,k);
```

Other OpenMP constructs

- Further synchronization directives
  - BARRIER: force a general synchronization point among threads
  - TASKWAIT: synchronize all child tasks generated by a given task
  - FLUSH: force a consistent view of memory
- Possible optimization for loop parallelism using FOR/DO
  - COLLAPSE: useful with nested loops to merge iterations for scheduling

```
!$OMP DO COLLAPSE(2) PRIVATE(k, j, i)
do k=1,100
   do j=1,200
      do i=1,300
         process(i,j,k);
```

General good practices

- For all implementation-dependent features, please check the standard
- When defining shared and private variables, use DEFAULT(NONE)
- Lock functions exist in OpenMP, but there should always be a better solution
- Handle the NOWAIT option with care: data dependencies may appear
- The parallel code may not give out the exact same results of the serial one
  - E.g. rounding errors appear because of different execution order in loops
  - Is it possible to relax the requirements for sequential execution?
- Consider the different execution overheads of the directives (and clauses)
  - ORDERED in loops and the creation of PARALLEL and CRITICAL regions is slow
  - Define a very large array as PRIVATE only if it is absolutely needed
  - (A few) ATOMIC calls and REDUCTIONs are generally efficient

Hands-on: performance analysis part

- Parallel computing may be tedious at some point... but there are tools that can help
- Automatic parallelization: code generation and easy optimization
- Try the tool Parallelware Trainer, developed by Appentra
  - Available on Cartesius
  - Explore the possibilities of different parallelizations
- Check the performance of different codes
  - Use a different input size
  - Use a different number of threads (you have up to 24 cores to play around!)

PARALLEL PROGRAMMING WITH OPENMP

Caspar van Leewen
Carlos Teijeiro Barjas