www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

OpenMP Fundamentals
Fork-join model and data environment

Xavier Teruel and Xavier Martorell

EXCELENCIA SEVERO OCHOA

Agenda: OpenMP Fundamentals

OpenMP brief introduction

- overview, a bit of history, main components, execution model, memory model, language syntax

The fork-join model

- creating parallel regions: the parallel construct
- manually distributing work among threads
- sequential code inside the parallel region: the master construct

Data environment

- data-sharing attributes: private and shared data
- data races when sharing variables and critical sections
- data-sharing rules, default attributes in the data environment

OpenMP overview

Parallel Programming Model

- (initially) Designed for shared memory parallel computers
  - » single address space across the host memory system
- But now it also includes multi-device architectures (GPUs, Accelerators,…)
  - » it may imply additional (per device) address spaces
  - » support of data mapping from/to each address space

Maintained by the Architecture Review Board (ARB)

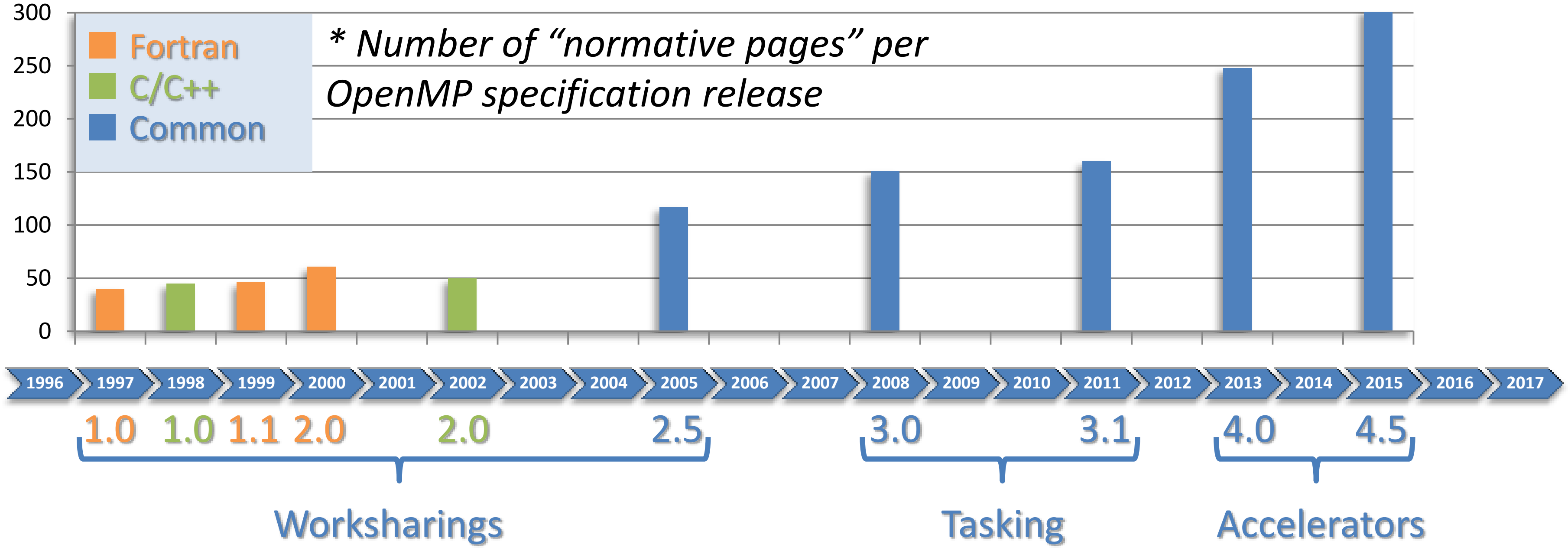
- Permanents members: AMD, ARM, Cray, Fujitsu, HP, IBM, Intel, Micron, NEC, NVIDIA, Oracle, Red Hat and Texas Instruments
- Auxiliary members: ANL, LLNL, BSC, cOMPunity, EPCC, LANL, LBNL, NASA, ORNL, RWTH Aachen University, SNL, TACC and UH

Supported by most compiler vendors

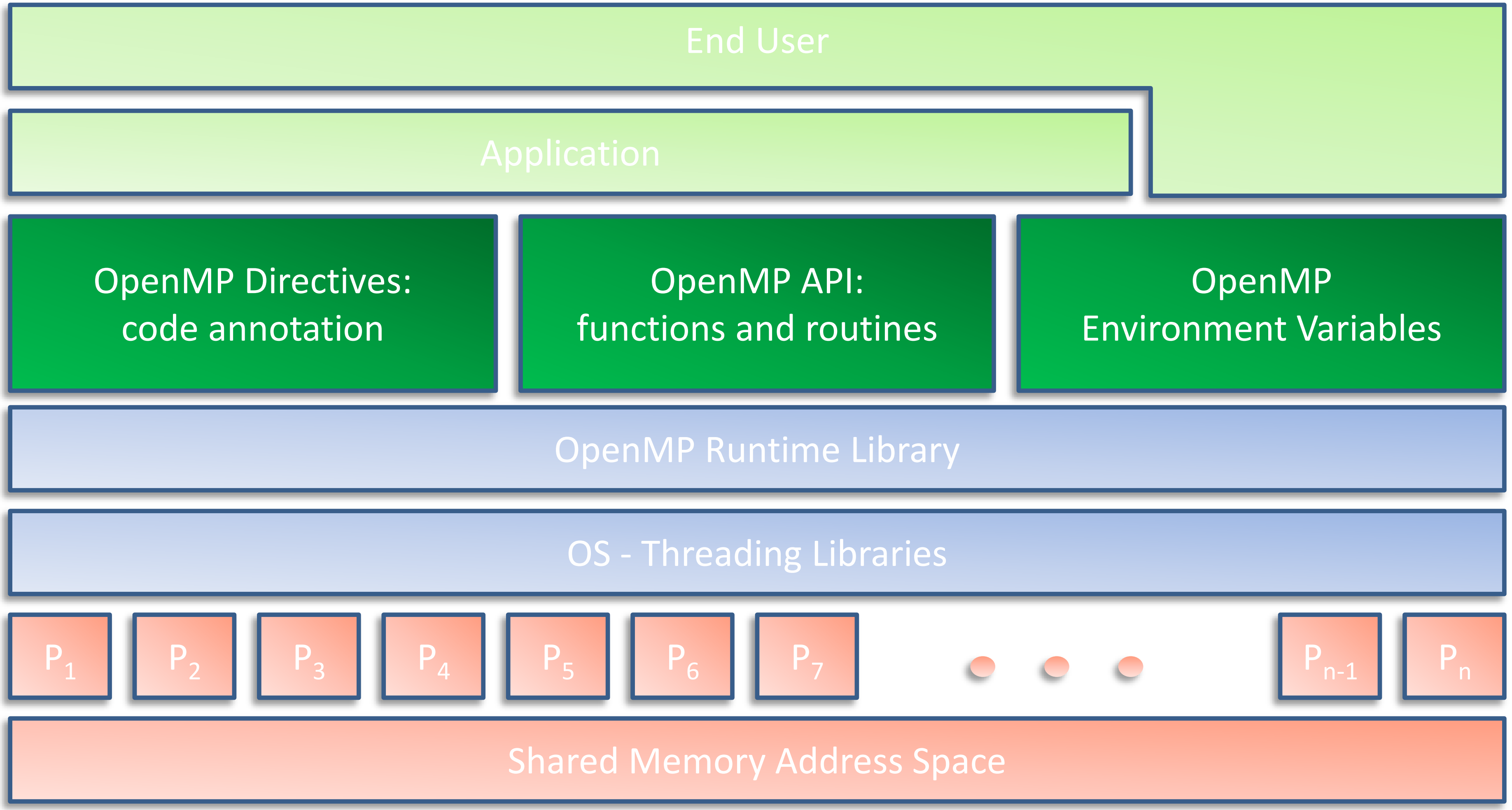
- Intel, IBM, PGI, TI, Sun, Cray, Fujitsu, MS, HP, GCC,…

History of the OpenMP specification

A mature parallel programming model (more than 20 years old)
Complex to face the whole (latest) specification



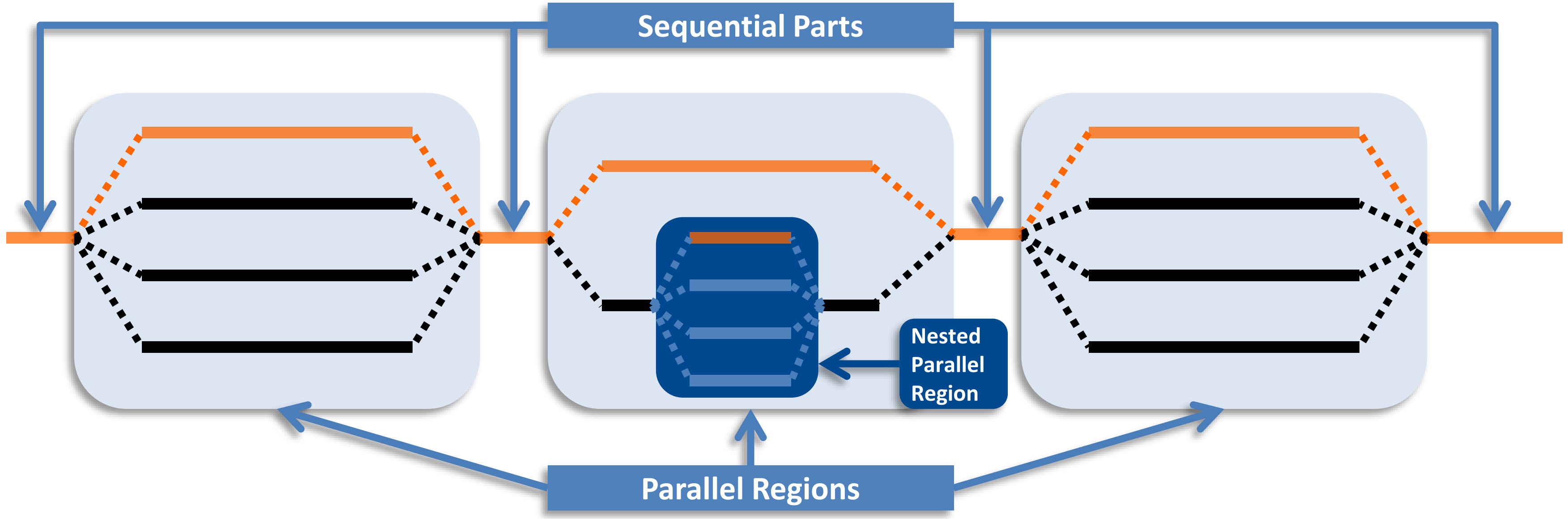
OpenMP components


End User
Application
OpenMP Directives: code annotation
OpenMP API: functions and routines
OpenMP Environment Variables
OpenMP Runtime Library
OS - Threading Libraries
P_1
P_2
P_3
P_4
P_5
P_6
P_7
P_{n-1}
P_n
Shared Memory Address Space

Execution model

Based on the fork-join paradigm

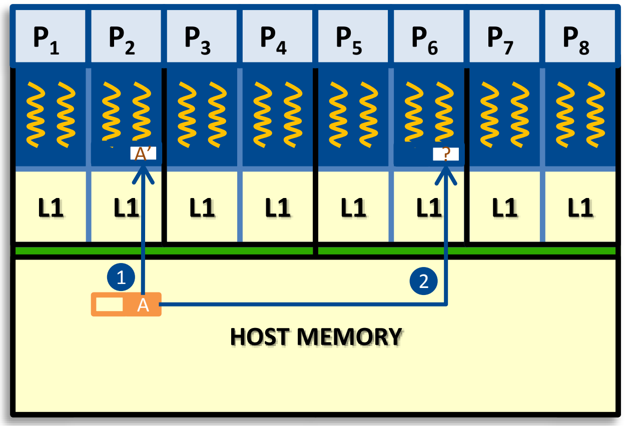
- a thread team is a set of threads which co-operate on a task
- the **master thread** is responsible for coordinating the team
- usually running one thread per processor (but could be more / or less)
- different threads may follow different control flows



Memory model

A relaxed-consistency memory model

- different threads may see different values for the same (shared) variable → not consistent
- consistency is only guaranteed at specific points
  - » explicit points: the flush directive
  - » implicit points: other directives
- luckily, the implicit points are usually enough



The operation enforcing consistency is called the flush operation

- all previous read and writes by this thread have been completed
- all these changes are visible to all other threads
- they are also known as *fences* or *memory barriers*
- In the example: At moment (1) P2 has read the variable A from memory and it has modified it, then at moment (2) P6 wants to read variable A.

OpenMP (directive) syntax

In Fortran language

– through a specially formatted comment

```
sentinel directive-name [clause[[,] clause]...]
```

– where sentinel is one of
  - » !$OMP or C$OMP or *$OMP in fixed format
  - » !$OMP in free format

– API runtime services
  - » omp_lib module contains the subroutine and function definitions

In C/C++ language

– using compiler directives*

```
#pragma omp directive-name [clause[[,] clause]...]
```

– API runtime services
  - » omp.h contains the API prototypes and data types definitions

** directives are ignored if compiler does not recognize OpenMP*

The structured block

Most directives apply to a structured block:

```
#pragma omp directive-name [clause[[,] clause]...]
structured-block
```

- block of one or more statements with one entry point / one exit point
  - » i.e. branching in or out is not allowed
  - » terminating the program is allowed (abort/exit)

```
#pragma omp directive-name clause1(…) clause2(…)
{
  set_of_instructions (no  branch in/out);
}
```
✓

```
#pragma omp directive-name clause1(…) clause2(…)
for (int i = 0;  i < SIZE; i++) {
    A [ i ] = 0;
}
```
✓

```
#pragma omp directive-name clause1(…) clause2(…)
{
  set_of_instructions;
  if ( expr ) exit(0);
}
```
✓

```
#pragma omp directive-name clause1(…) clause2(…)
for (int i = 0;  i < SIZE; i++) {
    A [ i ] = 0;
    if ( i == INDEX ) break;
}
```
✗

www.bsc.es

The Fork-Join Model

Parallel programming with OpenMP

The parallel construct

Creating a parallel region

– always attached to a structured block

```
#pragma omp parallel [clause[[,] clause]...]
{structured-block}
```

Where clause:

– num_threads (expression)
– if (expression)
– shared (var-list)
– private (var-list)
– firstprivate (var-list)
– default (dtype)
– reduction (var-list)

Specifying the number of threads

The maximum number of threads is controlled by

– an internal control variable (ICV) called nthreads-var
  - » the OpenMP API nthreads-var setter

```
void omp_set_num_threads (int value);   // subsequent parallel region
```

  - » the OpenMP API nthreads-var getters

```
int omp_get_num_threads (void);   // current team number of threads
int omp_get_max_threads (void);   // maximum number of threads
```

  - » the OpenMP environment variable nthreads-var setter

```
$ export OMP_NUM_THREADS=<list>
$ ./myProgram
```

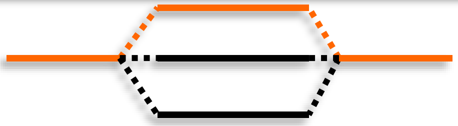
– the num_threads clause (overriding nthreads-var value)

Example: creating a parallel region (1)

Creating a parallel region of 3 threads (num_threads clause)

```
#include <stdio.h>

void main (void)
{
  #pragma omp parallel num_threads(3)
  {
    printf("Hello world!\n");
  }
}
```

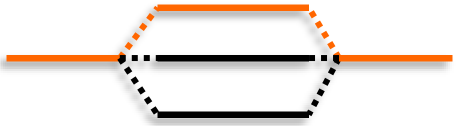
```
$ gcc -fopenmp myHello.c -o myHello
$ ./myHello
Hello world!
Hello world!
Hello world!
```



Creating a parallel region of 3 threads (omp_set_num_threads)

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

void main (void)
{
  omp_set_num_threads(3);
  #pragma omp parallel
  {
    printf("Hello world!\n");
  }
}
```

```
$ gcc -fopenmp myHello.c -o myHello
$ ./myHello
Hello world!
Hello world!
Hello world!
```

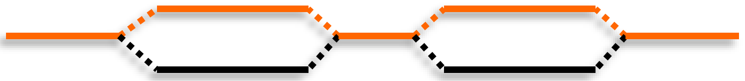


Example: creating a parallel region (3)

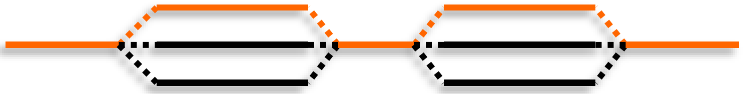
But still more useful is to use the environment variable

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

void main (void)
{

  #pragma omp parallel
  {
    printf("Hello world...\n");
  }

  #pragma omp parallel
  {
    printf("...and godbye!\n");
  }

}
```

```
$ gcc -fopenmp myHello.c -o myHello
$ OMP_NUM_THREADS=2 ./myHello
Hello world...
Hello world...
...and goodbye!
...and goodbye!
```



```
$ OMP_NUM_THREADS=3 ./myHello
Hello world...
Hello world...
Hello world...
...and goodbye!
...and goodbye!
...and goodbye!
```

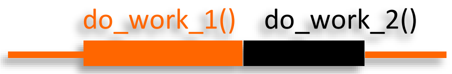


Replicate work inside the parallel region

When two "blocks of code" may run in parallel…

```
#include <stdio.h>

void main (void)
{
  do_work_1();
  do_work_2();
}
```

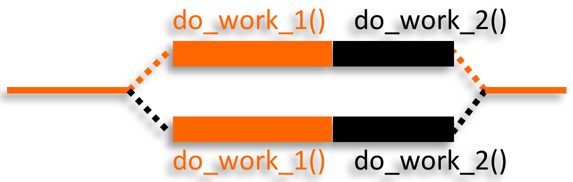
```
$ time ./myProgram
real    0m4.003s
user    0m0.000s
sys     0m0.000s
```



… we just include them within a parallel region (replicate)

```
#include <stdio.h>
#include <omp.h>
void main (void)
{
  #pragma omp parallel num_threads(2)
  {
    do_work_1();
    do_work_2();
  }
}
```

```
$ time ./myProgram
real    0m4.104s
user    0m0.000s
sys     0m0.000s
```



Identifying threads inside the parallel region

Inside a parallel region each thread has its own identifier

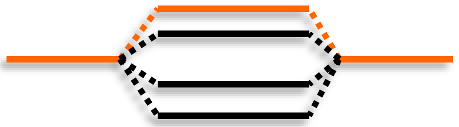
```
int omp_get_thread_num (void);   // get the identification number for the current thread/team
```

- from 0 to N-1 (where N is the number of threads of the team)
- master thread is always identified by 0 (zero)
- routine returns 0 (zero) if called outside a parallel region

Example using the thread identifier

```
#include <stdio.h>
#include <omp.h>
void main (void)
{
  #pragma omp parallel num_threads(4)
  {
    int id = omp_get_thread_num();
    printf("Hello world! I am the thread %d.\n“, id);
  }
}
```

```
$ ./myThreadId
Hello world! I am the thread 2.
Hello world! I am the thread 1.
Hello world! I am the thread 0.
Hello world! I am the thread 3.
```

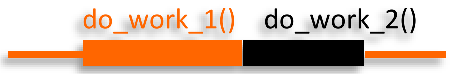


Distribute work inside the parallel region (1)

When two “blocks of code” may run in parallel…

```c
#include <stdio.h>

void main (void)
{
  do_work_1();
  do_work_2();
}
```

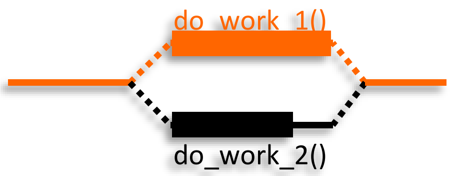
```
$ time ./myProgram
real    0m4.003s
user    0m0.000s
sys     0m0.000s
```



… we can use the thread identifier to distribute work

```c
#include <stdio.h>
#include <omp.h>
void main (void)
{
  #pragma omp parallel num_threads(2)
  {
    int id = omp_get_thread_num();
    if ( id == 0 ) do_work_1();
    if ( id == 1 ) do_work_2();
  }
}
```

```
$ time ./myProgram
real    0m2.604s
user    0m0.000s
sys     0m0.000s
```



Distribute work inside the parallel region (2)

Thread identifier must be carefully used

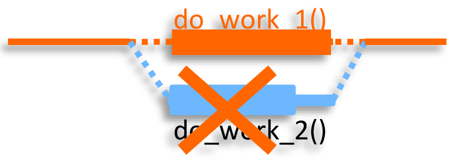
– Rely on the number of threads is never a good idea

– OpenMP offers other mechanisms to distribute work

The following example is actually wrong

```c
#include <stdio.h>
#include <omp.h>
void main (void)
{
  #pragma omp parallel
  {
    int id = omp_get_thread_num();
    if ( id == 0 ) do_work_1();
    if ( id == 1 ) do_work_2();
  }
}
```

```
$ export OMP_NUM_THREADS=1
$ time ./myProgram
real    0m2.604s
user    0m0.000s
sys     0m0.000s
```

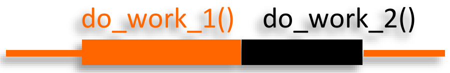


Distribute work inside the parallel region (3)

Workaround to the unassigned work problem

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
  if ( id == 0 ) do_work_1();
  if ( id == 1  || omp_get_num_threads() < 2) do_work_2();
}
```

```
$ export OMP_NUM_THREADS=1
$ time ./myProgram
real    0m4.003s
user    0m0.000s
sys     0m0.000s
```



But still non-optimal solution

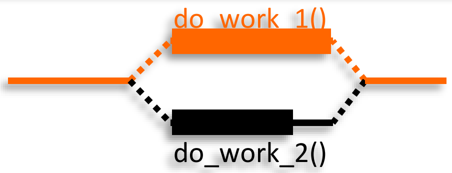
– Think on more than 2 sections?

WARNING!!!

Don't try this at home*

** But you can use it during this tutorial*

```
$ export OMP_NUM_THREADS=2
$ time ./myProgram
real    0m2.604s
user    0m0.000s
sys     0m0.000s
```

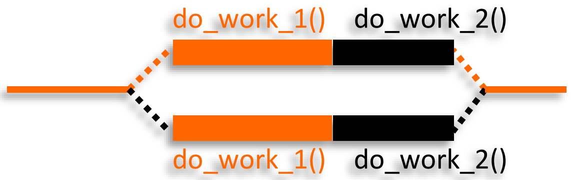


Summary: replicate vs distribute work

Replicate work (all threads execute the same work)

```c
#include <stdio.h>
#include <omp.h>
void main (void) {
  #pragma omp parallel num_threads(2)
  {
    do_work_1();
    do_work_2();
  }
}
```

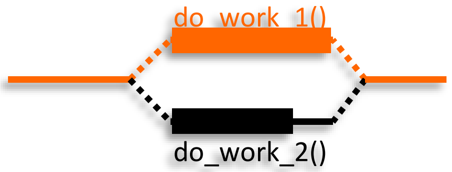
```
$ time ./myProgram
real    0m2.604s
user    0m0.000s
sys     0m0.000s
```



Distribute work (threads “share” the amount of work)

```c
#include <stdio.h>
#include <omp.h>
void main (void) {
  #pragma omp parallel num_threads(2)
  {
    int id = omp_get_thread_num();
    if ( id == 0 ) do_work_1();
    if ( id == 1 ) do_work_2();
  }
}
```

```
$ time ./myProgram
real    0m2.604s
user    0m0.000s
sys     0m0.000s
```



Distribute loop iterations inside the parallel region

Target: independent loop

```
#define SIZE 1204
double A[SIZE];
void main (void)
{
    for (int i = 0;  i < SIZE; i++) {
     A [ i ] = 0;
    }
}
```

— Programmer must guarantee no dependences across loop iterations

— Compute lower bound and upper bound for each thread (using actual boundaries, thread id and number of threads)

Parallel approach

```
#include <omp.h>
#define SIZE 1204
double A[SIZE];
void main (void)
{
  #pragma omp parallel
  {
    int id = omp_get_thread_num();
    int nt = omp_get_num_threads();
    int lb = id * (SIZE/nt);
    int ub = (id+1)*(SIZE/nt) + ( (id==nt-1)? (SIZE%nt) : 0 );
    for (int i = lb; i < ub; i++) {
     A [ i ] = 0;
    }
  }
}
```

— But still non-optimal solution
  » more threads than iterations
  » load imbalance (iters/threads)

Parallel construct: the if clause

Avoids creating parallel regions

```
#pragma omp parallel if(expr)
{structured-block}
```

- sometimes we only want to run in parallel under certain conditions
- if expr evaluates to false parallel construct will only use 1 thread
- still creates a new team and data environment

Example of the if-clause usage

```
#include <omp.h>
#define SIZE ...
double A[SIZE];
void main (void)
{
  #pragma omp parallel if(SIZE>256)
  {
    int id = omp_get_thread_num();
    int nt = omp_get_num_threads();
    int lb = id*(SIZE/nt);
    int ub = (id+1)*(SIZE/nt)+((id==nt-1)?(SIZE%nt):0);
    for (int i = lb; i < ub; i++) {
     A [ i ] = 0;
    }
  }
}
```

Master construct

Only the master thread executes a given region

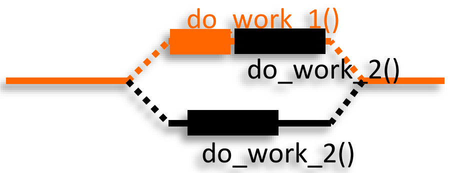
```
#pragma omp master
{structured-block}
```

- the master construct has no clauses

Master construct's semantics

- other threads do not execute the structured block
- there is no implicit barrier at the entry
- there is no implicit barrier at the end

```
#pragma omp parallel num_threads(2)
{
  int id = omp_get_thread_num();
  #pragma omp master
  do_work_1(); // execute with one thread
  do_work_2(id); // execute with N threads
}
```



www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Data Environment

Parallel programming with OpenMP

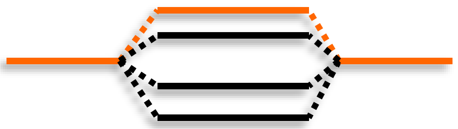
OpenMP constructs and data environment

Scoping variables in an OpenMP construct (ownership)

- determine the scope for each variable: shared and private
- shared data can be accessed by all the threads
- private data can only be accessed by the owner thread

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

double PI = 3.14159265359;

void main (void)
{
  int id = 0;
  #pragma omp parallel num_threads(4) shared(PI) private(id)
  {
    id = omp_get_thread_num();
    printf("Hello world! I am thread %d. I like %f.\n", id, PI);
  }
}
```

```
$ ./myProgram
Hello world! I am thread 2. I like 3.141593.
Hello world! I am thread 0. I like 3.141593.
Hello world! I am thread 1. I like 3.141593.
Hello world! I am thread 3. I like 3.141593.
```



Privatizing variables inside the construct

The variable inside the construct is a new variable

- the new variables have the same type than original variable
- in parallel construct it means all threads have a different variable
- they can be accessed without any kind of synchronization

The private (storage) and firstprivate (storage + copy) clauses

```
#pragma omp parallel {private|firstprivate}(list)
{structured-block}
```

- private variables have undefined value when starting the block
- firstprivate variables are initialized to the value of the original one

```
double PI = 3.14159265359;

#pragma omp parallel private(PI)
{
   PI = <expr>;
   ...
}
printf("PI = %f \n", PI);
```

```
double PI = 3.14159265359;

#pragma omp parallel firstprivate(PI)
{
     <lvalue> = f(PI); // including PI = f(PI);
     ...
}
printf("PI = %f \n", PI);
```

The threadprivate directive

Allows to create a per-thread copy of “global” variables

```
#pragma omp threadprivate(var-list)
```

threadprivate can be applied to:
- global or static variables
- class static data members (C++)

The threadprivate storage persist
- but persistence is complex

```
#include <stdio.h>
char buffer[SIZE];
#pragma omp threadprivate(buffer)

void main (void)
{
  #pragma omp parallel
  {
    buffer = <expr>;
    . . .
  }
}
```

Now buffer have a per-thread copy (~private)

Using *static* variable:

```
#include <stdio.h>

char* foo(void)
{
  static char buffer[SIZE];
  #pragma omp threadprivate(buffer)
  . . .
  return buffer;
}

void main (void)
{
  #pragma omp parallel
  {
    char *a = foo();
    . . .
  }
}
```

Now foo() can be called by multiple threads at the same time

Returns correct address to caller

Sharing variables inside the parallel region (1)

The variable is “the same” outside/inside the construct

- in parallel construct it means all threads see the same variable (address)
- but not necessarily the same value (consistency issue)
- usually need some kind of synchronization to update them correctly
  - » synchronization: mutual exclusion or atomic updates
  - » synchronization also guarantees consistency points

```c
#include <stdio.h>

double PI = 3.14159265359;

void main (void)
{
  int id = 0;
  #pragma omp parallel num_threads(4) shared(PI)
  {
    PI = 3;
  }
  printf("PI = %f \n", PI);
}
```

- all threads read same variable
- after the parallel region variable modifications still are visible

```
$ ./myProgram
PI = 3.000000;
```

Sharing variables inside the parallel region (2)

Modifying shared variables ('a' and 'b') inside the parallel region

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

int a = 0, b = 0 , ITERS = 100;
void main (void)
{
  int NT = 4; // Number of threads
  #pragma omp parallel num_threads(NT) shared(a, b, NT, ITERS)
  {
    #pragma omp master
    a = NT*ITERS;

    for (int i = 0; i<ITERS; i++) {
      b = b + 1;
    }
  }
  assert ( a == NT*ITERS, "Value of 'a' is incorrect!!!")  // correct
  assert ( b == NT*ITERS, "Value of 'b' is incorrect!!!")  // incorrect
}
```

- variables 'NT' & 'ITERS' have no data race
- variable 'a' has no data race
- variable 'b' may give incorrect results
- a bit of assembly… b = b + 1 → load, arithmetic-op(+) and store
- An example of two 'b=b+1' executed concurrently: b=5, ((b+1)+1) → b=7

Reg-1	Thread-1	b	Thread-2	Reg-2
r1=5	load b, r1	5		r1=X
r1=6	increment r1	5		r1=X
r1=6		5	load b, r1	r1=5
r1=6	store r1, b	6		r1=5
r1=6		6	increment r1	r1=6
r1=6		6	store r1, b	r1=6

The critical construct

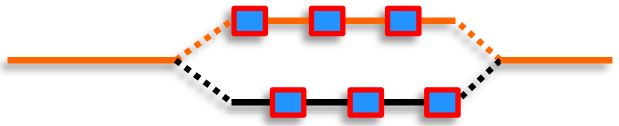
Mutual exclusion regions

```
#pragma omp critical [(name) [hint(hint-expression)] ]
{structured-block}
```

Critical construct's semantics

– only one thread can be executing the region at any given time
– by default all critical regions are synchronized all-to-all
– if you provide a name only those regions with the same name synchronize

```
#include <omp.h>
void main (void)
{
  int b = 0, NT = 4 , ITERS = 100;

  #pragma omp parallel num_threads(NT) shared(b, ITERS)
  for (int i = 0; i<ITERS; i++) {
    #pragma omp critical
    b = b + 1;
  }
  assert ( b == NT*ITERS, "Value of 'b' is incorrect!!!"); // correct
}
```



– in this example we would get a extremely poor performance:
– almost all the code has been serialized!!!
– … but this is a well-know pattern

The reduction pattern and the manual approach

All threads are accumulating values into a single variable

problem

```
#include <omp.h>
void main (void) {
  int b = 0, NT = 2;
  omp_set_num_threads(NT);
  #pragma omp parallel shared(b, NT)
  {
    for (int i = 0; i<ITERS; i++) {
      b = b + 1;
    }
  }
  assert ( b == NT*ITERS, "Value of 'b' is incorrect!!!")
}
```

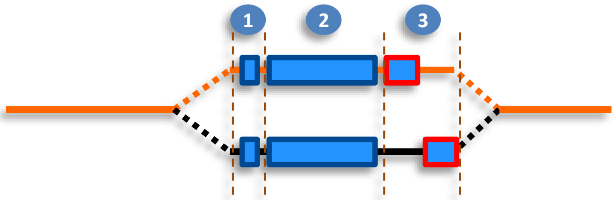
Data Race!!!

solution

```
#include <omp.h>
void main (void)
{
  int b = 0, NT = 2;
  omp_set_num_threads(NT);
  #pragma omp parallel shared(b, NT)
  {
1   int p_b = 0;
    for (int i = 0; i<ITERS; i++) {
2     p_b = p_b + 1;
    }
    #pragma omp critical
3   b = b + p_b;
  }
  assert ( b == NT*ITERS, "Value of 'b' is incorrect!!!")
}
```

The manual approach:

1. create-initialize a *per-thread* copy
2. accumulate partial results using this private copy (no synchro)
3. accumulate each partial results into the original variable (critical)



Parallel construct and the reduction mechanism

The reduction clause

```
#pragma omp parallel reduction(operator:list)
{structured-block}
```

Applying it to previous example (data-sharing attribute)

```
#include <omp.h>
void main (void)
{
  int b = 0, NT = 2;
  omp_set_num_threads(NT);
  #pragma omp parallel reduction(+:b)
  for (int i = 0; i<ITERS; i++) {
      b = b + 1;
  }
  assert ( b == NT*ITERS, "Value of 'b' is incorrect!!!")
}
```

– the compiler creates a private copy that is properly initialized (identity)
– the compiler ensures that the shared variable is properly (and safely) updated with all partial results
– valid operators are: `+, -, *, |, ||, &, &&, ^, min, max`
– but we can also specify user-defined reductions

This doesn't mean that all data races are solved with reduction!!!

Data environment: what is the default?

Pre-determined data-sharing attributes

- threadprivate variables are threadprivate
- dynamic storage duration objects are shared (malloc, new,…)
- static data members are shared
- variables declared inside the construct
  - » static storage duration variables are shared
  - » automatic storage duration variables are private
- the loop iteration variable(s)…

Explicit data-sharing clauses (shared, private, firstprivate,…)

- If default clause present, what the clause says
  - » none means that the compiler will issue an error if the attribute is not explicitly set by the programmer (very useful!!!)

Implicit data-sharing rules, depends on the construct

- For the parallel region the default is shared

Default data-sharing attributes (in practice)

Data-sharing attribute for each variable referenced in parallel?

```
int a ;
void foo ( int b ) {
  int c;
  #pragma omp parallel
  {
    int d ;
    a = <expr>;
    b = <expr>;
    c = <expr>;
    d = <expr>;
  }
}
```

– default(none) may help when you are not sure of understand the default

Summary: OpenMP fundamentals

OpenMP constructs: parallel, master and critical

- fork-join model: the **parallel region** → team of threads
- how to **replicate** and (manually) **distribute** work among threads

```
#pragma omp parallel
{
  do_work_1();
  do_work_2();
}
```

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
  if ( id == 0 ) do_work_1();
  if ( id == 1 || NT < 2) do_work_2();
}
```

```
#pragma omp parallel
  {
    int id = < expr >, nt = < expr >;
    int lb = id * (SIZE/nt);
    int ub = (id+1)*(SIZE/nt) + ( (id==nt-1)? (SIZE%nt) : 0 );
    for (int i = lb; i < ub; i++) A [ i ] = 0;
}
```

- restrictions inside the parallel region: **master** and **critical** constructs

The data environment: data sharing clauses

- scoping variables inside a construct: **private** and **shared**
- data sharing attribute **rules**: pre-determined, explicit and implicit determined
- the **data race** problem: no controlled access on shared variables
- using **reduction** variables: partial results reduced into original variable

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Intellectual Property Rights Notice

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Thank you!

For further information please visit/contact

http://www.linkedin.com/in/xteruel

xavier.teruel@bsc.es