22nd Summer School on PARALLEL COMPUTING

Introduction to Standard OpenMP 3.1

Massimiliano Culpo - m.culpo@cineca.it

Gian Franco Marras - g.marras@cineca.it

Marco Rorro - m.rorro@cineca.it

CINECA - SuperComputing Applications and Innovation Department

CINECA

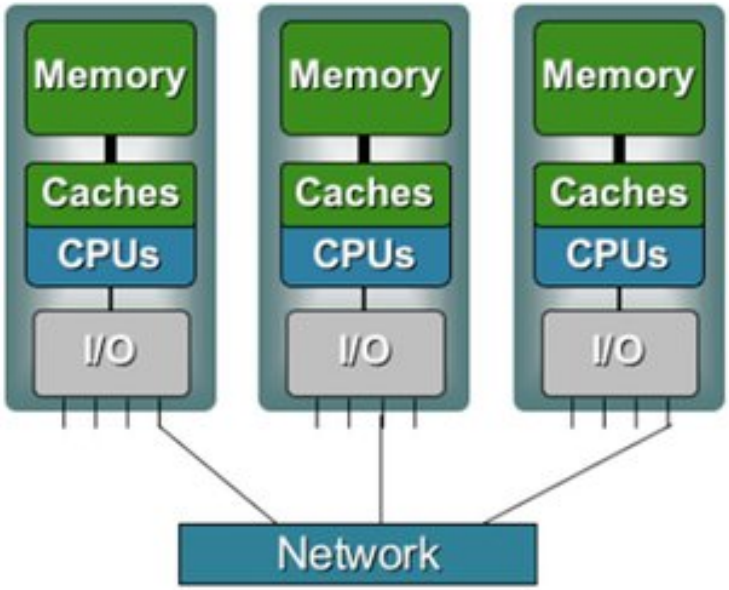
Outline

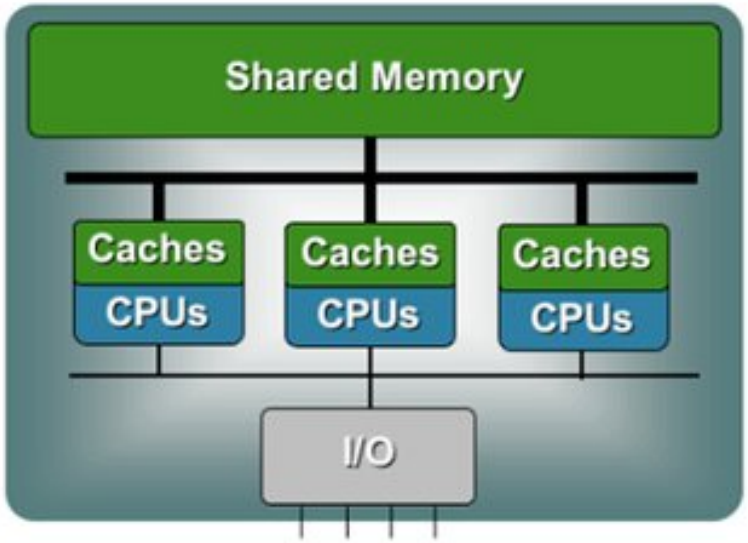
1 Introduction

2 Directives

3 Runtime library routines and environment variables

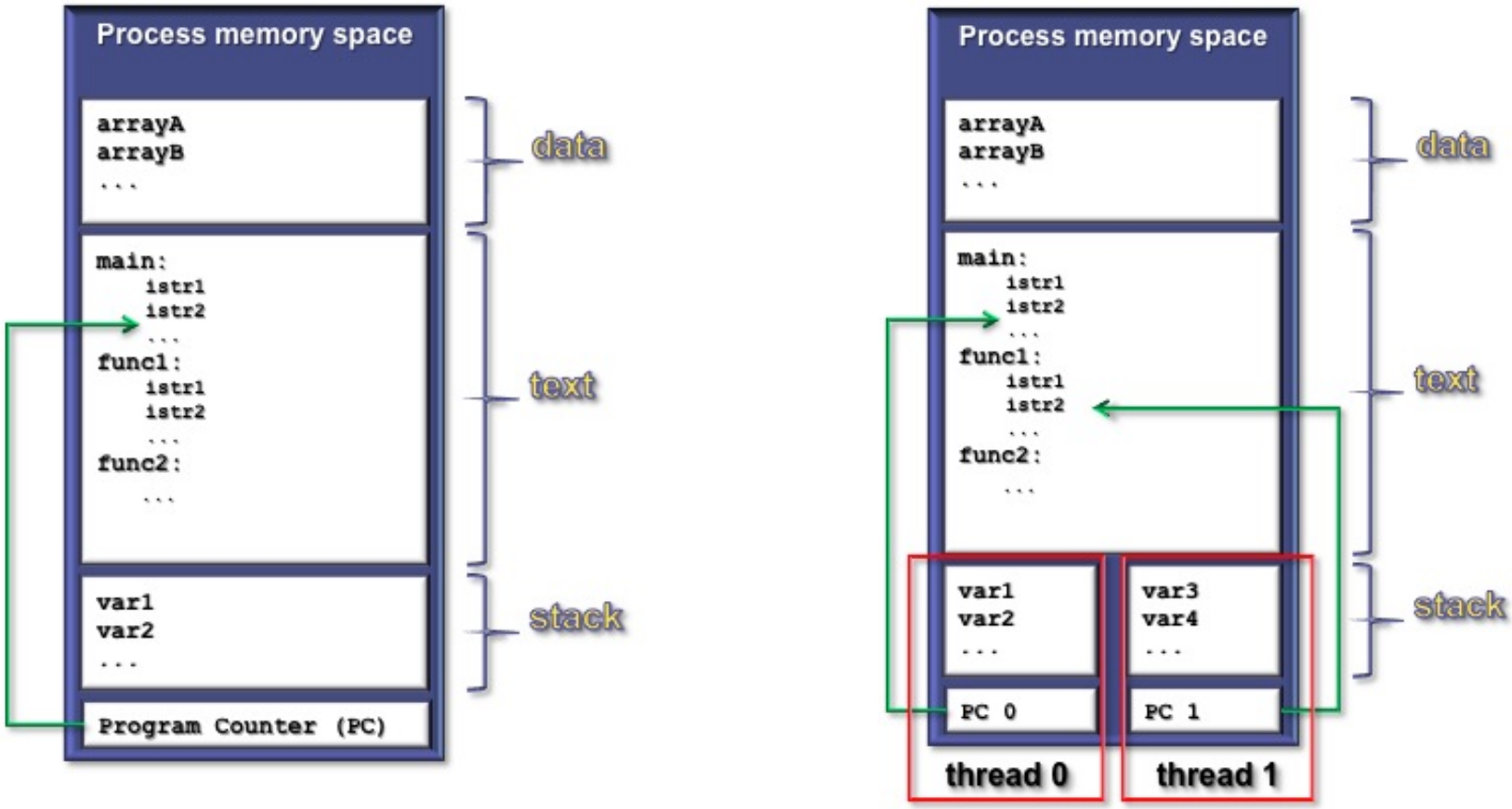
Shared memory systems





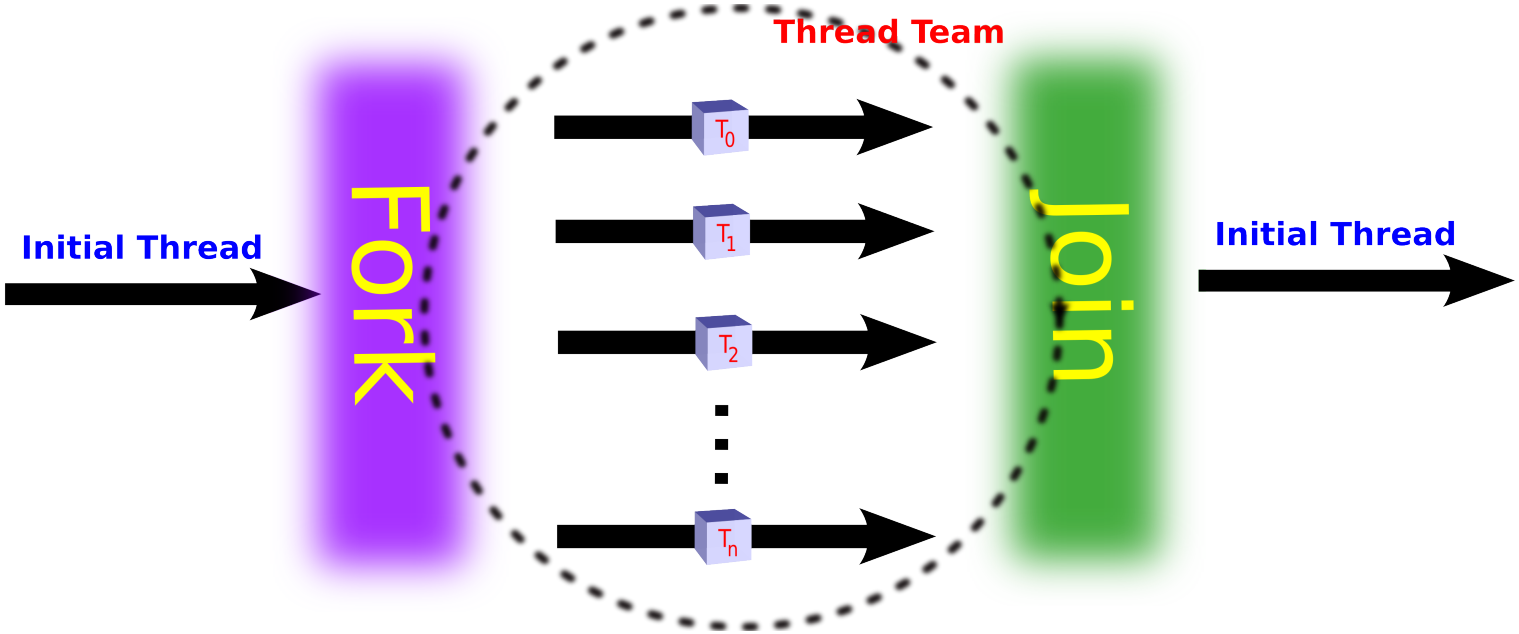
1. Memory is shared among a pool of threads
2. Distinction between UMA / NUMA systems

Multi-threaded processes



1. Each thread may be regarded as a concurrent execution flow

Execution model



1. Fork-join model of parallel execution
2. Work may be shared in the parallel regions of execution
3. Constructs to manage data-sharing and synchronization

Why should I use OpenMP?

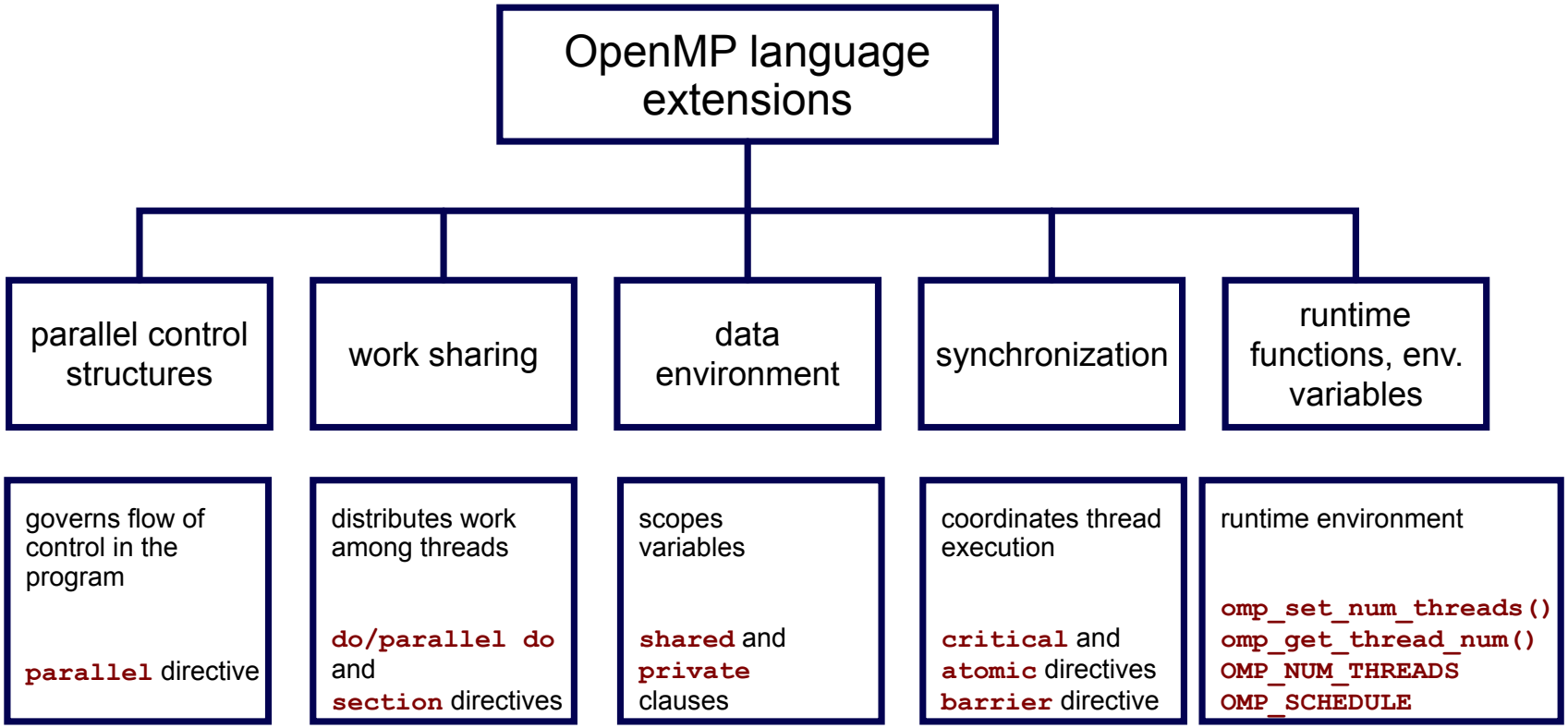
1. **Standardized**
   - enhance portability
2. **Lean and mean**
   - limited set of directives
   - fast code parallelization
3. **Ease of use**
   - parallelization is incremental
   - coarse / fine parallelism
4. **Portability**
   - `C`, `C++` and `Fortran` API
   - part of many compilers

1. **Performance**
   - may be non-portable
   - increase memory traffic
2. **Limitations**
   - shared memory systems
   - mainly used for loops

Structure of an OpenMP program

1. **Execution model**
   - fork-join parallel execution
   - the program starts with an initial thread
   - when a `parallel` construct is encountered a team is created
   - `parallel` regions may be nested arbitrarily
   - worksharing constructs permit to divide work among threads
2. **Shared-memory model**
   - all threads have access to the memory
   - each thread is allowed to have a temporary view of the memory
   - each thread has access to a thread-private memory
   - two kinds of data-sharing attributes: private and shared
   - data-races trigger undefined behavior
3. **Programming model**
   - compiler directives + environment variables + run-time library

OpenMP core elements



OpenMP releases

October 1997	`Fortran` 1.0
October 1998	`C` and `C++` 1.0
November 2000	`Fortran` 2.0
March 2002	`C` and `C++` 2.0
May 2005	`Fortran`, `C` and `C++` 2.5
May 2008	`Fortran`, `C` and `C++` 3.0
July 2011	`Fortran`, `C` and `C++` 3.1
July 2013	`Fortran`, `C` and `C++` 4.0

Outline

1 Introduction

2 Directives

3 Runtime library routines and environment variables

Conditional compilation

C/C++

```
#ifdef _OPENMP
printf("OpenMP support:%d",_OPENMP);
#else
printf("Serial execution.");
#endif
```

Fortran

```
!$ print *,"OpenMP support"
```

1. The macro `_OPENMP` has the value `yyyymm`
2. `Fortran 77` supports `!$`, `*$` and `c$` as sentinels
3. `Fortran 90` supports `!$` only

Directive format

C/C++

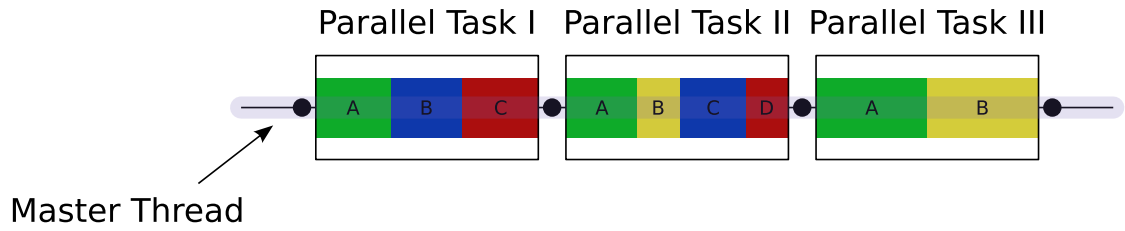
```
#pragma omp directive-name [clause...]
```

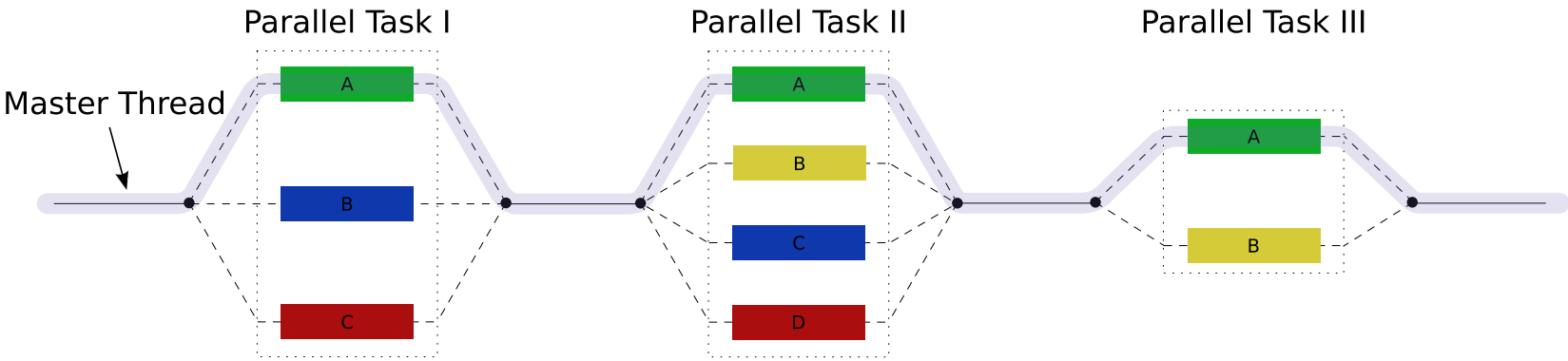
Fortran

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

1. Follows conventions of `C` and `C++` compiler directives
2. From here on free-form directives will be considered

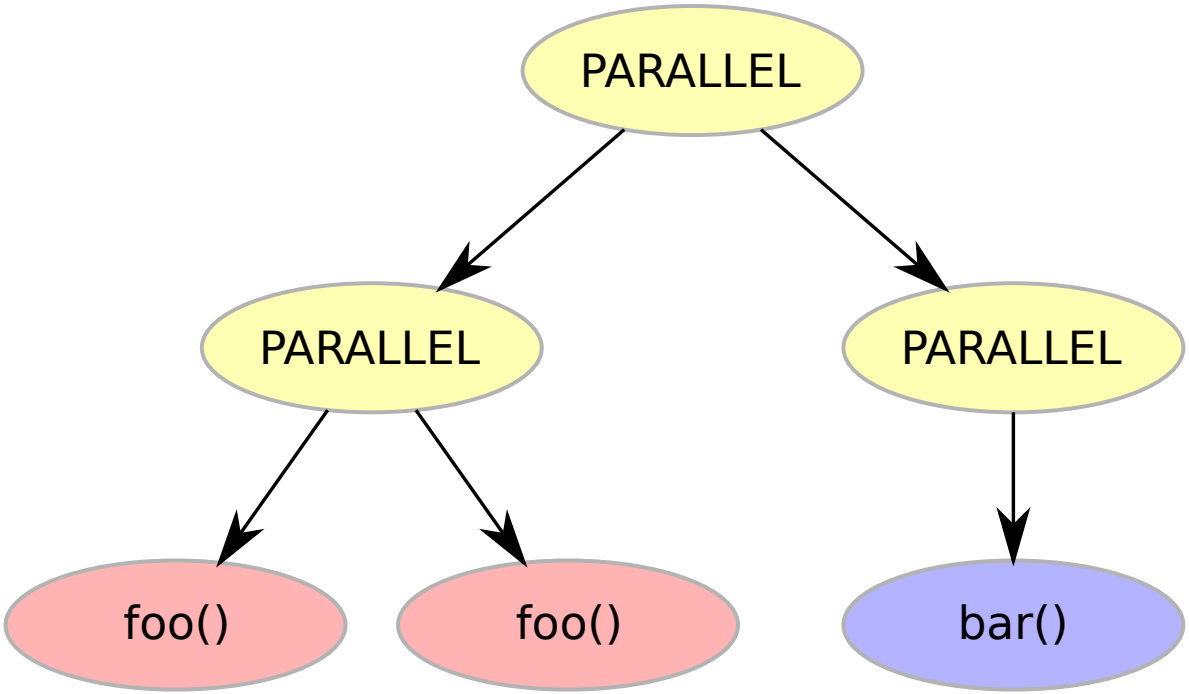
`parallel` construct





1. The encountering thread becomes the master of the new team
2. All threads execute the parallel region
3. There is an implied barrier at the end of the parallel region

Nested parallelism



1. Nested parallelism is allowed in OpenMP 3.1
2. Most constructs bind to the innermost parallel region

OpenMP: Hello world

C/C++

```
int main () {



    printf("Hello world\n");



  return 0;
}
```

OpenMP: Hello world

C/C++

```
int main () {
  /* Serial part */

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

  /* Serial part */
  return 0;
}
```

OpenMP: Hello world

Fortran

```
PROGRAM HELLO

  Print *, "Hello World!!!"

END PROGRAM HELLO
```

OpenMP: Hello world

Fortran

```
PROGRAM HELLO
! Serial code

!$OMP PARALLEL
  Print *, "Hello World!!!"
!$OMP END PARALLEL

! Resume serial code

END PROGRAM HELLO
```

OpenMP: Hello world

What's wrong?

```
int main() {
  int ii;
#pragma omp parallel
  {
    for(ii = 0; ii < 10; ++ii)
      printf("iteration %d\n", i);
  }
  return 0;
}
```

Worksharing constructs

1. Distribute the execution of the associated region
2. A worksharing region has no barrier on entry
3. An implied barrier exists at the end, unless `nowait` is present
4. A `nowait` clause may omit the implied barrier
5. Each region must be encountered by all threads or none at all
6. Every thread must encounter the same sequence of:
   - worksharing regions
   - barrier regions
7. The OpenMP API defines four worksharing constructs:
   - loop construct
   - `sections` construct
   - `single` construct
   - `workshare` contruct

Loop construct: syntax

C/C++

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

Fortran

```
!$omp do [clause[[,] clause] ... ]
  do-loops
[!$omp end do [nowait] ]
```

Loop construct: restrictions

C/C++

```
for (init-expr; test-expr; incr-expr)
  structured-block

init-expr:  var = lb
            integer-type var = lb

test-expr:  relational expr.

incr-expr:  addition or subtraction expr.
```

Loop construct: the rules

1. The iteration variable in the `for` loop
   - if shared, is implicitly made private
   - must not be modified during the execution of the loop
   - has an unspecified value after the loop
2. The `schedule` clause:
   - may be used to specify how iterations are divided into chunks
3. The `collapse` clause:
   - may be used to specify how many loops are parallelized
   - valid values are constant positive integer expressions

Loop construct: fast quiz

Right or wrong?

```
SUBROUTINE DO_LOOP
   INTEGER I, J
   DO 100 I = 1,10
      !$OMP DO
      DO 100 J = 1,10
         CALL WORK(I,J)
      100 CONTINUE
      !$OMP ENDDO
END SUBROUTINE DO_LOOP
```

Loop construct: scheduling

C/C++

```
#pragma omp for schedule(kind[, chunk_size])
  for-loops
```

Fortran

```
!$omp do schedule(kind[, chunk_size])
  do-loops
[!$omp end do [nowait] ]
```

Loop construct: schedule kind

1. **Static**
   - iterations are divided into chunks of size `chunk_size`
   - the chunks are assigned to the threads in a round-robin fashion
   - must be reproducible within the same parallel region
2. **Dynamic**
   - iterations are divided into chunks of size `chunk_size`
   - the chunks are assigned to the threads as they request them
   - the default `chunk_size` is 1
3. **Guided**
   - iterations are divided into chunks of decreasing size
   - the chunks are assigned to the threads as they request them
   - `chunk_size` controls the minimum size of the chunks
4. **Run-time**
   - controlled by environment variables

Loop construct: schedule kind

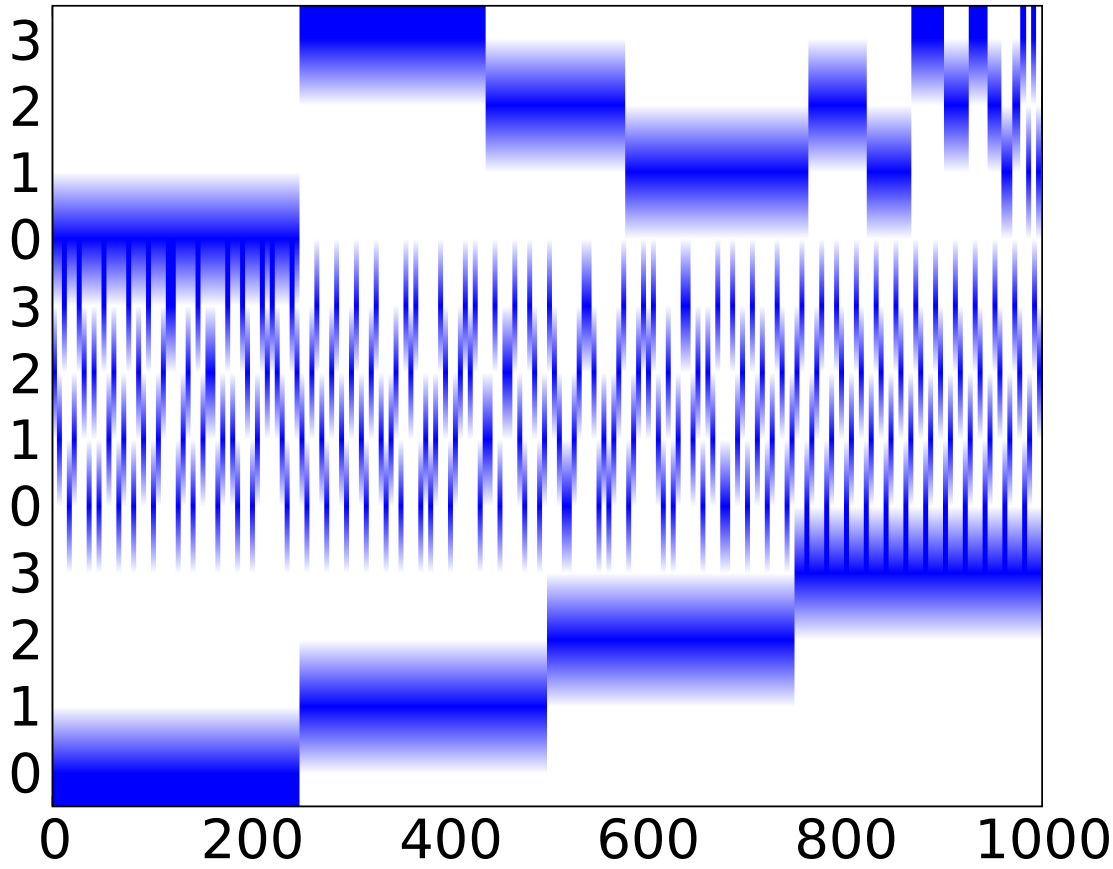


Figure: Different scheduling for a 1000 iterations loop with 4 threads: guided (top), dynamic (middle), static (bottom)

Loop construct: nowait clause

Where are the implied barriers?

```
void nowait_example(int n, int m, float *a,
  float *b, float *y, float *z) {
#pragma omp parallel
  {
#pragma omp for nowait
    for (int i=1; i<n; i++)
      b[i] = (a[i] + a[i-1]) / 2.0;
#pragma omp for nowait
    for (int i=0; i<m; i++)
      y[i] = sqrt(z[i]);
  }
}
```

Loop construct: nowait clause

Is the following snippet semantically correct?

```
void nowait_example2(int n, float *a,
  float *b, float *c, float *y) {
#pragma omp parallel
  {
#pragma omp for schedule(static) nowait
    for (int i=0; i<n; i++)
      c[i] = (a[i] + b[i]) / 2.0f;
#pragma omp for schedule(static) nowait
    for (int i=1; i<=n; i++)
      y[i] = sqrtf(c[i-1]) + a[i];
  }
}
```

Loop construct: nested loops

Am I allowed to do the following?

```
#pragma omp parallel
{
#pragma omp for
  for(int ii = 0; ii < n; ii++) {
#pragma omp for
    for(int jj = 0; jj < m; jj ++) {
      A[ii][jj] = ii*m + jj;
    }
  }
}
```

Loop construct: collapse clause

The right way to collapse nested loops

```
#pragma omp parallel
{
#pragma omp for collapse(2)
  for(int ii = 0; ii < n; ii++) {
    for(int jj = 0; jj < m; jj ++) {
      A[ii][jj] = ii*m + jj;
    }
  }
}
```

1. The collapsed loops must be perfectly nested

Sections construct: syntax

C/C++

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

Sections construct: syntax

Fortran

```
!$omp sections [clause[[,] clause]...]
!$omp section
  structured-block
!$omp section
  structured-block
...
!$omp end do [nowait]
```

Sections construct: some facts

1. `sections` is a non-iterative worksharing construct
   - it contains a set of `structured-blocks`
   - each one is executed once by one of the threads
2. Scheduling of the sections is implementation defined
3. There is an implied barrier at the end of the construct

Single construct: syntax

C/C++

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

Fortran

```
!$omp single [clause[[,] clause] ... ]
  structured-block
[!$omp end single [nowait] ]
```

Single construct: some facts

1. The associated structured block is executed by only one thread
2. The other threads wait at an implicit barrier
3. The method of choosing a thread is implementation defined

Workshare construct: syntax

Fortran

```
!$omp workshare
  structured-block
!$omp end workshare [nowait]
```

Divides the following into shared units of work:

1. array assignments
2. `FORALL` statements or constructs
3. `WHERE` statements or constructs

Master construct: syntax

C/C++

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

Fortran

```
!$omp master
  structured-block
!$omp end master
```

Master construct: some facts

1. The `master` construct specifies a structured block:
   - that is executed by the master thread
   - with no implied barrier on entry or exit
2. Used mainly in:
   - hybrid `MPI-OpenMP` programs
   - progress/debug logging

Critical construct: syntax

C/C++

```
#pragma omp critical [name]
  structured-block
```

Fortran

```
!$omp critical [name]
  structured-block
!$omp end critical [name]
```

Critical contruct: some facts

1. The `critical` construct restricts the execution:
   - to a single thread at a time (wait on entry)
   - disregarding team information
2. An optional name may be used to identify a region
3. All `critical` without a name share the same unspecified tag
4. In `Fortran` the names of `critical` constructs:
   - are global entities of the program
   - may conflict with other names (and trigger undefined behavior)

Critical construct: example

Named critical regions

```
#pragma omp parallel
{
#pragma omp critical(long_critical_name)
  doSomeCriticalWork_1();
#pragma omp critical
  doSomeCriticalWork_2();
#pragma omp critical
  doSomeCriticalWork_3();
}
```

Barrier construct: syntax

C/C++

```
#pragma omp barrier
```

Fortran

```
!$omp barrier
```

The `barrier` construct specifies an explicit barrier at the point at which the construct appears

Barrier construct: example

Waiting for the master to come

```
int counter = 0;
#pragma omp parallel
{
#pragma omp master
  counter = 1;
#pragma omp barrier
  printf("%d\n", counter);
}
```

Atomic construct: syntax

C/C++

```
#pragma omp atomic \
  [read | write | update | capture]
  expression-stmt

#pragma omp atomic capture
  structured-block
```

Atomic construct: syntax

Fortran

```
!$omp atomic read
  capture-statement
[!$omp end atomic]

!$omp atomic write
  write-statement
[!$omp end atomic]
```

Atomic construct: syntax

Fortran

```
!$omp atomic [update]
  update-statement
[!$omp end atomic]

!$omp atomic capture
  update-statement
  capture-statement
!$omp end atomic
```

Atomic construct: some facts

1. The `atomic` construct:
   - ensures a specific storage location to be updated atomically
   - does not expose it to multiple, simultaneous writing threads
2. The binding thread set for an atomic region is all threads
3. The `atomic` construct with the clause:

`read`	forces an atomic read regardless of the machine word size
`write`	forces an atomic write regardless of the machine word size
`update`	forces an atomic update (default)
`capture`	same as an update, but captures original or final value

4. Accesses to the same location must have compatible types

Data-sharing attributes: C/C++

1. The following are always private:
   - variables with automatic storage duration
   - loop iteration variable in the loop construct
2. The following are always shared:
   - objects with dynamic storage duration
   - variables with static storage duration
3. Arguments passed by reference inherit the attributes

Data-sharing attributes: Fortran

1. The following are always private:
   - variables with automatic storage duration
   - loop iteration variable in the loop construct
2. The following are always shared:
   - assumed size arrays
   - variables with save attribute
   - variables belonging to common blocks or in modules
3. Arguments passed by reference inherit the attributes

Data-sharing clauses: syntax

C/C++

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

Fortran

```
!$omp directive-name [clause[[,]clause]
...
!$omp end directive-name [clause]
```

Default/shared/private clauses

1. The clause `default`:
   - is valid on `parallel`
   - accepts `shared` or `none` in `C/C++` and `Fortran`
   - accepts `private` and `firstprivate` in `Fortran`
   - `default(none)` requires each variable to be listed in a clause
2. The clause `shared(list)`:
   - is valid on `parallel`
   - declares one or more list items to be shared
3. The clause `private(list)`:
   - is valid on `parallel`, `for`, `sections`, `single`
   - declares one or more list items to be private
   - allocates a new item of the same type with undefined value

Default/shared/private clauses

Example

```
int q,w;
#pragma omp parallel private(q) shared(w)
{
  q = 0;
#pragma omp single
  w = 0;
#pragma omp critical(stdout_critical)
  printf("%d %d\n", q, w);
}
```

Firstprivate clause

Example

```
int q = 3, w;
#pragma omp parallel firstprivate(q) shared(w)
{
#pragma omp single
  w = 0;
#pragma omp critical(stdout_critical)
  printf("%d %d\n", q, w);
}
```

Same as `private`, but initializes items

Lastprivate clause

Example

```
#pragma omp parallel
{
#pragma omp for lastprivate(i)
  for(i = 0; i < (n1); ++i)
    a[i] = b[i] + b[i + 1];
}
a[i] = b[i];
```

1. valid on `for`, `sections`
2. the value of each new list item is the sequentially last value

Reduction clause: some facts

1. The `reduction` clause:
   - is valid on `parallel`, `loop` and `sections` constructs
   - specifies an operator and one or more list item
2. A list item that appears in a `reduction` clause must be shared
3. For each item in the list:
   - a private copy is created and initialized appropriately
   - at the end of the region the original item is updated
4. Aggregate types may not appear in a reduction clause
5. Items must not be const-qualified

Reduction clause: example

Sum over many iterations

```
int a = 5;
#pragma omp parallel
{
#pragma omp for reduction(+:a)
  for(int i = 0; i < 10; ++i)
    ++a;
}
printf("%d\n", a);
```

Reduction clause: example

Fortran features?

```
PROGRAM REDUCTION_WRONG
  MAX = HUGE(0)
  M = 0
!$OMP PARALLEL DO REDUCTION(MAX: M)
  DO I = 1, 100
    CALL SUB(M,I)
  END DO
END PROGRAM REDUCTION_WRONG
```

Copyprivate clause

C/C++

```
#pragma omp single copyprivate(tmp)
{
  tmp = (float *) malloc(sizeof(float));
} /* copies the pointer only */
```

1. Valid only on `single`
2. Broadcasts the value of a private variable

Outline

1 Introduction

2 Directives

3 Runtime library routines and environment variables

Runtime library routines

Most used functions

```
int omp_get_num_threads(void);// # of threads
int omp_get_thread_num(void);// thread id
double omp_get_wtime(void);// get wall-time
```

1. Prototypes for `C/C++` runtime are provided in `omp.h`
2. Interface declarations for `Fortran` are provided as:
   - a `Fortran` include file named `omp_lib.h`
   - a `Fortran 90` module named `omp_lib`

Environment variables

OMP_NUM_THREAD	sets the number of threads for parallel regions
OMP_STACKSIZE	specifies the size of the stack for threads
OMP_SCHEDULE	controls schedule type and chunk size of `runtime`
OMP_PROC_BIND	controls whether threads are bound to processors
OMP_NESTED	enables or disables nested parallelism

OpenMP: just take a shot at it!