www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

OpenMP Worksharings

Distributing the work among threads

Xavier Teruel & Xavier Martorell

Parallel Programming with OpenMP

Worksharing introduction

- Worksharing constructs divide the execution of a code region among the threads of a team
  - threads cooperate to do some work (i.e. to share some work)
  - better way to split work than using thread-ids
  - lower overhead than using tasks → less flexible
- In OpenMP, there are four worksharing constructs:
  - single construct
  - sections construct
  - loop construct
  - workshare construct (only Fortran)
- Restriction: worksharings cannot be nested

The single construct

- Serializing (1-thread) a portion of the parallel region
  - always attached to a structured block

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

- Where clause:
  - private(list) → explained
  - firstprivate(list) → explained
  - nowait
  - copyprivate(list)
- Only one thread of the team executes the structured block
- Very useful in I/O operations

- Single construct example

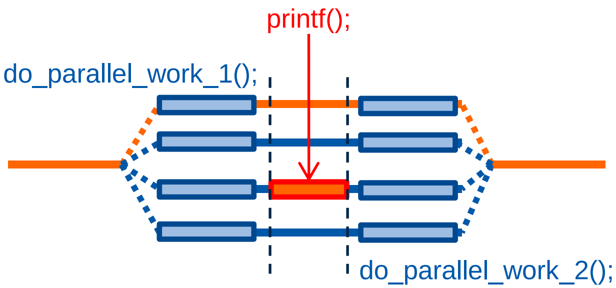
```
#include <stdio.h>

int main ( void )
{
  #pragma omp parallel
  {
    do_parallel_work_1();
    #pragma omp single
    {
      printf ("Hello world!\n" ) ;
    }
    do_parallel_work_2();
  }
}
```

This program writes just one "Hello world!"

Implicit barrier (single)

- A implicit barrier at the end of the construct

```
#pragma omp parallel
{
  do_parallel_work_1();
  #pragma omp single
  {
    printf ("Hello world!\n" ) ;
  }
  do_parallel_work_2();
}
```

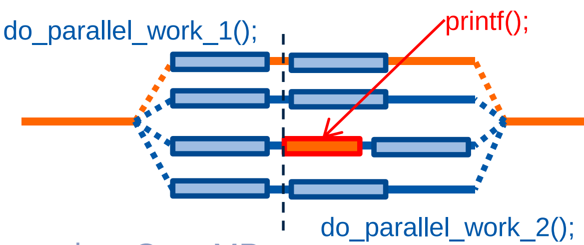


- The nowait clause

```
#pragma omp single nowait
{structured-block}
```

  – eliminates the barrier at the end of the construct

```
...
#pragma omp single nowait
printf ("Hello world!\n" ) ;
...
```



Copying variables from/to the construct (broadcasting)

The copyprivate clause

```
#pragma omp single copyprivate(list)
{structured-block}
```

Copyprivate description

- support the broadcast of data values to other threads in the team
- apply only to private, firstprivate or threadprivate variables
- occurs after the execution of the structured block…
- … but before of the threads have left the barrier (at the end of the construct)

Copyprivate example (input data)

```
#include <stdio.h>

void main (void)
{
  float x, y;
  #pragma omp parallel private(x,y)
  {
    ...
    #pragma omp single copyprivate(x,y)
    {
      scanf("%f %f", &x, &y);
    }
    ...
  }
}
```

At this point variables 'x' and 'y' have been broadcasted

Single construct vs master construct

- In both cases the structured block is executed by just one thread

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

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

- The single construct has more overhead (additional synchronization)
  - which thread has captured the token
  - and the implicit barrier at the end
- … but also is more flexible: any thread may execute the block
- The master construct has less overhead
  - it is just a test (if *thread-id* == 0)
  - it has no implicit barrier at the end
- … but also is more restrictive: only master thread may execute the block

- Rule of thumb: if all threads reach the structured block at the same time use master, otherwise use single

The sections construct

- Set of structured blocks distributed among threads

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

- Where clause:
  - private(list) → already explained in previous constructs
  - firstprivate(list) → already explained in previous constructs
  - lastprivate(list)
  - reduction(operator: variable-list) → already explained in previous constructs
  - nowait → already explained in previous constructs

The sections construct: description (1)

- Building the syntaxis of the sections construct
  - each (selected) structured block is preceded by a section directive
  - only in the first structured block the section directive is optional
  - any section directive must be lexically enclosed in a sections construct
- Section construct example

```
#include "synthetic.h“

void main (void)
{
  #pragma omp parallel
  #pragma omp sections
  {
    #pragma omp section
    synthetic_phase1();
    #pragma omp section
    synthetic_phase2();
    #pragma omp section
    synthetic_phase3();
  }
}
```

Only in the first structured block the section directive is optional

```
#include "synthetic.h“

void synthetic_phase2()
{
  #pragma omp section
  synthetic_phase2_1();
}
```

✗

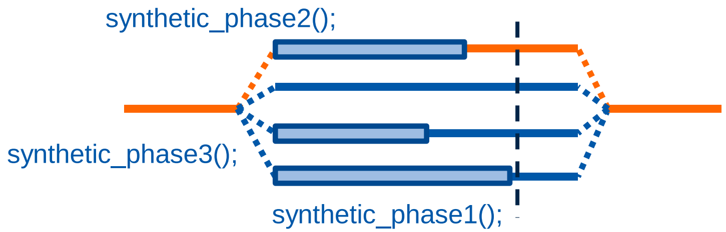
The sections construct: description (2)

- Executing the sections construct
  - assignment blocks/threads is implementation defined
  - if no 'nowait' clause is present there is an implicit barrier at the end
- It can be combined with the parallel construct

```
#pragma omp parallel sections [clause[[,] clause]...]
{structured-blocks: sections}
```

- Using the "parallel sections" combined construct

```
void main (void)
{
  #pragma omp parallel sections
  {
    synthetic_phase1();
    #pragma omp section
    synthetic_phase2();
    #pragma omp section
    synthetic_phase3();
  }
}
```



Privatizing variables inside the construct (lastprivate)

- The variable inside the construct is a new variable
  - the new variables have the same type than original variable
  - in any worksharing construct it means all threads have a different variable
  - they can be accessed without any kind of synchronization
- Already discussed privatization clauses
  - private variables have undefined value when starting the block
  - firstprivate variables are initialized to the value of the original one
- The lastprivate clause

```
#pragma omp sections lastprivate(list)
{structured-blocks: sections}
```

  - lastprivate variables (by default) have undefined value when starting the block
  - the value of the variable in the lexically last section of the set of sections is copied back to the original variable
  - a variable can be both firstprivate and lastprivate

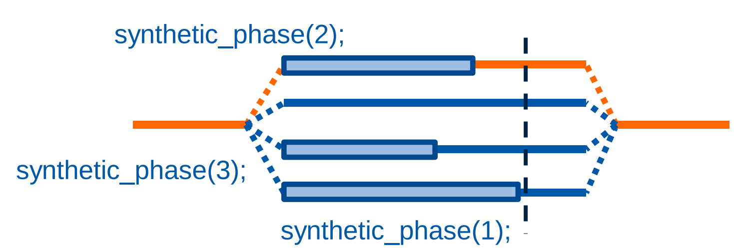
A lastprivate example (with sections construct)

- Recovering the sequential consistency with the lastprivate clause

```c
#include <stdio.h>
void main (void) {
  int v = 0;
  #pragma omp parallel sections lastprivate(v)
  {
    #pragma omp section
    {
       v = 1;
       synthetic_phase( v );
    }
    #pragma omp section
    {
       v = 2;
       synthetic_phase( v );
    }
    #pragma omp section
    {
       v = 3;
       synthetic_phase( v );
    }
  }
  printf("v = %d\n", v);
}
```

The lexically last section determines the value of the original variable

```c
#include "synthetic.h"

void synthetic_phase( int  s ) {
  switch case(s)
  {
    case 1:
      matrix_multiply();
      break;
    . . .
    default:
      exit(NOT_IMPLEMENTED);
  }
}
```

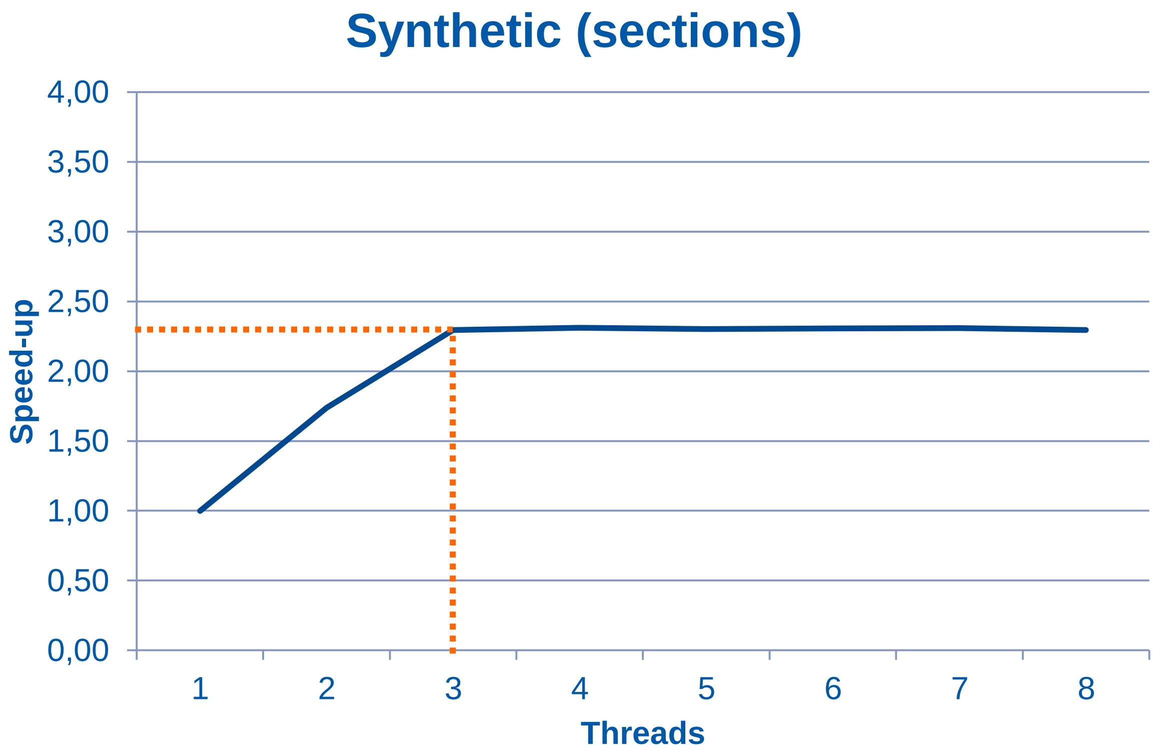


Some performance results (synthetic)

- Computing speed-up for the synthetic benchmark (using sections)

Time Results

Threads	Total Time	Speed-up
1	4,454202	1,00
2	2,562986	1,74
3	1,940174	2,30
4	1,927576	2,31
5	1,934126	2,30
6	1,929955	2,31
7	1,927792	2,31
8	1,941034	2,29
S	4,452954	1,00



$$Speedup = \frac{T_{seq}}{T_p}$$

The optimal amount of parallelism

- Parallel decomposition (choosing the entity's granularity)
  - Where entity may be a (section) structured block, or a (loop) chunk, or a task
  - Parallelization may occur at different application levels
    - Higher levels → coarse grain granularity
      - Small synchronization overhead
      - Load imbalance (including lack of parallelism)
    - Deeper levels → fine grain granularity
      - Greater potential for parallelism (and hence speed-up)
      - More synchronization overhead
  - The optimal decission is a trade off (but sometimes is difficult to find)

The loop construct

- Distributing a loop among threads
  - always attached to a for loop (do in Fortran)

```
#pragma omp for [clause[[,] clause]...]
{structured-block: loop}
```

- Where clause:
  - private(list) → already explained in previous constructs
  - firstprivate(list) → already explained in previous constructs
  - lastprivate(list) → already explained, but…
  - reduction(operator: list) → already explained, but…
  - schedule(schedule-kind)
  - nowait → already explained in previous constructs
  - collapse(n)
  - ordered

The loop construct: description (1)

- The iterations of the loop(s) associated to the construct are divided among the threads of the team
- Parallel loop requierements
  - loop iterations must be independent (user's responsibility)
  - loops must follow a form that allows to compute the number of iterations

```
#pragma omp for [clause[[,] clause]...]
for ( init_expr; test_expr; inc_expr )
```

  - valid data types for induction variables are: integer types, pointers and random access iterators (in C++)

The loop construct: description (2)

- It can be combined with the parallel construct

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

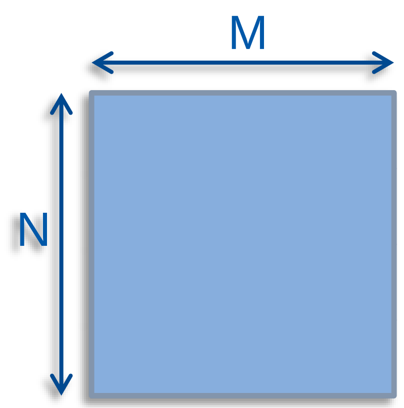
- Matrix initialization (using the loop construct)

```
void foo ( int *m, int N, int M)
{
  int i, j ;
#pragma omp parallel for private( j )
  for ( i = 0; i < N; i ++ )
    for ( j = 0; j < M; j ++ )
      m[ i * N + j ] = 0;
}
```

New created threads cooperate to execute all the iterations of the loop

The i variable is automatically privatized

The j variable must be manually privatized



… but other distributions are also possible

Loop construct and the lastprivate clause

The lastprivate clause

```
#pragma omp for lastprivate(list)
{structured-block: loop}
```

- lastprivate variables (by default) have undefined value when starting the block
- the value of the variable in the last logical iteration of the loop is copied back to the original variable
- a variable can be both firstprivate and lastprivate

A lastprivate example

```
void main(void)
{
  int i;

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

  a[ i ] = b[ i ]; /* i == n-1 here */
}
```

The logical last iteration determines the value of the original variable

Loop construct and the reduction clause

- All threads accumulate some values into a single variable

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

- Reduction clause example (loop construct)

```
int vector_sum ( int n , int v [ n ] )
{
  int i , sum = 0;
#pragma omp parallel for reduction ( + : sum)
  {
    for ( i = 0; i < n ; i ++ )
      sum += v [ i ] ;
  }
  return sum;
}
```

Private copies initialized to the identity

Shared variable updated with all the partial results

  - 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

- Using critical is not good enough (besides being error prone)

Loop 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 → shared / automatic → private)*
  - the loop iteration variable(s) in the associated for-loop(s) of a for, parallel for, distribute or taskloop constructs is (are) private
  - *the loop iteration variable in the associated (and unique) for-loop of a simd construct is linear*
  - *the loop iteration variables in the associated (multiple) for-loops of a simd construct are lastprivate*
- *Explicit data-sharing clauses (shared, private, firstprivate,…)*
  - *If default clause present, what the clause says (none is very usefull!!!)*
- Implicit data-sharing rules, depends on the construct
  - For the loop region the default data sharing attribute is shared

The schedule clause

- The schedule clause determines which iterations are executed by each of the threads in the team

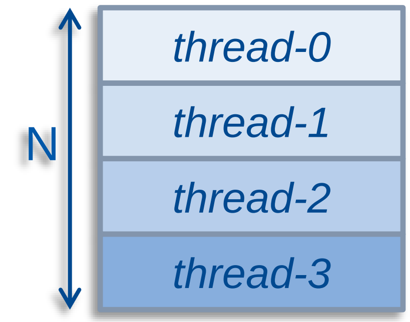
```
#pragma omp for schedule(kind[,chunk-size])
{structured-block: loop}
```

  - If no schedule clause is present then is implementation defined
- There are several possible options as schedule kind
  - static[,chunk-size]
  - dynamic[,chunk-size]
  - guided[,chunk-size]
  - auto
  - runtime

The loop's schedule clause: static

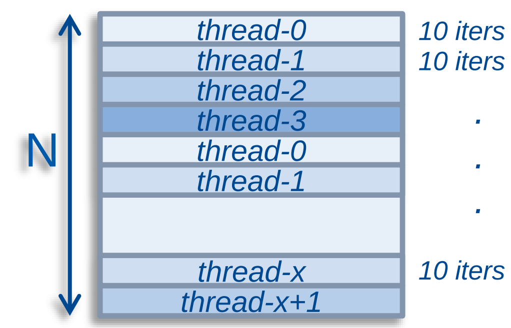
- The static schedule (with no chunk-size parameter)
  - the iteration space is broken in chunks of approximately the same size
  - then these chunks are assigned to the threads in a Round-Robin fashion

```
...
#pragma omp parallel for private( j ) schedule(static)
  for ( i = 0; i < N; i ++ )
    for ( j = 0; j < M; j ++ )
      m[ i * N + j ] = 0;
...
```



- The static schedule (with chunk-size parameter) → interleaved
  - the iteration space is broken in chunks of size N
  - these chunks are assigned to the threads in a Round-Robin fashion

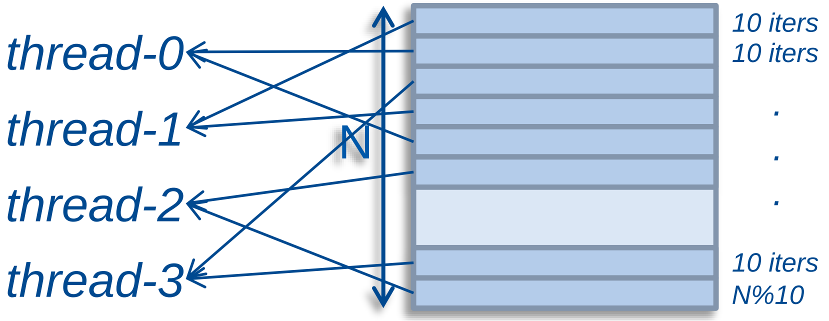
```
...
#pragma omp parallel for private( j ) schedule(static,10)
  for ( i = 0; i < N; i ++ )
    for ( j = 0; j < M; j ++ )
      m[ i * N + j ] = 0;
...
```



The loop's schedule clause: dynamic & guided

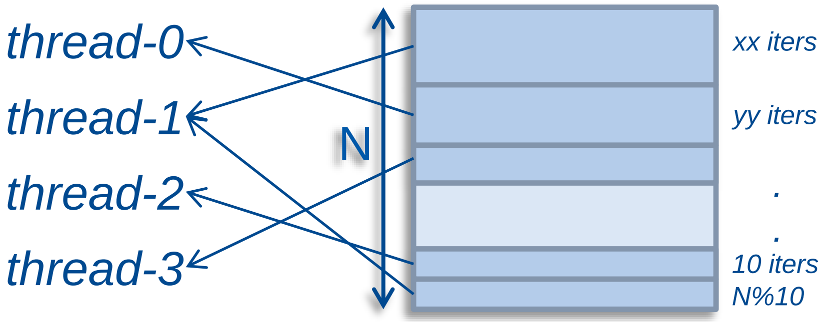
- The dynamic schedule
  - if no chunk-size is specified, default is 1.
  - threads dynamically grab iterations until all iterations have been executed

```
...
#pragma omp parallel for private( j ) schedule(dynamic, 10)
  for ( i = 0; i < N; i ++ )
    for ( j = 0; j < M; j ++ )
      m[ i * N + j ] = 0;
...
```



- The guided schedule (variant of dynamic)
  - if no chunk-size is specified, default is 1
  - chunks decreases in size as threads grab iterations (at least chunk-size)

```
...
#pragma omp parallel for private( j ) schedule(guided, 10)
  for ( i = 0; i < N; i ++ )
    for ( j = 0; j < M; j ++ )
      m[ i * N + j ] = 0;
...
```



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Loop's schedulers: static vs dynamic (and guided)

- Characteristics of static schedules
  - low overhead
  - good locality (usually)
  - can have load imbalance problems

- Dynamic (and guided) schedulers
  - higher overhead
  - not very good locality (usually)
  - can solve imbalance problems

- Which scheduler should work better with a specific loop
  - if all threads reach the loop region at the same time
  - if all the iterations have the same weight (work)
  - if consequtive loops using the same data (e.g. matrix)

  } static

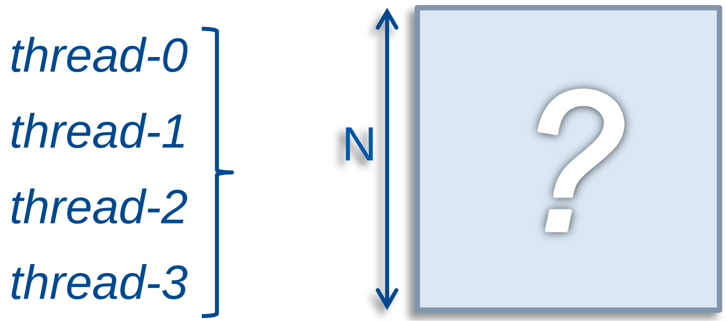
  - if threads may reach the loop at different times
  - if not all the iterations have the same weight (work)

  } dynamic (guided)

The schedule clause: auto & runtime

- The auto schedule (if you want to experiment)
  - in this case, the implementation is allowed to do whatever it wishes
  - do not expect much of it as of now

```
...
#pragma omp parallel for private( j ) schedule(auto)
  for ( i = 0; i < N; i ++ )
    for ( j = 0; j < M; j ++ )
      m[ i * N + j ] = 0;
...
```



- The runtime schedule (delayed until run-time)
  - using the OMP_SCHEDULE environment variable
  - using the omp_set_schedule() API service call

```
...
#pragma omp parallel for private( j ) schedule(runtime)
  for ( i = 0; i < N; i ++ )
    for ( j = 0; j < M; j ++ )
      m[ i * N + j ] = 0;
...
```

```
$ export OMP_SCHEDULE=static,1024
$ ./myMatrixMultiply
Computing matrix multiplication…
```

Avoiding the implicit barrier (loop)

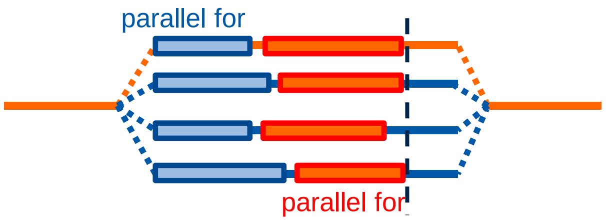
- The nowait clause: eliminates the barrier at the end of the loop

```
#pragma omp for nowait
{structured-block}
```

- This allows to overlap the execution of non-dependent loops

```
#define N 1000
void main (void) {
  int i, a[N], b[N];

  #pragma omp parallel
  {
    #pragma omp for nowait
    for ( i = 0; i < N ; i ++ )
      a [ i ] = 0;

    #pragma omp for
    for ( i = 0; i < N ; i ++ )
      b [ i ] = 0;
  }
}
```

  - independant iterations (in between loops) → we can overlap them
  - if same iteration space → a better solution would be to (manually) fuse the loops



Avoiding the implicit barrier (loop)

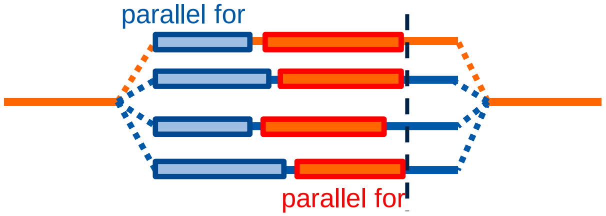
- The nowait clause: eliminates the barrier at the end of the loop

```
#pragma omp single nowait
{structured-block}
```

- But also overlap the execution of "some" dependant loops

```
#define N 1000
void main (void) {
  int i, a[N], b[N];

  #pragma omp parallel
  {
    #pragma omp for schedule(static) nowait
    for ( i = 0; i < N ; i ++ )
      a [ i ] = 0;

    #pragma omp schedule(static) for
    for ( i = 0; i < N ; i ++ )
      a [ i ] = a [ i ] + foo ( i );
  }
}
```

  - static scheduler, same iteration space, and dependant (on index) iterations (in between loops) → we can overlap them
  - a better solution would be to (manually) fuse the loops



Avoiding the implicit barrier (loop)

- The nowait clause: eliminates the barrier at the end of the loop

```
#pragma omp single nowait
{structured-block}
```

- But also overlap the execution of “some” dependant loops

```
#define N 1000
void main (void) {
  int i, a[N], b[N];

  #pragma omp parallel
  {
    #pragma omp for schedule(dynamic) nowait
    for ( i = 0; i < N ; i ++ )
      a [ i ] = 0;

    #pragma omp for
    for ( i = 0; i < N ; i ++ )
      a [ i ] = a [ i ] + foo ( i );;
  }
}
```

- no static scheduler: same iteration space, and dependant (on index) iterations (in between loops) → NO
  - a better solution would be to (manually) fuse the loops
- not the same iteration space: static scheduler and dependant (on index) iterations (in between loops) → NO
- dependence (arbitrary in any index): same iteration space and static scheduler → NO

The collapse clause

- Allows to distribute work from a set of n-nested loops
  - loops must be perfectly nested (no instruction in between)
  - the nest must traverse a rectangular iteration space
  - combines both iteration spaces to create a single one
- Using the collapse clause over two loops

```
#define N 1000
#define M 4000

void main (void) {
  int i, j;
  #pragma omp parallel
  {
    #pragma omp for collapse(2)
    for ( i = 0; i < N; i ++ )
      for ( j = 0; j < M; j ++ )
        foo ( i , j ) ;
  }
}
```

- useful when first loop (or both) have only a few iterations (e.g. N = 64)
- increase the amount of created parallelim

```
#pragma omp for
for ( idx = 0; idx < (N * M); idx ++ )
{
    foo ( fi(idx) , fj(idx) ) ;
}
```

Synchronizing the execution

- Threads need to impose some ordering in the sequence of their actions
  - execute in a logical order certain regions
  - mutual exclusion in the execution of a given region
  - wait in a location until all other threads have reach the same location
  - wait until a given condition is acomplished
- OpenMP provides different synchronization mechanisms
  - master construct→ already explained in previous sessions
  - critical construct → already explained in previous sessions
  - barrier directive
  - atomic construct
  - taskwait directive → will be explained in following sessions (tasking)
  - taskgroup construct → will be explained in folowing session (tasking)
  - depend clause → will be explained in following sessions (tasking)

The barrier directive

- Threads cannot proceed past a barrier point until all threads reach the barrier and all previously generated work is completed

```
#pragma omp barrier
```

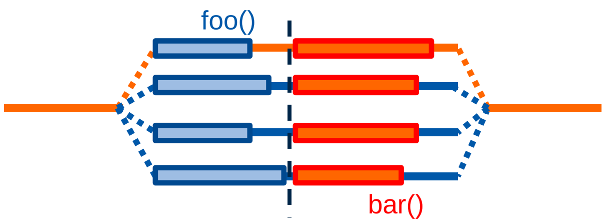
  – Some constructs have an implicit barrier at the end (e.g. the parallel construct)

- Synchronizing threads between two phases in a parallel region

```
#pragma omp parallel
{
  foo ();
  #pragma omp barrier
  bar ();
}
```

Forces all foo()'s too happen before all bar()'s

Implicit barrier



Mutual exclusion for simple read & update operations

- The atomic construct
  - special mechanism of mutual exclusion to “read & update” operations
  - only supports simple read & update expressions
    - e.g., x += 1 → whole expression is protected
    - x = x - foo() → only protects the read & update part, foo() is not protected
- Usually much more efficient than a critical construct…
- … but it is not compatible with it →

```
int x =1;
#pragma omp parallel num_threads( 2 )
{
  #pragma omp atomic
  x++;
}
printf("%d\n", x );
```

Only one thread at a time updates x here

Prints “3”

```
int x =1;
#pragma omp parallel num_threads( 2 )
{
  #pragma omp atomic
  x++;
  . . .
  #pragma omp critical
  x++;
}
printf("%d\n", x );
```

May execute an atomic and a critical block at the same time

Prints “?”

- An additional mechanism to fix data races

Summary: OpenMP worksharings

- OpenMP worksharings: single, section, loop and workshare
  - distribute work among threads withoud using thread-id (neither num-threads)
  - parallel decomposition trade off: coarse and fine granularity
  - control how the work is distribute (loop) using the schedule clause
  - new ways to control the data environment in these news constructs
- Additional synchronization constructs
  - the barrier directive → synchronize threads
  - the atomic directive → other mechanism to fix data races

For further information please visit/contact:

http://www.linkedin.com/in/xteruel
xavier.teruel@bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

THANKS