Parallelism (PAR)

Short tutorial on OpenMP 5.0 tasking

Eduard Ayguadé[1] and Alex Durán[2]

[1]Barcelona Supercomputing Center – Universitat Politècnica de Catalunya
[2]Intel Corporation

Part I

OpenMP Basics

Outline

- OpenMP overview
- OpenMP model
- Creating threads and accessing data
- Some API calls
- The single construct
- Thread synchronization
- Memory consistency

Outline

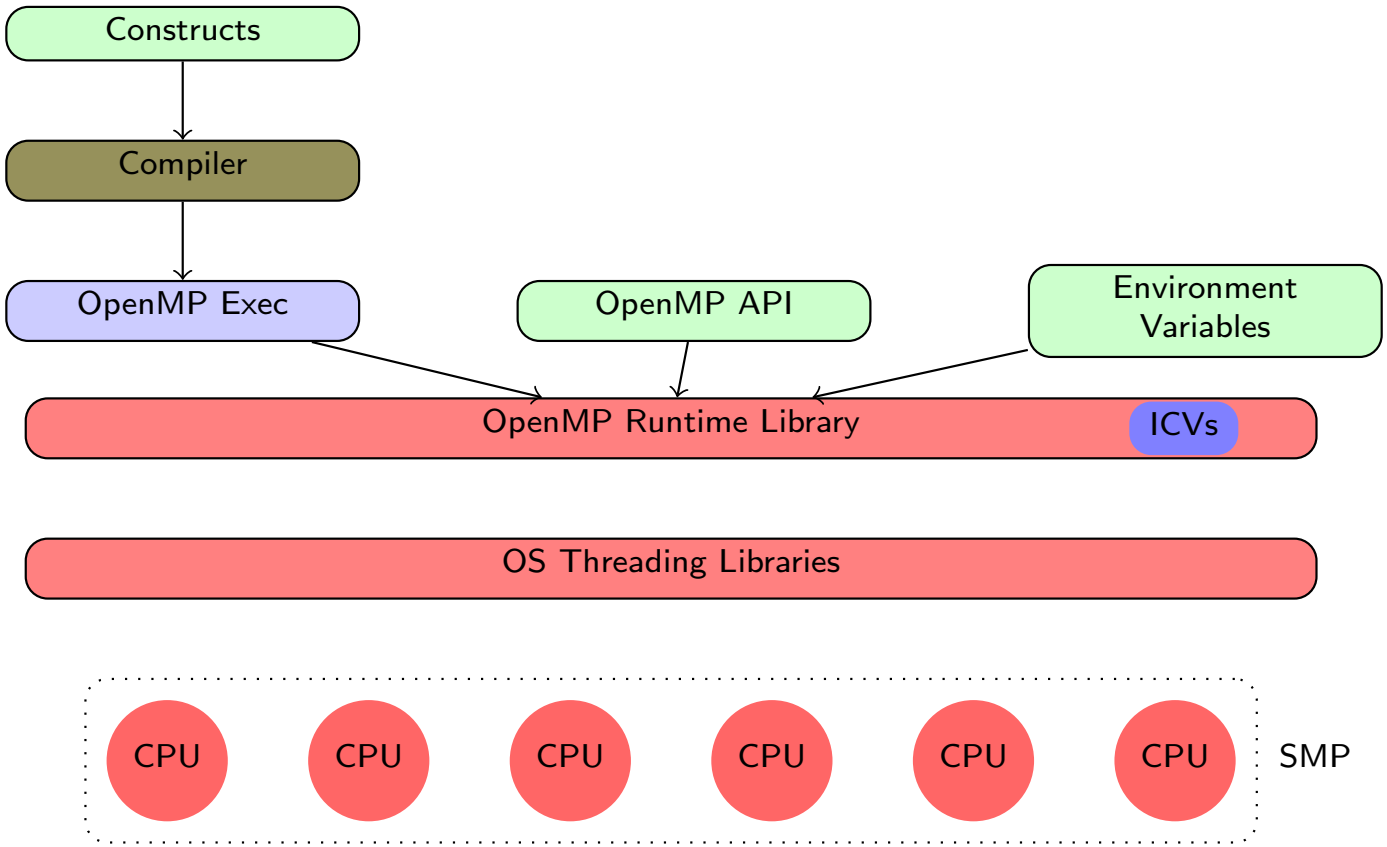
- OpenMP overview
- OpenMP model
- Creating threads and accessing data
- Some API calls
- The single construct
- Thread synchronization
- Memory consistency

What is OpenMP?

- It's an API extension to the C, C++ and Fortran languages to write parallel programs for shared memory machines
  - Current version is 5.0 (November 2018)
    - Releases 5.1 and 5.2 appeared on November 2020 and 2021 respectively
  - Supported by most compiler vendors and implementors
    - Intel, IBM, PGI, GCC, LLVM, ...
- Maintained by the Architecture Review Board (ARB), a consortium of industry and academia
- This mini-tutorial just covers part of the specification, for the complete reference please consult the documentation online

http://www.openmp.org

OpenMP components



OpenMP components

Constructs

These form the major elements of OpenMP programming

- Create threads and tasks
- Share the work amongst threads and accelerators (not covered in this mini-tutorial)
- Synchronize threads and memory, and wait for termination of tasks

Library routines

To control and query the parallel execution environment (internal control variables - ICVs)

Environment variables

The execution environment can also be set before the program execution is started

OpenMP directives syntax

In C/C++

Through a compiler directive:

```
#pragma omp construct [clauses]
```

- OpenMP syntax is ignored if the compiler does not have the appropriate compilation flag activated

Structured block

Most directives apply to:

- A block of one or more statements
- One entry point, one exit point (no branching in or out allowed)

Headers/Macros

C/C++ only

- omp.h contains the API prototypes and data types definitions
- The _OPENMP is defined by OpenMP enabled compiler
  - Allows conditional compilation of OpenMP

```
#ifdef _OPENMP
    printf("Parallel execution with %d threads\n",
                omp_get_num_threads());
#endif
```

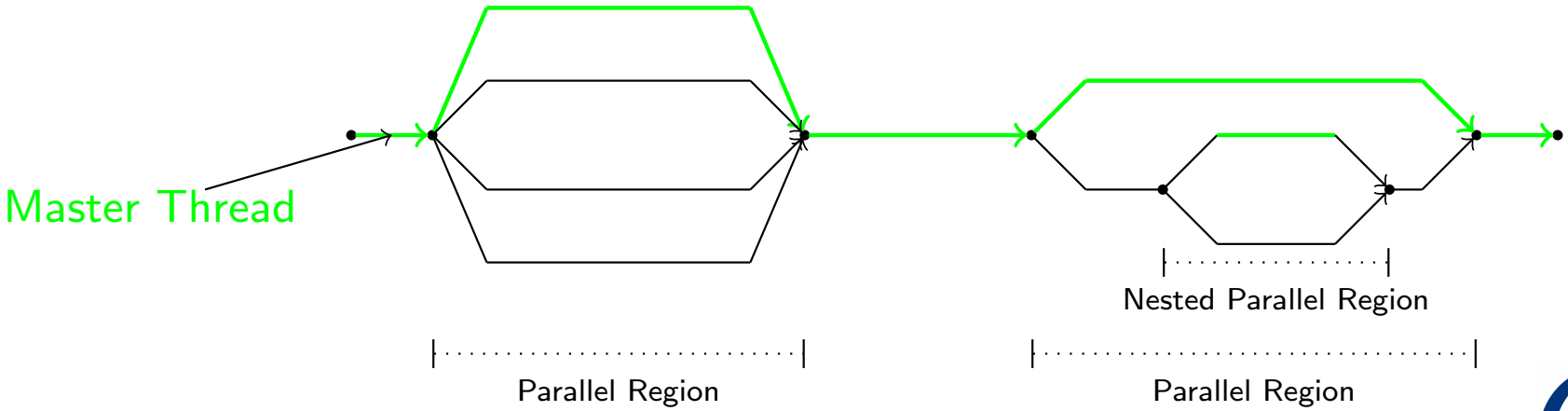
Outline

- OpenMP overview
- OpenMP model
- Creating threads and accessing data
- Some API calls
- The single construct
- Thread synchronization
- Memory consistency

Execution model

Fork-join model

- OpenMP uses a fork-join model
  - The master thread spawns a team of threads that joins at the end of the parallel region
  - Threads in the same team can collaborate to do work



Memory model

- OpenMP defines a relaxed memory model
  - Threads can see different values for the same variable
  - Memory consistency is only guaranteed at specific points
  - Luckily, the default points are usually enough
    - If not ... there is a mechanism to guarantee it! (described at the end of Part I)
- Variables can be shared or private to each thread

Outline

- OpenMP overview
- OpenMP model
- Creating threads and accessing data
- Some API calls
- The single construct
- Thread synchronization
- Memory consistency

The parallel construct

Directive

```
#pragma omp parallel [clauses]
    structured block
```

where some of the `clauses` are:

- `num_threads(`*expression*`)`
- `if(`*expression*`)`
- shared(*var-list*)
- private(*var-list*)
- firstprivate(*var-list*)
- reduction(*operator:var-list*)

Coming shortly!

We'll see it later

The parallel construct

Specifying the number of threads

- The `nthreads-var` ICV is used to determine the number of threads to be used for encountered parallel regions
  - It is a list of positive integer values, its first element specifying the number of processors for the next nesting level
  - When a `parallel` construct is encountered, and the generating task's `nthreads-var` list contains multiple elements, the generated task(s) inherit the value of `nthreads-var` as the list obtained by deletion of the first element
  - If the generating task's `nthreads-var` list contains a single element, the generated task(s) inherit that list as the value of `nthreads-var`

The parallel construct

Specifying the number of threads

- The `nthreads-var` list can be defined from the execution command line by setting the `OMP_NUM_THREADS` environment variable
  - Example: `setenv OMP_NUM_THREADS 4,3,2`
- After that, the `threads-var` list can be modified through:
  - the `omp_set_num_threads` API, which sets the value of the first element of the current list
  - the `num_threads` clause, which causes the implementation to ignore the ICV and use the value of the clause for that region.

The `if` clause

Avoiding parallel regions

- Sometimes we only want to run in parallel under certain conditions
  - E.g., enough input data, not running already in parallel, ...
- The `if` clause allows to specify an *expression*. When evaluates to false the `parallel` construct will only use 1 thread
  - Note that still creates a new team and data environment

The parallel construct

```
void main () {
  #pragma omp parallel
    ...
  omp_set_num_threads(2);
  #pragma omp parallel
    ...
  #pragma omp parallel num_threads(random()%4+1) if(0)
    ...
}
```

How many threads are used in each parallel region above?

Data-sharing attributes

Shared

When a variable is marked as `shared`, the variable inside the construct is the same as the one outside the construct

- In a parallel construct this means all threads see the same variable
  - but not necessarily the same value
- Usually need some kind of synchronization to update them correctly
  - OpenMP has consistency points at synchronizations
- By default, variables are implicitly `shared`

Data-sharing attributes

Private

When a variable is marked as `private`, the variable inside the construct is a new variable of the same type with an undefined value

- In a parallel construct this means all threads have a different variable
- Can be accessed without any kind of synchronization

Data-sharing attributes

Firstprivate

When a variable is marked as `firstprivate`, the variable inside the construct is a new variable of the same type but it is initialized to the original variable value

- In a parallel construct this means all threads have a different variable with the same initial value
- Can be accessed without any kind of synchronization

Data-sharing attributes

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

What does appear on the screen if `XXXXXX` is `shared(x)`, `private(x)` or `firstprivate(x)`?

Example: computation of PI

```
static long num_steps = 100000;
double step;
void main ()
{
    int i;
    double x, pi, sum = 0.0;

    step = 1.0/(double) num_steps;

    for (i=1;i<= num_steps; i++){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

Example: computation of PI (not equivalent to sequential!)

```
static long num_steps = 100000;
double step;
#include <omp.h>
#define NUM_THREADS 2
void main ()
{
    int i, id;
    double x, pi, sum=0.0;

    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel private(x, i)
    for (i=1;i<= num_steps; i++){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }

    pi = sum * step;
}
```

Outline

- OpenMP overview
- OpenMP model
- Creating threads and accessing data
- Some API calls
- The single construct
- Thread synchronization
- Memory consistency

Some useful routines

int `omp_get_num_threads`()	Returns the number of threads in the current team. 1 if outside a parallel region
int `omp_get_thread_num`()	Returns the id of the thread in the current team. id between 0 and `omp_get_num_threads()-1`
void `omp_set_num_threads`()	Sets the number of threads to be used in parallel regions at the next nesting level
int `omp_get_max_threads`()	Returns the number of threads that could be used in parallel regions at the next nesting level
double `omp_get_wtime`()	Returns the number of seconds since an arbitrary point in the past

Example: computation of PI (data race!)

```
static long num_steps = 100000;
double step;
#include <omp.h>
#define NUM_THREADS 2
void main ()
{
    int i, id;
    double x, pi, sum=0.0;

    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel private(x, i, id)
    {
        id = omp_get_thread_num();
        for (i=id+1; i<=num_steps; i=i+NUM_THREADS) {
            x = (i-0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
        }
    }
    pi = sum * step;
}
```

Example: computation of PI (measuring execution time)

```
static long num_steps = 100000;
double step;
#include <omp.h>
#define NUM_THREADS 2
void main ()
{
    int i, id;
    double x, pi, sum=0.0;
    double TimeStart, TimeEnd;

    TimeStart = omp_get_wtime();
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel private(x, i, id)
    {
    ...
    }
    pi = sum * step;
    TimeEnd = omp_get_wtime();
    printf("Wall clock time = %.20f\n", TimeEnd-TimeStart);
}
```

Outline

- OpenMP overview
- OpenMP model
- Creating threads and accessing data
- Some API calls
- The single construct
- Thread synchronization
- Memory consistency

Only one thread in the team doing the work

The single construct

```
#pragma omp single [clauses]
    structured block
```

- where clauses can be:
  - `private`
  - `firstprivate`
  - `nowait`
- Only one thread of the team executes the structured block
- There is an implicit `barrier` at the end

The nowait clause

When `single` has a `nowait` clause then the implicit `barrier` at the end of the `single` is removed.

- This allows to overlap the execution of non-dependent work inside single with what continues after it.

```
#pragma omp single nowait
for ( i = 0; i < n ; i++ )
   v[i] = 0;
#pragma omp single
for ( i = 0; i < n ; i++ )
   a[i] = 0;
```

First and second single regions are independent so we can overlap them

Example: computation of PI with `single`

```
...
void main ()
{
    ...
    #pragma omp parallel private(x, i) firstprivate(sum)
    {
        #pragma omp single nowait
        {
            for (i=1; i<=num_steps; i++) {
                x = (i-0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
            }
            pi = sum * step;
        }
        if (sum==0.0)
            printf("Life is good when there is nothing to do...\n");
}
```

Outline

- OpenMP overview
- OpenMP model
- Creating threads and accessing data
- Some API calls
- The single construct
- Thread synchronization
- Memory consistency

Why synchronization?

OpenMP is a shared memory model

- Threads communicate by sharing variables
- Unintended sharing of data causes race conditions (i.e. the execution outcome may change as threads are scheduled differently)
- Threads need to synchronize to impose some ordering in their sequence of actions

Some OpenMP synchronization mechanisms:

- `barrier`
- `critical`
- `atomic`
- Use of locks through API

Thread Barrier

The barrier construct

#pragma `omp barrier`

- Threads cannot proceed past a barrier point until all threads reach the barrier AND all previously generated work is completed
- Some constructs have an implicit `barrier` at the end
  - E.g., the `parallel` construct

Barrier

Example

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

Forces all foo occurrences too happen before all bar occurrences

Implicit barrier at the end of the `parallel` region

Exclusive access: critical construct

```
#pragma omp critical [(name)]
    structured block
```

- Provides a region of mutual exclusion where only one thread can be working at any given time
- By default all critical regions are the same
- Multiple mutual exclusion regions by providing them with a name
  - Only those with the same name synchronize

Example: computation of PI

```
...
void main ()
{
    int i, id;
    double x, pi, sum=0.0;

    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel private(x, i, id)
    {
        id = omp_get_thread_num();
        for (i=id+1; i<=num_steps; i=i+NUM_THREADS) {
            x = (i-0.5)*step;
            #pragma omp critical
            sum = sum + 4.0/(1.0+x*x);
        }
    }
    pi = sum * step;
}
```

Critical construct

```
int x=1,y=0;
#pragma omp parallel num_threads(4)
{
#pragma omp critical (x)
    x++;
#pragma omp critical (y)
    y++;
}
```

Different names: One thread can update x while another updates y

Exclusive access: atomic construct

```
#pragma omp atomic [update | read | write]
        expression
```

- Ensures that a specific storage location is accessed atomically, avoiding the possibility of multiple, simultaneous reading and writing threads
  - Atomic updates: `x += 1`, `x = x - foo()`, `x[index[i]]++`
  - Atomic reads: `value = *p`
  - Atomic writes: `*p = value`
- Only protects the read/operation/write
- Usually more efficient than a `critical` construct
- Other clauses and forms for `atomic` are allowed in the specification

First example: computation of PI

```
...
void main ()
{
    int i, id;
    double x, pi, sum=0.0;

    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel private(x, i, id)
    {
        id = omp_get_thread_num();
        for (i=id+1; i<=num_steps; i=i+NUM_THREADS) {
            x = (i-0.5)*step;
            #pragma omp atomic
            sum = sum + 4.0/(1.0+x*x);
        }
    }
    pi = sum * step;
}
```

The reduction clause

Reduction is a very common pattern where all threads accumulate values into a single variable

`reduction(operator:list)`

- Valid operators are: +,-,*,|,||,&,&&,^,min, max
- The compiler creates a `private` copy of each variable in list that is properly initialized to the identity value
- At the end of the region, the compiler ensures that the `shared` variable is properly (and safely) updated with the partial values of each thread, using the specified operator

First example: computation of PI

```
...
void main ()
{
    int i, id;
    double x, pi, sum;

    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel private(x, i, id) reduction(+:sum)
    {
        id = omp_get_thread_num();
        for (i=id+1; i<=num_steps; i=i+NUM_THREADS) {
            x = (i-0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
        }
    }
    pi = sum * step;
}
```

Locks

OpenMP provides lock primitives for low-level synchronization

`omp_init_lock`	Initialize the lock
`omp_set_lock`	Acquires the lock
`omp_unset_lock`	Releases the lock
`omp_test_lock`	Tries to acquire the lock (won't block)
`omp_destroy_lock`	Frees lock resources

Locks

Example

```
#include <omp.h>
void foo ()
{
    omp_lock_t lock;

    omp_init_lock(&lock);
    #pragma omp parallel
    {
        omp_set_lock(&lock);
        // mutual exclusion region
        omp_unset_lock(&lock);
    }
    omp_destroy_lock(&lock);
}
```

Lock must be initialized before being used

Only one thread at a time here

Outline

- OpenMP overview
- OpenMP model
- Creating threads and accessing data
- Some API calls
- The single construct
- Thread synchronization
- Memory consistency

The flush construct

Relaxed consistency memory model

- A thread's temporary view of memory is not required to be consistent with memory at all times
- A value written to a variable can remain in the thread's temporary view until it is forced to memory at a later time
- Likewise, a read from a variable may retrieve the value from the thread's temporary view, unless it is forced to read from memory

The flush construct

```
#pragma omp flush (list)
```

- It enforces consistency between the temporary view and memory for those variables in list
- Synchronization (implicit or explicit) constructs have an associated flush operation

Part II

Task Parallelism in OpenMP

Outline

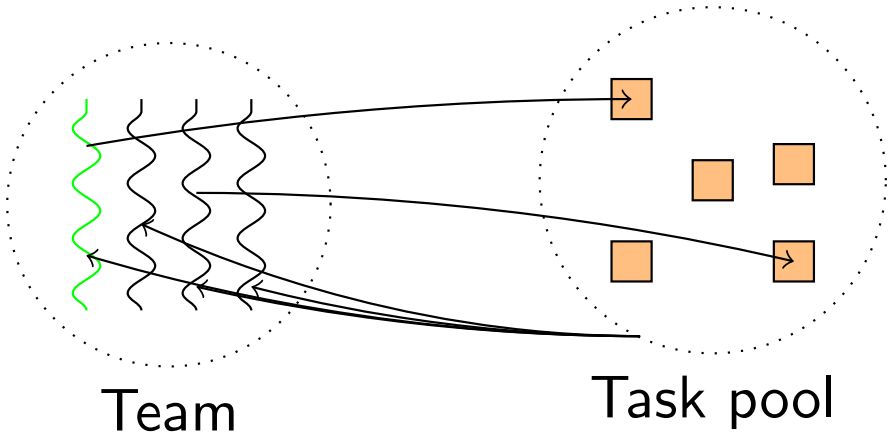
- OpenMP tasks
- Task synchronization
- Taskloop construct
- Reductions in the tasking model

Outline

- OpenMP tasks
- Task synchronization
- Taskloop construct
- Reductions in the tasking model

Task parallelism model

- Tasks are work units whose execution may be deferred
  - they can also be executed immediately
- Threads of the team cooperate to execute them



Creating tasks

Implicit and explicit tasks

- A `parallel` region creates tasks
  - One implicit task is created and assigned to each thread in the team of threads
- Each thread that encounters a `task` construct
  - Packages the code and data
  - Creates a new explicit task

Creating (explicit) tasks

The task construct

```
#pragma omp task [clauses]
        structured block
```

Where some possible clauses are:

- `shared`
- `private`
- `firstprivate`
  - Values are captured at creation time
- `if(`*`expression`*`)`
- `final(`*`expression`*`)`
- `mergeable`

Example: list traversal

```
void traverse_list ( List l )
{
  Element e;
  for ( e = l->first; e ; e = e->next )
     #pragma omp task
     process(e);
}
```

Example: list traversal

Completing the picture

We need threads to execute the tasks ...

```
List l

#pragma omp parallel
traverse_list(l);
...

void traverse_list ( List l )
{
   Element e;
   for ( e = l->first; e ; e = e->next )
      #pragma omp task     // e is firstprivate by default
      process(e);
}
```

... but not that many! This will generate multiple traversals

Example: list traversal

Using `single` ...

```
List l

#pragma omp parallel
#pragma omp single
traverse_list(l);
...

void traverse_list ( List l )
{
  Element e;
  for ( e = l->first; e ; e = e->next )
     #pragma omp task
     process(e);
}
```

One thread creates the tasks of the traversal. The rest (and this one once task generation is finished) cooperate to execute them

Default task data-sharing attributes

When no data clauses are specified, some rules apply:

- Global variables are shared
- Variables declared in the scope of a task are private
- The rest are `firstprivate` except when a `shared` attribute can be lexically inherited

Task default data-sharing attributes

In practice...

```
int a;
void foo() {
    int b,c;
    #pragma omp parallel
        {
        #pragma omp parallel private(b)
            {
            int d;
            #pragma omp task
                {
                int e;

                a = // shared
                b = // firstprivate
                c = // shared
                d = // firstprivate
                e = // private
}}}}
```

The `if` clause: immediate task execution

- If the expression of an `if` clause evaluates to false
  - The encountering task is suspended
  - The new task is executed immediately
    - with its own data environment
    - as a different task with respect to synchronization
  - The parent task resumes when the new task finishes
  - Allows implementations to optimize task creation

The `final` clause: immediate task execution (nested)

- If the expression of a `final` clause evaluates to `true`
  - The generated task and all of its child tasks will be final
  - The execution of a final task is sequentially **included** in the generating task (executed immediately)
- When a `mergeable` clause is present on a task construct, and the generated task is an **included** task, the implementation may generate a merged task instead (i.e. no task and context creation for it).

Final and mergeable tasks (data race!)

```
int fib(int n) {
  int i, j;

  if (n<2)
    return n;
  #pragma omp task shared(i) final(n <= THOLD) mergeable
  i=fib(n-1);
  #pragma omp task shared(j) final(n <= THOLD) mergeable
  j=fib(n-2);

  ....

  return i+j;
}
```

Outline

- OpenMP tasks
- Task synchronization
- Taskloop construct
- Reductions in the tasking model

Task synchronization

There are two types of task barriers:

- `taskwait`
  - Suspends the current task waiting on the completion of **child tasks** of the current task. The taskwait construct is a stand-alone directive
- `taskgroup`
  - Suspends the current task at the end of structured block waiting on completion of **child tasks** of the current task **and their descendent** tasks

Taskwait

```
#pragma omp taskwait
```

```
#pragma omp task {}          // T1
#pragma omp task             // T2
{
    #pragma omp task {}      // T3
}
#pragma omp task {}          // T4

#pragma omp taskwait
                  <-- Only T1, T2 and T4 are guaranteed to have finished here
}
```

Taskwait for correct Fibonacci parallelization

```
int fib(int n) {
  int i, j;

  if (n<2)
    return n;
  #pragma omp task shared(i) final(n <= THOLD) mergeable
  i=fib(n-1);
  #pragma omp task shared(j) final(n <= THOLD) mergeable
  j=fib(n-2);

  #pragma omp taskwait
  return i+j;
}
```

Taskgroup

```
#pragma omp taskgroup
        structured block
```

```
#pragma omp task {}                 // T1
#pragma omp taskgoup
{
    #pragma omp task                // T2
    {
        #pragma omp task {}         // T3
    }
    #pragma omp task {}             // T4
}
```

← Only T2, T3 and T4 are guaranteed to have finished here

Data sharing inside tasks

In addition one can use `critical` and `atomic` to synchronize the access to shared data inside the task

```
void process (Element e)
{
   ...
   #pragma omp atomic
   solutions_found++;
   ...
}
...
```

Task dependences

Definition of dependences between sibling tasks (i.e. from the same father)

```
#pragma omp task [depend (in : var_list)]
                 [depend (out : var_list)]
                 [depend (inout : var_list)]
```

Task dependences are derived from the dependence type (`in`, `out` or `inout`) and its items in `var_list`. This list may include array sections

Task dependences

- The `in` dependence-type: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `out` or `inout` dependence-type list
- The `out` and `inout` dependence-types: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `in`, `out`, or `inout` dependence-type list.

Task dependences

Example: wave-front execution

```
#pragma omp parallel private(i, j)
#pragma omp single
{
    for (i=1; i<n i++) {
        for (j=1; j<n;j++) {
            #pragma omp task
                    depend(in : block[i-1][j], block[i][j-1])
                    depend(out: block[i][j])
            foo(i,j);
        }
    }
}
```

Outline

- OpenMP tasks
- Task synchronization
- Taskloop construct
- Reductions in the tasking model

Creating (explicit) tasks from loop iterations

The taskloop construct

```
#pragma omp taskloop [clauses]
    for( init-expr ; test-expr ; inc-expr )
```

specifies that the iterations of one or more associated loops will be executed in parallel using OpenMP tasks. Implicit `taskgroup` synchronization associated with `taskloop`

Some clauses are used to specify data sharing attributes:

- `shared(list)`
- `private(list)`
- `firstprivate(list)`

Creating (explicit) tasks from loop iterations

Other clauses to control task generation:

- `grainsize(n)`
- `num_tasks(n)`
- `collapse(n)`
- `if(`*`expression`*`)`
- `final(`*`expression`*`)`
- `mergeable`

Or to override the implicit `taskgroup` associated with the `taskloop` construct:

- `nogroup`

Taskloop example

Granularity: BS iterations per task

```
void vector_add(int *A, int *B, int *C, int n) {
    #pragma omp taskloop grainsize(BS)
    for (int i=0; i< n; i++)
        C[i] = A[i] + B[i];
}
void main() {
    #pragma omp parallel
    #pragma omp single
    ... vector_add(a, b, c, N); ...
}
```

or alternatively

```
#pragma omp taskloop num_tasks(n/BS)
```

Outline

- OpenMP tasks
- Task synchronization
- Taskloop construct
- Reductions in the tasking model

Reduction clauses

Directives and clauses associated

```
#pragma omp taskloop
        [{reduction | in_reduction}(op:list)}]

#pragma omp task [in_reduction(op:list)]

#pragma omp taskgroup [task_reduction(id:list)]
```

Task reductions examples (1)

Reductions with explicit tasks always occur in the environment of a `taskgroup` (explicit if not implicit), which delimits the scope of reduction operation:

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp taskgroup task_reduction(+: sum)
    {
    for (i=0; i< SIZE; i++)
        #pragma omp task firstprivate(i) in_reduction(+: sum)
        sum += X[i];
    }
}
```

```
#pragma omp parallel
#pragma omp single
{
    // implicit taskgroup in taskloop construct
    #pragma omp taskloop reduction(+: sum)
    for (i=0; i< SIZE; i++)
        sum += X[i];
}
```

Task reductions examples (2)

Reductions with explicit tasks always occur in the environment of a `taskgroup` (explicit if not implicit), which delimits the scope of reduction operation:

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp taskgroup task_reduction(+: sum)
    {
    for (i=0; i< SIZE; i++)
        #pragma omp task firstprivate(i) in_reduction(+: sum)
        sum += X[i];

    #pragma omp taskloop in_reduction(+: sum)
    for (i=0; i< SIZE; i++)
        sum += X[i];
    }
}
```

Parallelism (PAR)

Short tutorial on OpenMP 5.0 tasking

Eduard Ayguadé[1] and Alex Durán[2]

[1]Barcelona Supercomputing Center – Universitat Politècnica de Catalunya
[2]Intel Corporation