OpenMP 3.0: What's new?

Alejandro Duran

Barcelona Supercomputing Center (BSC)
Computer Architecture Department
Universitat Politecnica de Catalunya

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

What's new in 3.0?

- Task parallelism
- Loop parallelism improvements
- Nested parallelism improvements
- Odds and ends

Why tasks parallelism?

- Main change to OpenMP 3.0
- Allows to parallelize irregular problems
  - unbounded loops
  - recursive algorithms
  - producer/consumer
  - ...

Task in OpenMP

- Tasks are work units which execution may be deferred
  - they can also be executed immediately
- Tasks are composed of:
  - **code** to execute
  - **data** environment
  - internal **control variables** (ICV)
    - change from 2.5!

Task in OpenMP

- Tasks are executed by threads of the **team**
- Task data environment is constructed at **creation** time
- Task can be **tied** to a thread
  - Only that thread can execute it

Parallel regions in 3.0

- The thread encountering a **parallel** construct
  - Creates as many implicit tasks as threads in team
  - Creates the team of threads
  - Implicit tasks are tied
    - one for each thread in the team

Task directive

#pragma omp task *[clause[[,] clause] ...]*

structured block

- Each encountering thread creates a new task
  - Packages code and data
- Can be nested
  - into another task
  - into a worksharing construct

Task directive clauses

- data scoping clauses:
  - **shared**(*list*)
  - **private**(*list*)
  - **firstprivate**(*list*)
  - **default**(*shared*|*none*)
- scheduling clauses:
  - **untied**
- other clauses:
  - **if** (*expr*)

Task synchronization

- Barriers (implicit or explicit):
  - All tasks created by any thread of the current team are guaranteed to be completed at barrier exit.
- Task barrier

  #pragma omp taskwait

  - Encountering task suspends until child tasks complete
    - Only direct child not descendants!

Simple example

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

N foo task created here one for each thread

All foo tasks guaranteed to be completed here

One bar task created here

Bar task guaranteed to be completed here

Data scoping rules

- Most rules from parallel regions apply
  - static variables are shared
  - global variables are shared
  - automatic storage variable are private
  - ...
  - default clause applies to the rest of variables

Data scoping rules

- If no default clause
  - **orphaned** tasks vars are **firstprivate** by default
  - non-orphaned tasks **shared** attribute is **inherit**
    - vars are firstprivate unless shared in the enclosing context

Fibonacci example

```
int fib ( int n )
{
    int x,y;
    if ( n < 2 ) return n;

    x = fib(n-1);

    y = fib(n-2);

    return x+y;;
}
```

Fibonacci example

```
int fib ( int n )
{
    int x,y;
    if ( n < 2 ) return n;
#pragma omp task
    x = fib(n-1);
#pragma omp task
    y = fib(n-2);
#pragma omp taskwait
    return x+y;;
}
```

guarantees results are ready

Fibonacci example

```
int fib ( int n )
{
    int x,y;
    if ( n < 2 ) return n;
#pragma omp task
    x = fib(n-1);
#pragma omp task
    y = fib(n-2);
#pragma omp taskwait
    return x+y;;
}
```

Correct

n is firstprivate

Wrong!

x,y are firstprivate

Fibonacci example

```
int fib ( int n )
{
    int x,y;
    if ( n < 2 ) return n;
#pragma omp task shared(x)
    x = fib(n-1);
#pragma omp task shared(y)
    y = fib(n-2);
#pragma omp taskwait
    return x+y;;
}
```

Correct

x,y are shared

List traversal

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

List traversal

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

Wrong!

e is shared here

List traversal

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

Right!

e is firstprivate

List traversal

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

Right!

e is firstprivate

Multiple list traversal

```
List l[N];
#pragma omp parallel
#pragma omp for
for ( int  i = 0; i < N; i++ ) {
      Element e;
      for ( e = l[i]->first; e ; e = e->next )
      #pragma omp task
            process(e);
}
```

Right!

e is firstprivate

Task scheduling: tied tasks

- By default, tasks are **tied** to the thread that **first executes** them
  - not the creator
- Tied tasks can be scheduled as the implementation wishes
  - Constraints:
    - Only the thread that the task is tied to can execute it
    - A task can only be suspended at a suspend point
      - task creation, task finish, taskwait, barrier
    - If the tasks is not suspended in a barrier it can only switch to a direct descendant of all tasks tied to the thread

Task scheduling: untied task

- Tasks created with the untied clause are never tied
- **No scheduling restrictions**
  - Can be suspended at any point
  - Can switch to any task
- More freedom to the implementation
  - Load balancing
  - Locality

Task scheduling: if clause

- If the the expression of a **if clause** evaluates to **false**
  - The encountering task is susended
  - The new task is **executed immediately**
    - own data environment
    - different task with respect to synchronization
  - The parent task resumes when the task finishes
- Useful to **optimize** the code
  - avoid creation of small tasks

Branch & bound

```
void branch ( int level, int m )
{
    int i;
    if (  solution() ) return;
    for ( i = 0; i < m; i++ )
        if ( !prune() )
            #pragma omp task untied if(level < LIMIT_LEVEL)
                branch(level+1,m);
}
```

Very unbalanced algorithms

- untied allows runtime to balance it better

Limits task creation after a certain level

level and m are firstprivate

Task pitfalls: Out of scope problem

```
void foo ()
{
    int a[LARGE_N];
    #pragma omp task shared(a)
    {
        bar(a);
    }
}
```

parent task may have exited foo by the time bar accesses a

- It's **users responsibility** to ensure data is alive

Task pitfalls: Out of scope problem

- One possible solution:

```
void foo ()
{
    int a[LARGE_N];
    #pragma omp task shared(a)
    {
        bar(a);
    }
    #pragma omp taskwait
}
```

guarantees data is still alive

Task pitfalls: untied tasks

```
int dummy;
#pragma omp threadprivate(dummy)

void bar() { dummy = ...; }
void foo () { ... = dummy; }

#pragma omp task untied
{
    foo();
    bar();
}
```

Wrong!

Task could switch to a different thread between foor and bar

Careful with untied tasks!

Task pitfalls: pointers

```
void foo (int n, char *state)
{
    int i;
    modify_state(state);
    for ( i = 0; i < n; i++ )
    #pragma omp task firstprivate(state)
        foo(n,state);
}
```

Every tasks needs its own state

Task pitfalls: pointers

```
void foo (int n, char *state)
{
    int i;
    modify_state(state);
    for ( i = 0; i < n; i++ )
    #pragma omp task firstprivate(state)
        foo(n,state);
}
```

Wrong!

Only the pointer is captured

All tasks modify the same state

Task pitfalls: pointers

- One solution: copy the data from the task

```
void foo (int n, char *state)
{
    int i;
    modify_state(state);
    for ( i = 0; i < n; i++ )
    #pragma omp task
    {
        char new_state[n];
        memcpy(new_state, state);
        foo(n,state);
    }
    #pragma omp taskwait
}
```

New state created for the task

Ensures original state does not go out of scope before copy

Loop parallelism improvements

- STATIC schedule guarantees
- Loop collapsing
- New induction variables types
- New AUTO schedule
- New schedule API

Static SCHEDULE guarantees

```
#pragma omp do schedule(static) nowait
    do i=1,N
        a(i) = ...
    enddo
#pragma omp do schedule(static)
    do i=1,N
        c(i) = a(i) + ...
    enddo
```

Wrong in 2.5

Static SCHEDULE guarantees

```
#pragma omp do schedule(static) nowait
    for ( i = 1; i < N; i++ )
        a[i] = ...
#pragma omp do schedule(static)
    for ( i = 1; i < N; i++ )
        c[i] = a[i] + ...
```

Right in 3.0 if (and only if)::

- number of iterations is the same
- chunk is the same (or no chunk)

Loop collapsing

```
do i = 1,N
  do j = 1,M
     do k = 1,K
        foo(i.j,k)
     enddo
  enddo
enddo
```

- loops i and j are parallel

If N and M are small and the number of processors is large

we need to get work from both loops!

Loop collapsing

```
!$omp parallel do
do i = 1,N
!$omp parallel do
   do j = 1,M
      do k = 1,K
         foo(i.j,k)
      enddo
   enddo
enddo
```

In 2.5:

Nested parallelism

- Unneeded sync
- High overhead

Loop collapsing

```
!$omp parallel do collapse(2)
do i = 1,N
  do j = 1,M
     do k = 1,K
        foo(i.j,k)
    enddo
  enddo
enddo
```

In 3.0:

Loop collapsing!

Iteration space from the two loops

is collapsed into a single one

Rules:

- Perfectly nested
- Rectangular iteration space

Loop collapsing

```
!$omp parallel do collapse(2)
do i = 1,N
   bar(i)
   do j = 1,M
       do k = 1,K
           foo(i.j,k)
      enddo
   enddo
enddo
```

illegal!

Not perfectly nested

Loop collapsing

```
!$omp parallel do collapse(2)
do i = 1,N
  do j = 1,i
    do k = 1,K
      foo(i.j,k)
    enddo
  enddo
enddo
```

illegal!

Triangular iteration space

New var types for loops

```
#pragma omp for
for ( unsigned int i = 0; i < N ; i++ )
    foo(i);

Vector v;
Vector::iterator it;
#pragma omp for
for ( it = v.begin(); it < v.end(); i++ )
    foo(i);
```

illegal types in 2.5

Legal in 3.0!

- unsigned integer types
- random access iterators (C++)

New var types for loops

```
Vector v;
Vector::iterator it;
#pragma omp for
for ( it = v.begin(); it != v.end(); i++ )
    foo(i);

char a[N];
#pragma omp for
for ( char *p = a; p < (a+N); p++ )
    foo(p)
```

ilegal relational operator!

legal

pointers are random access iterators

New SCHEDULE features

- **AUTO** schedule
  - Assignment of iterations to threads **decided** by the **implementation**
    - at compile time and/or execution time
    - from STATIC to advanced feedback guided schedules
- schedule API
  - new per-task ICV
  - omp_set_schedule
  - omp_get_schedule

Nested Parallelism improvements

- Multiple ICVs
- Nested parallelism API
- New environment variables

Multiple ICVs

- **Per task** Internal Control Variables
  - dyn-var
  - nest-var
  - nthreads-var
  - run-sched-var
- Each nested region can have its own behavior

Controlling parallel regions size

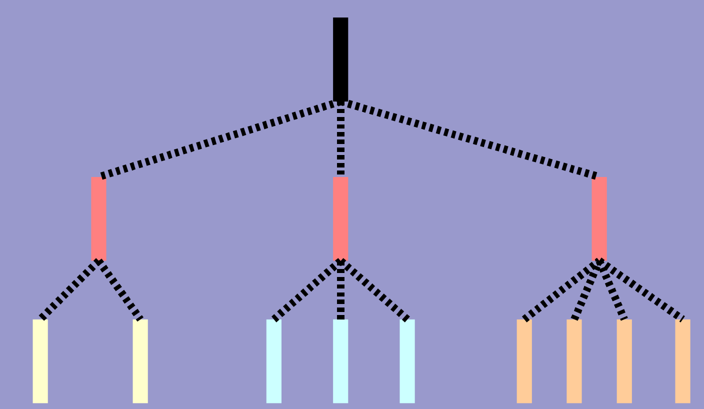
```
omp_set_num_threads(3);
#pragma omp parallel
{
    omp_set_num_threads(omp_get_thread_num()+2);
    #pragma omp parallel
        foo();
}
```

Unknow behavior in 2.5

Controlling parallel regions size

```
omp_set_num_threads(3);
#pragma omp parallel
{
    omp_set_num_threads(omp_get_thread_num()+2);
    #pragma omp parallel
        foo();
}
```

In 3.0, well defined



Other ICVs as well

```
omp_sched_t schedules[] = {
    omp_sched_static, omp_sched_dynamic, omp_sched_auto } ;
omp_set_num_threads(3)
#pragma omp parallel
{
    omp_set_schedule(schedules[omp_get_thread_num()],0);
    #pragma omp parallel for
        for ( i = 0; i < N; i++ ) foo(i);
}
```

Nested parallelism API

- New API, to obtain information about nested parallelism
  - How many nested parallel regions?

    omp_get_level()
  - How many active (with 2 or more threads) regions?

    omp_get_active_level()
  - Which thread-id was my ancestor?

    omp_get_ancestor_thread_num(level)
  - How many threads there are at previous regions?

    omp_get_team_size(level)

Nested parallelism env vars

- Control maximum number of active parallel regions

  OMP_MAX_NESTED_LEVEL

  omp_set_max_nested_levels()

  omp_get_max_nested_levels()

- Control maximum number of OpenMP threads created

  OMP_THREAD_LIMIT

  omp_get_thread_limit()

Odds and end

- New environment variables
  - Control of child threads' stack
    - OMP_STACKSIZE
  - Control of threads idle behavior
    - OMP_WAIT_POLICY
      - **active**
        - good for dedicated systems
      - **passive**
        - good for shared systems

Odds and ends

- C++ static class members can be threadprivate

```
class A  {
...
static int a;
#pragma omp threadprivate(a)
};
```

Thanks for your attention!

- Questions?