OpenMP Tasking: Past, Present and Future

Sergi Mateo Bellido

Compiler engineer

11/2018
Tasking history: past

- classical OpenMP
- Merge of the C/C++ and Fortran OpenMP spec

- task and taskwait construct
  - Basic set of tasking clauses
  - Spec in tasking terminology
Tasking history: OpenMP 3.0

- A task is a piece of code, its data environment and ICVs

```c
int foo(int x) {
    int res = 0;
    #pragma omp task shared(res) firstprivate(x)
    {
        res += x;
    }
    return res;
}
```

- Task’s execution may be deferred
  - Liveness of variables
  - Guarantee task completeness
Tasking history: OpenMP 3.0

- **if(expr) clause**
  - “Switch off” mechanism
  - The new task is *undefined* and executed immediately
  - The encountering task is suspended until the new task is completed
  - Data-sharing clauses are honored!

```c
int foo(int x) {
    printf("entering foo function\n");
    int res = 0;
    #pragma omp task shared(res) if(false)
    {
        res += x;
    }
    printf("leaving foo function\n");
}
```

Really useful to debug tasking applications :D
Tasking history: past

- Merge of the C/C++ and Fortran OpenMP spec
  - classical OpenMP
- task and taskwait construct
  - Basic set of tasking clauses
  - Spec in terms of tasks
- taskyield construct
- More basic clauses
**Tasking history: OpenMP 3.1**

- **final(expr) clause**
  - For recursive & nested applications
  - The new task is created and executed normally but in its context all tasks will be executed immediately by the same thread (*included tasks*)

```c
#pragma omp task final(e)
{
    #pragma omp task
    { ... }
    #pragma omp task
    { ... #C.1; #C.2 ... }
    #pragma omp taskwait
}
```

- Data-sharing clauses are honored too!!
- **The mergeable clause**
  - Optimizing

Unfortunately, there are no OpenMP commercial implementations taking advantage of `final` neither `mergeable` =(
Tasking history: past

- Merge of the C/C++ and Fortran OpenMP spec
- classical OpenMP
- task and taskwait construct
  - Basic set of tasking clauses
  - Spec in terms of tasks
- taskyield construct
  - More basic clauses
- Task dependences
- tasgroup construct
Tasking history: OpenMP 4.0

- **depend**(dep-type: list-items), where:
  - dep-type may be in, out or inout
  - item may be:

```c
int x = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(inout: x) //T1
    { ... }

    #pragma omp task depend(in: x)  //T2
    { ... }

    #pragma omp task depend(in: x)  //T3
    { ... }

    #pragma omp task depend(inout: x) //T4
    { ... }
}
```

- A task cannot be executed until all its predecessor tasks are completed.
- If a task defines an in dependence over a variable, the task will depend on all previously generated sibling tasks that reference that variable in an out or inout dependence.
- If a task defines an out/inout dependence over a variable, the task will depend on all previously generated sibling tasks that reference that variable in an in, out or inout dependence.
Tasking history: recent past

- **OpenMP 2.5**
  - Merge of the C/C++ and Fortran OpenMP spec
  - *classical OpenMP*

- **OpenMP 3.0**
  - `taskyield` construct
  - More basic clauses

- **OpenMP 3.1**
  - `tasgroup` construct
  - Task dependences

- **OpenMP 4.0**
  - `taskloop` construct
  - Priority clause

- **OpenMP 4.5**
  - `task and taskwait` construct
  - Basic set of tasking clauses
  - Spec in terms of tasks
Tasking history: OpenMP 4.5

- **The taskloop construct**
  - specifies that the iterations of a loop will be executed in parallel using tasks

```c
int foo(int n, int *v) {
    #pragma omp parallel for
    for(int i = 0; i < n; ++i)
        compute(v[i]);
}
```

- Use `num_tasks` or `grainsize` clauses to control the task granularity

```c
int foo(int n, int *v) {
    #pragma omp parallel
    #pragma omp single
    for(int i = 0; i < n; ++i) {
        #pragma omp task
        compute(v[i]);
    }
}
```

OpenMP 3.0

```c
int foo(int n, int *v) {
    #pragma omp parallel
    #pragma omp single
    #pragma omp taskloop
    for(int i = 0; i < n; ++i)
        compute(v[i]);
}
```

OpenMP 4.5
Tasking history: present

- Merge of the C/C++ and Fortran OpenMP spec
  - classical OpenMP

- **OpenMP 2.5**
  - task and taskwait construct
  - Basic set of tasking clauses
  - Spec in terms of tasks

- **OpenMP 3.0**
  - taskyield construct
  - More basic clauses

- **OpenMP 3.1**
  - tasgroup construct
  - Task dependences

- **OpenMP 4.0**
  - taskloop construct
  - Priority clause

- **OpenMP 4.5**
  - Task reductions
  - lvalue expressions in dependences
  - mutexinoutset dependency type
  - Dependences on taskwait construct
  - Dependable objects
  - Detach clause
  - Combined tasking constructs
  - Affinity clause
  - Iterators on deps and affinity clauses

- **OpenMP 5.0**
Affinity clause: hint to the runtime
- *Try to execute this task as close as possible to some data*

Iterators on deps (also on the affinity clause):
- define a dynamic number of dependences

```cpp
std::list<int> l = ...;
int n = l.size();
#pragma omp parallel
#pragma omp single
{
  for (int i = 0; i < n; ++i)
    #pragma omp task depend(out: l[i])
    compute_elem(l[i]); // Px

  #pragma omp task depend(iterator(j=0:n),in: l[j])
  print_elems(l);       // C
}
```
Tasking history: future?

- Merge of the C/C++ and Fortran OpenMP spec
  - classical OpenMP
  - task and taskwait construct
    - Basic set of tasking clauses
    - Spec in terms of tasks
- taskyield construct
  - More basic clauses
- tasgroup construct
  - Task dependences
- taskloop construct
  - Priority clause
- Task reductions
  - lvalue expressions in dependences
  - mutexinoutset dependency type
  - Dependences on taskwait construct
  - Dependable objects
  - Detach clause
  - Combined tasking constructs
  - Affinity clause
  - Iterators on deps and affinity clauses
Tasking history: OpenMP 5.X

- What I would like to see in OpenMP 5.X:
  - A way to express task dependences in a taskloop
  - Idem for task affinity
  - Task-only threads
  - inoutset dependency type
  - Better taskloop scheduling policies
  - ...

- What would you like to see?
Thank you!

sergi.mateo@bsc.es