Make the Most of OpenMP Tasking

Sergi Mateo Bellido
Compiler engineer
Outline

- Intro
- Data-sharing clauses
- Cutoff clauses
- Scheduling clauses
Intro: what’s a task?

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

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

- Task’s execution may be deferred
  - Liveness of variables
  - Guarantee task completeness

```c
int main() {
    int var = 0;
    #pragma omp parallel
    #pragma omp master
    var = foo(4);
}
```
Intro: the task construct

- Syntax:

```c
#pragma omp task [clauses]
{ structured-block }
```

C/C++

```fortran
!$omp task [clauses]
structured-block
!$omp end task
```

Fortran

- Where clauses can be:

<table>
<thead>
<tr>
<th>Data sharing clauses</th>
<th>Cutoff clauses</th>
<th>Scheduling clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>private</td>
<td>if(expr)</td>
<td>priority(expr)</td>
</tr>
<tr>
<td>firstprivate</td>
<td>final(expr)</td>
<td>untied</td>
</tr>
<tr>
<td>shared</td>
<td>mergeable</td>
<td>depend(dep-type: expr)</td>
</tr>
<tr>
<td>default(shared</td>
<td>none)</td>
<td></td>
</tr>
</tbody>
</table>
Intro: other task-generating constructs

- 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]);
}
```

- Several target constructs

```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]);
}
```

```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

OpenMP 3.0
Intro: task-synchronization constructs

- **The taskwait construct**
  - Waits on the completion of child tasks of the current task
  - Implicit task scheduling point

- **The taskgroup construct**
  - Waits on the completion of all descendant tasks of the current task created in the taskgroup region
  - Implicit task scheduling point

```c
#pragma omp task
{
    printf("child task\n");
    #pragma omp task
    printf("grandchild task\n");
}
#pragma omp taskwait
```

```c
#pragma omp taskgroup
{
    #pragma omp task
    {
        printf("Child task\n");
        #pragma omp task
        printf("Grandchild task\n");
    }
}
```
Data-Sharing clauses
Data-sharing clauses

- Why default data-shareings of a task construct are different from a parallel construct?
  - Synchronous vs Asynchronous execution

```c
int foo_parallel() {
    int r = 4;
    printf("1. &r=%p\n", &r);
    #pragma omp parallel
    #pragma omp single
    printf("2. &r=%p\n", &r);
}
```

1. &r=0x7ffd2b38010c
2. &r=0x7ffd2b38010c

```c
int foo_task() {
    int r = 4;
    printf("1. &r=%p\n", &r);
    #pragma omp task
    printf("2. &r=%p\n", &r);
}
```

1. &r=0x7ffd2b38010c
2. &r=0x7ffeae74ff8c

- Variables that don’t have a previous data-sharing are firstprivate
  - Be careful with arrays!!

- I recommend to use `default(none)`
Cutoff clauses
Cutoff clauses

- Cutoff clauses as a way to avoid creating small tasks!

- if(expr) clause
  - If expr evaluates to false, the execution of the task will be as if “the task construct was ignored”
    - deferred and executed immediately by the thread that was creating the task

```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!!!
Cutoff strategies: clauses (2)

- **final(expr) clause**
  - For recursive & nested applications, it stops task creation at a certain depth once we have enough parallelism
    - Reduce overhead!

```c
int fib(int n) {
    int r1, r2;
    #pragma omp task final(n < 10) shared(r1)
    r1 = fib(n-1);
    #pragma omp task final(n < 10) shared(r2)
    r2 = fib(n-2);
    #pragma omp taskwait
    return res1 + res2;
}
```

- **mergeable clause**
  - The implementation might merge the task’s data environment if the generated task is included or undeferred

No commercial implementation takes advantage of them :(
Scheduling clauses
Scheduling clauses

- Scheduling hints & constraints
- **untied clause**
  - Tasks are tied by default, using the untied clause they can potentially switch to another thread
- **priority(expr) clause**
  - Provides a HINT of the task’s relevance
  - Prioritize critical tasks
  - Balance unbalanced executions

```c
int foo(int N, int **elems, int *sizes)
{
    for (int i = 0; i < N; ++i) {
        #pragma omp task priority(sizes[i])
        compute_element(elems[i]);
    }
}
```
Scheduling clauses: depend clause

- Task dependences as a way to define task-execution constraints

```cpp
int x = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(in: x)
    std::cout << x << std::endl;

    #pragma omp task depend(inout: x)
    x++;
}
```

OpenMP 3.1

```
int x = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(in: x)
    std::cout << x << std::endl;
    x++;
}
```

OpenMP 4.0

Task dependences can help us to remove “strong” synchronizations, increasing the look ahead!!!
Scheduling clauses: depend clause (2)

- \texttt{depend(dep-type: list), where:}
  - \texttt{dep-type may be in, out or inout}
  - \texttt{list may be:}
    - A variable, e.g. \texttt{depend(inout: x)}
    - An array section, e.g. \texttt{depend(inout: a[10:20])}

- A task cannot be executed until all its predecessor tasks are completed

- If a task defines an \texttt{in} dependence over a variable
  - the task will depend on all previously generated sibling tasks that reference at least one of the list items in an \texttt{out} or \texttt{inout} dependence

- If a task defines an \texttt{out/inout} dependence over a variable
  - the task will depend on all previously generated sibling tasks that reference at least one of the list items in an \texttt{in}, \texttt{out} or \texttt{inout} dependence
Scheduling clauses: `depend` clause(2)

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

```c
// test1_raw.cc
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
        { ... }
}
```
Properly combining dependences and data-sharings allow us to define a task data-flow model

- Enhances the composability
- Eases the parallelization of new regions of your code

If all tasks are properly annotated, we only have to worry about the dependences & data-sharings of the new task!!!
Thank you

sergi.mateo@bsc.es