OpenMP Tasking, Part 2: Performant Tasking

Xavier Teruel
OpenMP Tasking, Part 1: Fundamentals

Xavier Teruel

PART 1 Outline: Fundamentals (July 30th)

The task construct

- Deferring (or not) a unit of work (executable for any member of the team)

```
#pragma omp task [clause[.,] clause]... [structured-block]
!omp task [clause[.,] clause]... [structured-block]
!omp end task
```

- Where clause is one of:
  - private(list)
  - firstprivate(list)
  - shared(list)
  - default(shared | none)
  - in_reduction(r-id: list)
  - allocate(allocator-list)
  - detach(event-handler)

  Data Environment

  - if(scalar-expression)
  - mergeable
  - final(scalar-expression)

  Cutoff Strategies

  - depend(dep-type: list)

  Synchronization

  - untied
  - priority(priority-value)
  - affinity(list)

  Task Scheduling

Tasking Use Cases

- Tasking use case: Fibonacci (recursion)
- Tasking use case: Cholesky (synchronization)
- Tasking use case: cpxy (blocking)

Pre-determined data-sharing attributes

- threadprivate variables are threadprivate (1)
- dynamic storage duration objects are shared (malloc, new,...) (2)
- static data members are shared (3)
- variables declared inside the construct
  - static storage duration variables are shared (4)
  - automatic storage duration variables are private (5)

  the loop iteration variable(s)...

  ```
  list A[SIZE];
  #pragma omp threadprivate(A)
  // ...
  #pragma omp task
  // A: threadprivate
  #pragma omp task
  // *p: shared
  #pragma omp task
  // foo_A.s: // ginfo: shared
  ```

The taskloop Construct

- Task generating construct: decompose a loop into chunks, create a task for each loop chunk

```
#pragma omp taskloop [clause[.,] clause]... [structured-for-loops]...
!omp taskloop [clause[.,] clause]... [structured-do-loops]...
!omp end taskloop
```

- Where clause is one of:
  - shared(list)
  - private(list)
  - firstprivate(list)
  - lastprivate(list)
  - default(shared | gc | Gc | none)
  - reduction(r-id: list)
  - in_reduction(r-id: list)
  - grainize(grain-size)
  - num_tasks(num-tasks)

  Data Environment

  - if(scalar-expression)
  - final(scalar-expression)
  - mergeable

  Cutoff Strategies

  - untied
  - priority(priority-value)

  Scheduler(R/H)

  - collapse(n)
  - nogroup

  Miscellaneous

  - allocate(allocator-list)

  Chunks/Grain
Slides and Exercises

https://tinyurl.com/openmp-umt-tasking

2021 Tasking UMT

- OpenMP-UMT-Exercises.tar.gz
- OpenMP-UMT-Tasking-1.pdf
- OpenMP-UMT-Tasking-2.pdf
PART 2 Outline: Performant Tasking

- Cutting-off strategies
  - Motivation
  - If-, final- and mergeable- clauses

- Task dependences
  - Motivation
  - What’s in the spec: depend clause, deps on taskwait and dependable objects
  - The data-flow philosophy

- More on performant tasking

* These slides are part of the tutorial “Mastering Tasking with OpenMP”; presented at SC and ISC conferences. Authors: Christian Terboven, Michael Klemm, Xavier Teruel, and Bronis R. de Supinski. All of them members of the OpenMP Language Committee.
Cutoff clauses and strategies
Parallel Brute-force Sudoku

- This parallel algorithm finds all valid solutions

<table>
<thead>
<tr>
<th>6</th>
<th>11</th>
<th>15</th>
<th>14</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>11</td>
<td>16</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>9</td>
<td>12</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>11</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>10</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>1</td>
<td>12</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>11</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>13</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>2</td>
<td>16</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>7</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>15</td>
<td>16</td>
<td>9</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>12</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>10</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

- (1) Search an empty field

- (2) Try all numbers:
  - Check Sudoku
    - If invalid: skip
    - If valid: Go to next field

- (3) Wait for completion

---

first call contained in a
#pragma omp parallel
#pragma omp single
such that one tasks starts the execution of the algorithm

#pragma omp task
needs to work on a new copy of the Sudoku board

#pragma omp taskwait
wait for all child tasks
Performance Evaluation

Sudoku on 2x Intel Xeon E5-2650 @2.0 GHz

- Blue: Intel C++ 13.1, scatter binding
- Black: speedup: Intel C++ 13.1, scatter binding

Runtime [sec] for 16x16 Sudoku on 2x Intel Xeon E5-2650 @2.0 GHz

#threads:

- Speedup
Event-based profiling:

Every thread is executing ~1.3M tasks… in ~5.7 seconds.

Average duration of a task is ~4.4 μs.

Tracing provides more details:

- Tasks get much smaller down the call-stack.
- Duration: 0.16 sec
- Duration: 0.001 sec
- Duration: 2.2 μs

If you have enough parallelism, stop creating more tasks!!

- if-clause, final-clause, mergeable-clause
- natively in your program code
Performance Evaluation (with cutoff)

Sudoku on 2x Intel Xeon E5-2650 @2.0 GHz

- Intel C++ 13.1, scatter binding
- Intel C++ 13.1, scatter binding, cutoff
- speedup: Intel C++ 13.1, scatter binding
- speedup: Intel C++ 13.1, scatter binding, cutoff

Graph shows runtime [sec] for 16x16 Sudoku on 2x Intel Xeon E5-2650 @2.0 GHz with varying number of threads.
The if clause

- Rule of thumb: the if(expression) clause as a “switch off” mechanism
  - Allows lightweight implementations of task creation and execution but it reduces the parallelism

- If the expression of the if clause evaluates to false
  - the encountering task is suspended
  - the new task is executed immediately (task dependences are respected!!)
  - the encountering task resumes its execution once the new task is completed
  - This is known as undeferred task

- Even if the expression is false, data-sharing clauses are honored
The final clause

- The final (expression) clause
  - Nested tasks / recursive applications
  - allows to avoid future task creation → reduces overhead but also reduces parallelism

- If the expression of the final clause evaluates to true
  - The new task is created and executed normally but in its context all tasks will be executed immediately by the same thread (included tasks)

- Data-sharing clauses are honored too!

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

- **The mergeable clause**
  - Optimization: get rid of “data-sharing clauses are honored”
  - This optimization can only be applied in *undeferred* or *included tasks*

- A Task that is annotated with the mergeable clause is called a *mergeable task*
  - A task that may be a *merged task* if it is an *undeferred task* or an *included task*

- A *merged task* is:
  - A task for which the data environment (inclusive of ICVs) may be the same as that of its generating task region

- A good implementation could execute a merged task without adding any OpenMP-related overhead
Example: Fibonacci
Fibonacci: without cutoff

```c
int fib(int n) {
    if (n < 2)
        return n;

    int res1, res2;
    #pragma omp task shared(res1)
    res1 = fib(n-1);

    #pragma omp task shared(res2)
    res2 = fib(n-2);

    #pragma omp taskwait
    return res1 + res2;
}
```
Fibonacci: if clause

```c
int fib(int n) {
    if (n < 2)
        return n;

    int res1, res2;
    #pragma omp task shared(res1) if(n > 30)
    res1 = fib(n-1);

    #pragma omp task shared(res2) if(n > 30)
    res2 = fib(n-2);

    #pragma omp taskwait
    return res1 + res2;
}
```

icc 2018.0
int fib(int n) {
    if (n < 30)
        return fib_serial(n);

    int res1, res2;
    #pragma omp task shared(res1)
    res1 = fib(n-1);

    #pragma omp task shared(res2)
    res2 = fib(n-2);

    #pragma omp taskwait

    return res1 + res2;
}
Task dependences
Task dependences as a way to define task-execution constraints

Motivation

Task dependences can help us to remove “strong” synchronizations, increasing the look ahead and, frequently, the parallelism!!!
Motivation: Cholesky factorization

void cholesky(int ts, int nt, double* a[nt][nt]) {
    for (int k = 0; k < nt; k++) {
        // Diagonal Block factorization
        potrf(a[k][k], ts, ts);
        // Triangular systems
        for (int i = k + 1; i < nt; i++) {
            #pragma omp task depend(inout: a[k][k])
            trsm(a[k][k], a[k][i], ts, ts);
        }
        #pragma omp taskwait
        // Update trailing matrix
        for (int i = k + 1; i < nt; i++) {
            for (int j = k + 1; j < i; j++) {
                #pragma omp task depend(inout: a[j][i])
                dgemm(a[k][i], a[k][j], a[j][i], ts, ts);
            }
            #pragma omp task depend(inout: a[i][i])
            syrk(a[k][i], a[i][i], ts, ts);
        }
        #pragma omp taskwait
    }
}

OpenMP 3.1

void cholesky(int ts, int nt, double* a[nt][nt]) {
    for (int k = 0; k < nt; k++) {
        // Diagonal Block factorization
        potrf(a[k][k], ts, ts);
        // Triangular systems
        for (int i = k + 1; i < nt; i++) {
            #pragma omp task depend(in: a[k][k])
            depend(inout: a[k][i])
            trsm(a[k][k], a[k][i], ts, ts);
        }
        #pragma omp taskwait
        // Update trailing matrix
        for (int i = k + 1; i < nt; i++) {
            for (int j = k + 1; j < i; j++) {
                #pragma omp task depend(inout: a[j][i])
                depend(in: a[k][i], a[k][j])
                dgemm(a[k][i], a[k][j], a[j][i], ts, ts);
            }
            #pragma omp task depend(inout: a[i][i])
            depend(in: a[k][i])
            syrk(a[k][i], a[i][i], ts, ts);
        }
        #pragma omp taskwait
    }
}

OpenMP 4.0
void cholesky(int ts, int nt, double* a[nt][nt]) {
    int k = 0;
    // Diagonal Block factorization
    potrf(a[k][k], ts, ts);
    // Triangular systems
    for (int i = k + 1; i < nt; i++) {
        #pragma omp task depend(in: a[k][k])
        depend(out: a[k][i])
        trsm(a[k][k], a[k][i], ts, ts);
    }
    #pragma omp taskwait
    // Update trailing matrix
    for (int i = k + 1; i < nt; i++) {
        for (int j = k + 1; j < i; j++) {
            #pragma omp task depend(in: a[j][i], a[k][j])
            dgemm(a[k][i], a[k][j], a[j][i], ts, ts);
        }
        #pragma omp task depend(out: a[i][i])
        syrk(a[k][i], a[i][i], ts, ts);
    }
    #pragma omp taskwait
}

Motivation: Cholesky factorization

Using 2017 Intel compiler
What’s in the spec: a bit of history

**OpenMP 4.0**
- The `depend` clause was added to the `task` construct

**OpenMP 4.5**
- The `depend` clause was added to the target constructs
- Support to `doacross` loops

**OpenMP 5.0**
- `lvalue` expressions in the `depend` clause
- New dependency type: `mutexinoutset`
- Iterators were added to the `depend` clause
- The `depend` clause was added to the `taskwait`
- Dependable objects

**OpenMP 5.1**
- New dependency type: `inoutset`
What’s in the spec: syntax `depend` clause

```
depend([depend-modifier,] dependency-type: list-items)
```

where:

→ `depend-modifier` is used to define iterators

→ `dependency-type` may be: `in`, `out`, `inout`, `inoutset`, `mutexinoutset` and `depobj`

→ A `list-item` may be:

  • C/C++: A lvalue expr or an array section  
    `depend(in: x, v[i], *p, w[10:10])`

  • Fortran: A variable or an array section  
    `depend(in: x, v(i), w(10:20))`
What’s in the spec: sema depend clause (1)

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

- If a task defines an *in* dependence over a list-item
  - the task will depend on all previously generated sibling tasks that reference that list-item in an *out* or *inout* dependence

- If a task defines an *out/inout* dependence over list-item
  - the task will depend on all previously generated sibling tasks that reference that list-item in an *in*, *out* or *inout* dependence
What’s in the spec: sema depend clause (1)

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

- If a task defines an in dependence over a list-item → the task will depend on all previously generated sibling tasks that reference that list-item in an out or inout dependence.

- If a task defines an out/inout dependence over a list-item → the task will depend on all previously generated sibling tasks that reference that list-item in an in, out or inout dependence.

```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
    { ... }
}
```
What's in the spec: sema depend clause (2)

- **Set types:** `inoutset` & `mutexinoutset`

```c
int x = 0, y = 0, res = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(out: res)  //T0
        res = 0;

    #pragma omp task depend(out: x)   //T1
        long_computation(x);

    #pragma omp task depend(out: y)   //T2
        short_computation(y);

    #pragma omp task depend(in: x)    depend(mutexinoutset: res)  //T3
        res += x;

    #pragma omp task depend(in: y)    depend(mutexinoutset: res)  //T4
        res += y;

    #pragma omp task depend(in: res)  //T5
        std::cout << res << std::endl;
}
```

1. **inoutset property:** tasks with a `mutexinoutset` dependence create a cloud of tasks (an `inoutset`) that synchronizes with previous & posterior tasks that dependent on the same list item

2. **mutex property:** Tasks inside the `inoutset` can be executed in any order but with mutual exclusion
What’s in the spec: sema depend clause (3)

- Task dependences are defined among **sibling tasks**

```c++
//test1.cc
int x = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(inout: x) //T1
    {
        #pragma omp task depend(inout: x) //T1.1
        x++;
        #pragma omp taskwait
    }
    #pragma omp task depend(in: x) //T2
    std::cout << x << std::endl;
}
```

- List items used in the depend clauses […] must indicate **identical** or **disjoint** storage

```c++
//test2.cc
int a[100] = {0};
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(inout: a[50:99]) //T1
    compute(/* from */ &a[50], /*elems*/ 50);
    #pragma omp task depend(in: a) //T2
    print(/* from */ a, /* elem */ 100);
}
```
What’s in the spec: sema depend clause (4)

- Iterators + deps: a way to define a dynamic number of dependences

```c
std::list<int> list = ...;
int n = list.size();

#pragma omp parallel
#pragma omp single
{
  for (int i = 0; i < n; ++i)
    #pragma omp task depend(out: list[i]) //Px
    compute_elem(list[i]);

  #pragma omp task depend(in: list[0], list[1], ..., list[n-1]) //C
  print_elems(list);
}
```

Equivalent to:

```
depend(in: list[0], list[1], ..., list[n-1])
```
What’s in the spec: deps on taskwait

Adding dependences to the taskwait construct

- Using a taskwait construct to explicitly wait for some predecessor tasks
- Syntactic sugar!

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

    #pragma omp task depend(in: y) //T2
    std::cout << y << std::endl;

    #pragma omp taskwait depend(in: x)
    std::cout << x << std::endl;
}
```
What’s in the spec: dependable objects (1)

- Offer a way to manually handle dependences
  - Useful for complex task dependences
  - It allows a more efficient allocation of task dependences
- New `omp_depend_t` opaque type
- 3 new constructs to manage dependable objects
  - `#pragma omp depobj(obj) depend(dep-type: list)`
  - `#pragma omp depobj(obj) update(dep-type)`
  - `#pragma omp depobj(obj) destroy`
What’s in the spec: dependable objects (2)

- Offer a way to manually handle dependences

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

    //T2
    #pragma omp task depend(in: x)
    std::cout << x << std::endl;
}
```

```c
int x = 0;
#pragma omp parallel
#pragma omp single
{
    omp_depend_t obj;
    #pragma omp depobj(obj) depend(inout: x)
    x++;

    #pragma omp task depend(depobj: obj) //T1
    x++;

    #pragma omp depobj(obj) update(in)

    #pragma omp task depend(depobj: obj) //T2
    std::cout << x << std::endl;

    #pragma omp depobj(obj) destroy
}
```
Philosophy: data-flow model

- Task dependences are orthogonal to data Sharings
  - **Dependences** as a way to define a **task-execution constraints**
  - Data-Sharings as **how the data is captured** to be used inside the task

```c
// test1.cc
int x = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(inout: x) \ 
    firstprivate(x) //T1
    x++;

    #pragma omp task depend(in: x) //T2
    std::cout << x << std::endl;
}
```

```c
// test2.cc
int x = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(inout: x) //T1
    x++;

    #pragma omp task depend(in: x) \ 
    firstprivate(x) //T2
    std::cout << x << std::endl;
}
```

OK, but it always prints ‘0’ :(  

We have a data-race!!
Philosophy: data-flow model (2)

- Properly combining dependences and data-sharings allow us to define a **task data-flow model**
  - Data that is read in the task → input dependence
  - Data that is written in the task → output dependence

- 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!!!
Summary

- Cutt-offs, may allow to control task granularity
  - Trade-off among overhead and amount of parallelism

- Dependences, may allow to avoid strong synchronization
  - Increase the look-ahead and the amount of parallelism

- More on performant tasks
  - Affinity clause, reduce runtime by improving data locality (hint)
  - Cancellation, avoid unneeded execution of tasks (when possible, e.g., pruning)
Thanks!
Slides and Exercises

https://tinyurl.com/openmp-umt-tasking