Tasking in OpenMP 5.0
Christian Terboven <terboven@itc.rwth-aachen.de>

June 22nd, 2021
NHR PerfLab Seminar Series
What is OpenMP?

• De-facto standard Application Programming Interface (API) to write shared memory parallel applications in C, C++, and Fortran
• Consists of Compiler Directives, Runtime routines and Environment variables

• Version 5.0 has been released at SC 2018
• Version 5.1 has been released at SC 2020
What is OpenMP?

• De-facto standard Application Programming Interface (API) to write shared memory parallel applications in C, C++, and Fortran
• Consists of Compiler Directives, Runtime routines and Environment variables

• Version 5.0 has been released at SC 2018
• Version 5.1 has been released at SC 2020
Motivation
### Sudoku for Lazy Computer Scientists

- **Lets solve Sudoku puzzles with brute multi-core force**

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

- **(1) Search an empty field**
- **(2) Try all numbers:**
  - **(2 a) Check Sudoku**
    - If invalid: skip
    - If valid: Go to next field
- **Wait for completion**
Parallel Brute-force Sudoku

- This parallel algorithm finds all valid solutions

<table>
<thead>
<tr>
<th>6</th>
<th>8</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>15</td>
<td>11</td>
<td>16</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:
  - (2 a) Check Sudoku
    - If invalid: skip
    - If valid: Go to next field
- Wait for completion

first call contained in a
#pragma omp parallel
#pragma omp single
such that one task 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
First Performance Evaluation

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

- Intel C++ 13.1, scatter binding
- Speedup: Intel C++ 13.1, scatter binding

Is this the best we can do?
What is a task in OpenMP?

• Tasks are work units whose execution
  - may be deferred or…
  - … can be executed immediately

• Tasks are composed of
  - **code** to execute, a **data** environment (initialized at creation time), internal **control** variables (ICVs)

• Tasks are created…
  ... when reaching a parallel region ✅ implicit tasks are created (per thread)
  ... when encountering a task construct ✅ explicit task is created
  ... when encountering a taskloop construct ✅ explicit tasks per chunk are created
  ... when encountering a target construct ✅ target task is created
Tasking execution model

- Supports unstructured parallelism
  - unbounded loops
    ```
    while ( <expr> ) {
      ...
    }
    ```
  - Several scenarios are possible:
    - single creator, multiple creators, nested tasks (tasks & WS)
  - All threads in the team are candidates to execute tasks

- Example (unstructured parallelism)
  ```
  #pragma omp parallel
  #pragma omp master
  while (elem != NULL) {
    #pragma omp task
    compute(elem);
    elem = elem->next;
  }
  ```
# The task construct

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

```plaintext
#pragma omp task [clause[[],] clause]...
{structured-block}
```

```plaintext
!$omp task [clause[[],] clause]...
...structured-block...
!$omp end task
```

### Where clause is one of:

<table>
<thead>
<tr>
<th>Private(list)</th>
<th>Data Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>- firstprivate(list)</td>
<td></td>
</tr>
<tr>
<td>- shared(list)</td>
<td></td>
</tr>
<tr>
<td>- default(shared</td>
<td>none)</td>
</tr>
<tr>
<td>- in_reduction(r-id: list)</td>
<td></td>
</tr>
</tbody>
</table>

### Cutoff Strategies

- if(scalar-expression)
- mergeable
- final(scalar-expression)
- depend(dep-type: list)

### Synchronization

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

### Task Scheduling

- allocate([allocator:] list)
- detach(event-handler)
The taskloop Construct

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

```c
#pragma omp taskloop [clause[[], clause]...] {structured-for-loops}
```

Where clause is one of:

- private(list)
- firstprivate(list)
- lastprivate(list)
- default(sh | pr | fp | none)
- reduction(r-id: list)
- in_reduction(r-id: list)
- grainsize(grain-size)
- num_tasks(num-tasks)

```c
!$omp taskloop [clause[[], clause]...] ...
structured-do-loops...
!$omp end taskloop
```

<table>
<thead>
<tr>
<th>Data Environment</th>
<th>Cutoff Strategies</th>
<th>Scheduler (R/H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>if(scalar-expression)</td>
<td>if(scalar-expression)</td>
<td>untied</td>
</tr>
<tr>
<td>final(scalar-expression)</td>
<td>mergeable</td>
<td>priority(priority-value)</td>
</tr>
<tr>
<td>mergeable</td>
<td></td>
<td>collapse(n)</td>
</tr>
<tr>
<td>mergeable</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>collapse(n)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>collapse(n)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>collapse(n)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>collapse(n)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>collapse(n)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
<tr>
<td>in_reduction(r-id: list)</td>
<td>nogroup</td>
<td>allocate({allocator:} list)</td>
</tr>
</tbody>
</table>
Task Synchronization
Task synchronization: taskwait directive

- The taskwait directive (shallow task synchronization)
  - It is a stand-alone directive
    ```
    #pragma omp taskwait
    ```
  - wait on the completion of child tasks of the current task; just direct children, not all descendant tasks; includes an implicit task scheduling point (TSP)

```c
#pragma omp parallel
#pragma omp single
{
#pragma omp task
{ ...
  #pragma omp task
  { ...
    #pragma omp task
    { ... #C.1; #C.2; ...}
  #pragma omp taskwait

  // implicit barrier will wait for C.x
} // implicit barrier will wait for C.x
```
Task synchronization: taskgroup construct

- The taskgroup construct (deep task synchronization)
  - attached to a structured block; completion of all descendants of the current task; TSP at the end

```c
#pragma omp taskgroup [clause[,,] clause]...
{structured-block}
```

```c
#pragma omp parallel
#pragma omp single
{
  #pragma omp taskgroup
  {
    #pragma omp task
    {
      ...
    }
    #pragma omp task
    {
      ...
      #C.1; #C.2;
      ...
    }
  }
  // end of taskgroup
}
```

### Diagram

![Diagram showing task synchronization with taskgroup construct](image)
Task Scheduling
Task scheduling: taskyield directive

- Task scheduling points (and the taskyield directive)
  - tasks can be suspended/resumed at TSPs \(\square\) some additional constraints to avoid deadlock problems
  - implicit scheduling points (creation, synchronization, ... )
  - explicit scheduling point: the taskyield directive

- `#pragma omp taskyield`

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

- **tied:**
  - `foo()`
  - `bar()`
  - `single`

- **untied:**
  - `foo()`
  - `bar()`
  - `single`

(default)
Task reductions (using taskgroup)

- Reduction operation
  - perform some forms of recurrence calculations
  - associative and commutative operators
- The (taskgroup) scoping reduction clause
  
  ```c
  #pragma omp taskgroup task_reduction(op: list)
  {structured-block}
  
  #pragma omp task in_reduction(op: list)
  {structured-block}
  ```

- The (task) in_reduction clause [participating]

  ```c
  int res = 0;
  node_t* node = NULL;
  ...
  #pragma omp parallel
  {
    #pragma omp single
    {
      #pragma omp taskgroup
      task_reduction(+: res)
      {
        // [1]
        while (node)
        {
          #pragma omp task in_reduction(+: res) firstprivate(node)
          {
            // [2]
            res += node->value;
            node = node->next;
          }
        // [3]
        }
      }
    }
  }
  
  #pragma omp task in_reduction(op: list)
  {structured-block}
  
  #pragma omp taskgroup task_reduction(op: list)
  {structured-block}
  ```

- Task participates in a reduction operation [2]
Task reductions (+ modifiers)

- Reduction modifiers
  - Former reductions clauses have been extended
  - task modifier allows to express task reductions
  - Registering a new task reduction [1]
  - Implicit tasks participate in the reduction [2]
  - Compute final result after [4]
- The (task) in_reduction clause [participating]

```c
int res = 0;
node_t* node = NULL;
...
#pragma omp parallel reduction(task,+: res)
{ // [1][2]
    #pragma omp single
    { #pragma omp taskgroup
        while (node) {
            #pragma omp task in_reduction(+: res) \ firstprivate(node)
            { // [3]
                res += node->value;
            }
            node = node->next;
        }
    }
} // [4]
```
Task Dependencies
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 for 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: 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 variable
  - the task will depend on all previously generated sibling tasks that reference at least one of the list items 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 at least one of the list items 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 variable:
  - the task will depend on all previously generated sibling tasks that reference at least one of the list items in an `in`, `out` or `inout` dependence

- If a task defines an `out` or `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 `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` & `mutxinoutset`

```cpp
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(mutxinoutset: res)  //T3
    res += x;

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

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

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

2. **mutex property:**
   - Tasks inside the inout set can be executed in any order but with mutual exclusion
Advanced features: 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;
}
```
Advanced features: 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

```c
int x = 0;
#pragma omp parallel
#pragma omp single
{
  omp_depend_t obj;
  #pragma omp depobj(obj) depend(inout: x) //T1
  x++;
  #pragma omp depobj(obj) update(in) //T2
  std::cout << x << std::endl;
  #pragma omp depobj(obj) destroy
}
```

```c
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(inout: x) //T1
  x++;
  #pragma omp task depend(depobj: obj) //T1
  #pragma omp depobj(obj) update(in)
  #pragma omp task depend(depobj: obj) //T2
  std::cout << x << std::endl;
  #pragma omp depobj(obj) destroy
}
```
Clauses to optimize Task Scheduling
Task scheduling: programmer’s hints

- Programmers may specify a priority value when creating a task

```
#pragma omp task priority(pvalue)
{structured-block}
```

- pvalue: the higher the better (will be scheduled earlier)
- once a thread becomes idle, gets one of the highest priority tasks

```
#pragma omp parallel
#pragma omp single
{
  for (i = 0; i < SIZE; i++) {
    #pragma omp task priority(1)
    { code_A; }
  }
  #pragma omp task priority(100)
  { code_B; }
  ...
}
```

Task pool priority-aware
affinity clause

• New clause: #pragma omp task affinity (list)
  - Hint to the runtime to execute task closely to physical data location
  - Clear separation between dependencies and affinity

• Expectations:
  - Improve data locality / reduce remote memory accesses
  - Decrease runtime variability

• Still expect task stealing
  - In particular, if a thread is under-utilized
A map is introduced to store location information of data that was previously used.

Do you still remember the Motivation?
Performance Evaluation

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

Runtime [sec] for 16x16

#threads

Speedup

Intel C++ 13.1, scatter binding
speedup: Intel C++ 13.1, scatter binding
Event-based profiling provides a good overview:

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: 0.047 sec
Duration: 0.001 sec
Duration: 2.2 μs
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

Runtime [sec] for 16x16

Speedup

#threads

1 2 3 4 5 6 7 8 9 10 11 12 16 24 32
What did I leave out?

• **Cancellation:** Cancellation (since OpenMP 4.0) provides a best-effort approach to terminate OpenMP regions
  - Best-effort: not guaranteed to trigger termination immediately
  - Triggered “as soon as” possible

• **Asynchronous API Interaction:** This provides a mechanism to marry asynchronous APIs with the parallel task model of OpenMP
  - How to synchronize completions events with task execution?