# Tasking in OpenMP 5.0

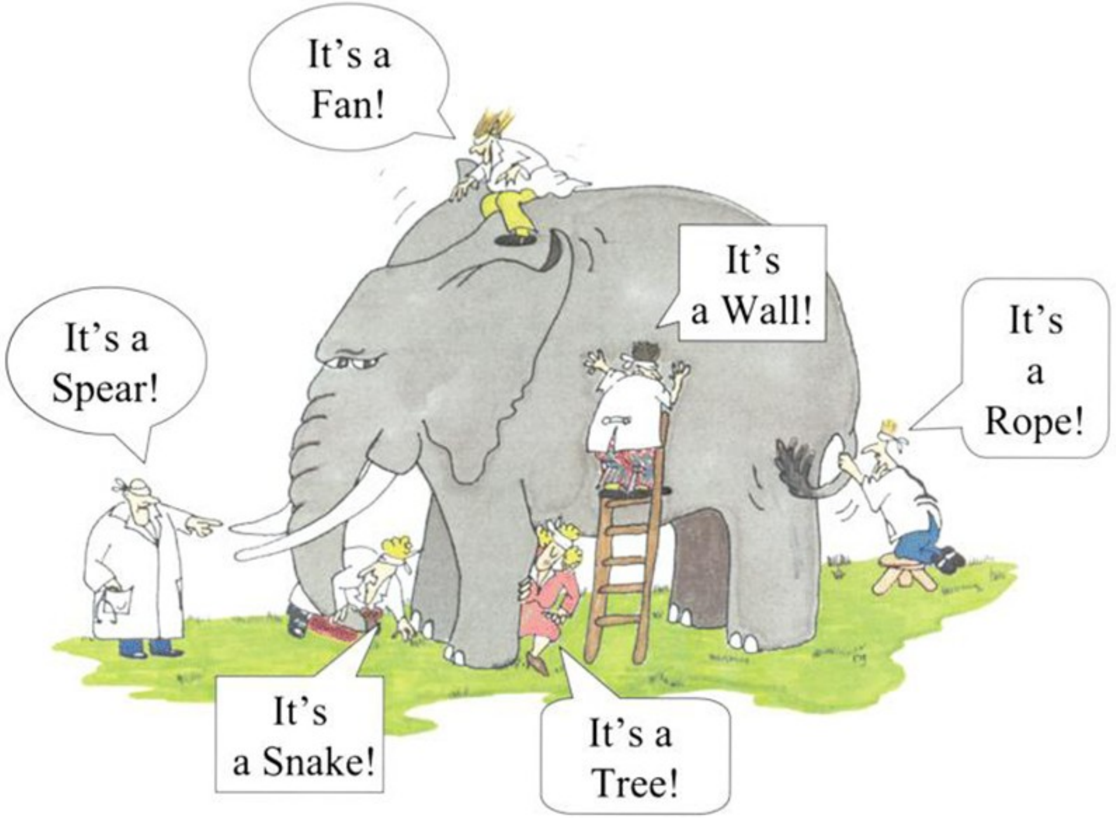
Christian Terboven <terboven@itc.rwth-aachen.de>

June 22nd, 2021
NHR PerfLab Seminar Series

# What is OpenMP?

- De-facto standard Application Programming Interface (API) to write <u>shared memory parallel</u> 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



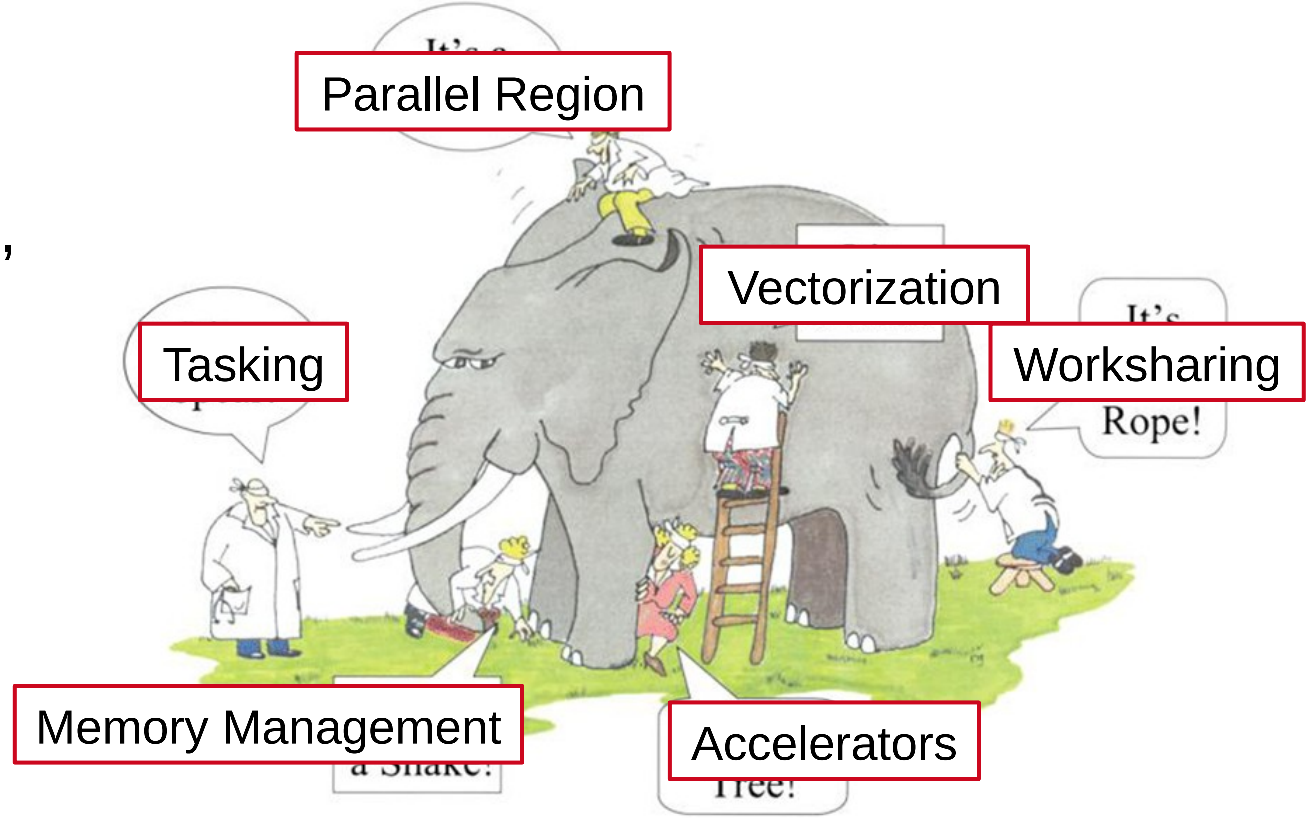



IT Center RWTH AACHEN UNIVERSITY

## What is OpenMP?

- De-facto standard Application Programming Interface (API) to write <u>shared memory parallel</u> 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

## Sudoko for Lazy Computer Scientists

- Lets solve Sudoku puzzles with brute multi-core force

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 6 | | | | | | 8 | 11 | | | 15 | 14 | | | 16 |
| 15 | 11 | | | | 16 | 14 | | | | 12 | | | 6 | | |
| 13 | | 9 | 12 | | | | | 3 | 16 | 14 | | 15 | 11 | 10 | |
| 2 | | 16 | | 11 | | 15 | 10 | 1 | | | | | | | |
| | 15 | 11 | 10 | | | 16 | 2 | 13 | 8 | 9 | 12 | | | | |
| 12 | 13 | | | 4 | 1 | 5 | 6 | 2 | 3 | | | | | 11 | 10 |
| 5 | | 6 | 1 | 12 | | 9 | | 15 | 11 | 10 | 7 | 16 | | | 3 |
| | 2 | | | | 10 | | 11 | 6 | | 5 | | | 13 | | 9 |
| 10 | 7 | 15 | 11 | 16 | | | | 12 | 13 | | | | | | 6 |
| 9 | | | | | | 1 | | | 2 | | 16 | 10 | | | 11 |
| 1 | | 4 | 6 | 9 | 13 | | | 7 | | 11 | | 3 | 16 | | |
| 16 | 14 | | | 7 | | 10 | 15 | 4 | 6 | 1 | | | | 13 | 8 |
| 11 | 10 | | 15 | | | | 16 | 9 | 12 | 13 | | | 1 | 5 | 4 |
| | | 12 | | 1 | 4 | 6 | | 16 | | | | 11 | 10 | | |
| | | 5 | | 8 | 12 | 13 | | 10 | | | 11 | 2 | | | 14 |
| 3 | 16 | | | 10 | | | 7 | | | 6 | | | | 12 | |

- (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

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 6 | | | | | | 8 | 11 | | | 15 | 14 | | | 16 |
| 15 | 11 | | | | 16 | 14 | | | | 12 | | | 6 | | |
| 13 | | 9 | 12 | | | | | 3 | 16 | 14 | | 15 | 11 | 10 | |
| 2 | | 16 | | 11 | | 15 | 10 | 1 | | | | | | | |
| | 15 | 11 | 10 | | | 16 | 2 | 13 | 8 | 9 | 12 | | | | |
| 12 | 13 | | | 4 | 1 | 5 | 6 | 2 | 3 | | | | | 11 | 10 |
| 5 | | 6 | 1 | 12 | | 9 | | 15 | 11 | 10 | 7 | 16 | | | 3 |
| | 2 | | | | 10 | | 11 | 6 | | 5 | | | 13 | | 9 |
| 10 | 7 | 15 | 11 | 16 | | | | 12 | 13 | | | | | | 6 |
| 9 | | | | | | 1 | | | 2 | | 16 | 10 | | | 11 |
| 1 | | 4 | 6 | 9 | 13 | | | 7 | | 11 | | 3 | 16 | | |
| 16 | 14 | | | 7 | | 10 | 15 | 4 | 6 | 1 | | | | 13 | 8 |
| 11 | 10 | | 15 | | | | 16 | 9 | 12 | 13 | | | 1 | 5 | 4 |
| | | 12 | | 1 | 4 | 6 | | 16 | | | | 11 | 10 | | |
| | | 5 | | 8 | 12 | 13 | | 10 | | | 11 | 2 | | | 14 |
| 3 | 16 | | | 10 | | | 7 | | | 6 | | | | 12 | |

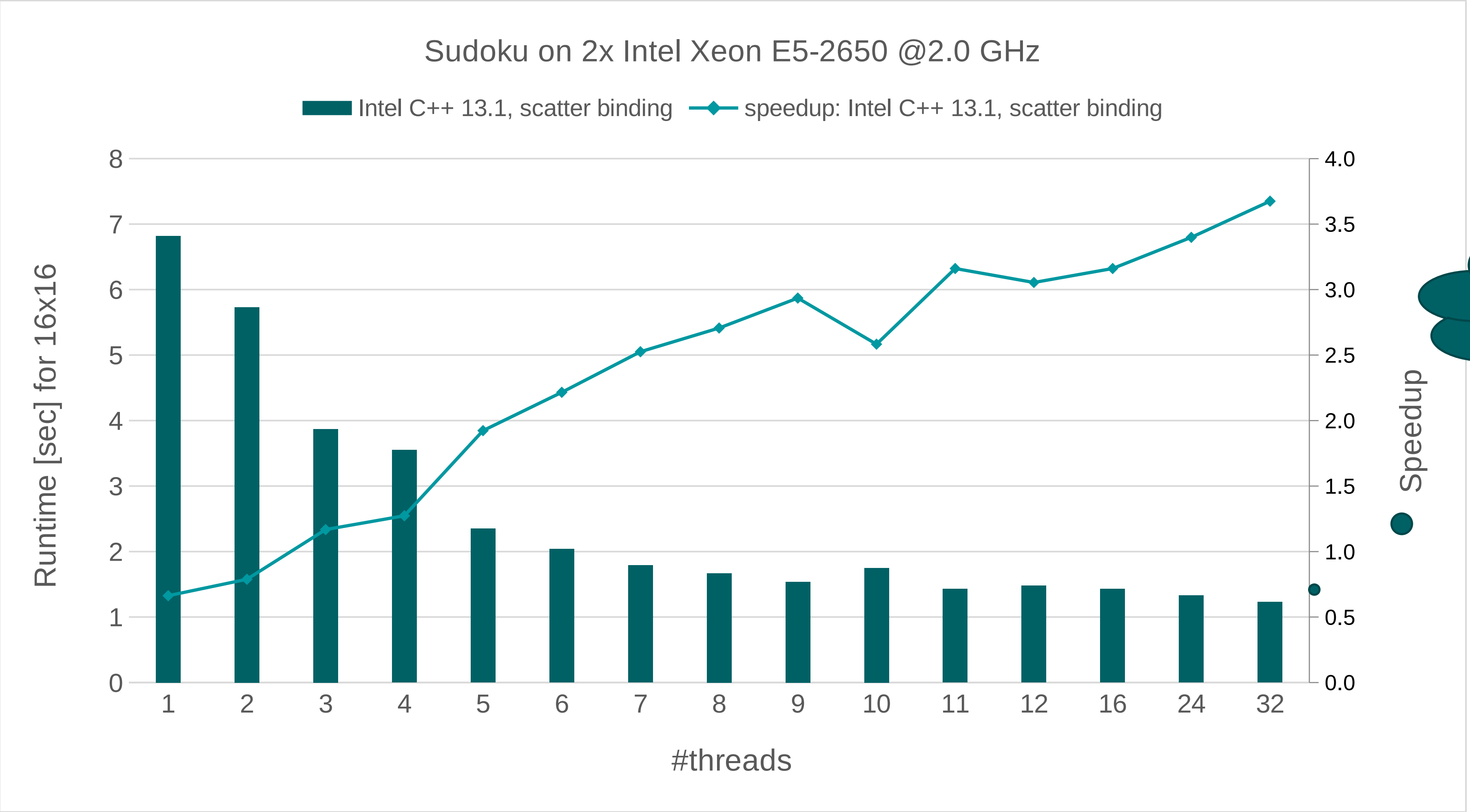
- (1) Search an empty field
- (2) Try all numbers:
  - (2 a) Check Sudoku
    - If invalid: skip
    - If valid: Go to next field
- Wait for completion

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

`#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




# Tasking Basics

# 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> ) {
   ...
}
```

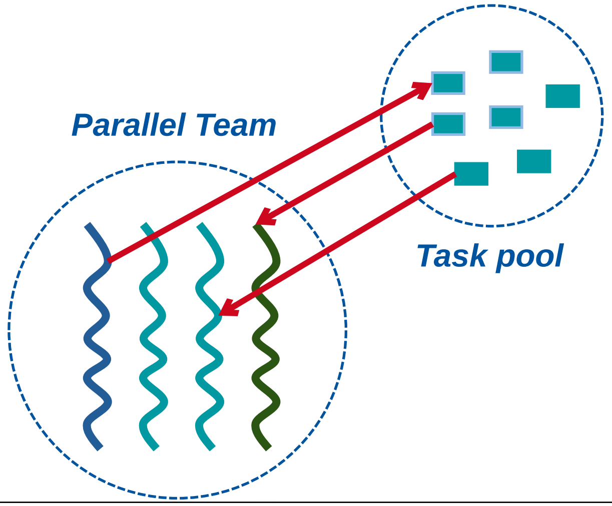
  -

```
void myfunc( <args> )
{
   ...; myfunc( <newargs> ); ...;
}
```

- 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)

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

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

Where clause is one of:

| | |
|---|---|
| → private(list)<br>→ firstprivate(list)<br>→ shared(list)<br>→ default(shared \| none)<br>→ in_reduction(r-id: list) | **Data Environment** |
| → allocate([allocator:] list)<br>→ detach(event-handler) | **Miscellaneous** |
| → if(scalar-expression)<br>→ mergeable<br>→ final(scalar-expression) | **Cutoff Strategies** |
| → depend(dep-type: list) | **Synchronization** |
| → untied<br>→ priority(priority-value)<br>→ affinity(list) | **Task Scheduling** |

# 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:

**Data Environment**
- → shared(list)
- → private(list)
- → firstprivate(list)
- → lastprivate(list)
- → default(sh | *pr* | *fp* | none)
- → reduction(r-id: list)
- → in_reduction(r-id: list)

**Chunks/Grain**
- → grainsize(grain-size)
- → num_tasks(num-tasks)

**Cutoff Strategies**
- → if(scalar-expression)
- → final(scalar-expression)
- → mergeable

**Scheduler (R/H)**
- → untied
- → priority(priority-value)

**Miscellaneous**
- → collapse(n)
- → nogroup
- → allocate([allocator:] list)

# Task Synchronization

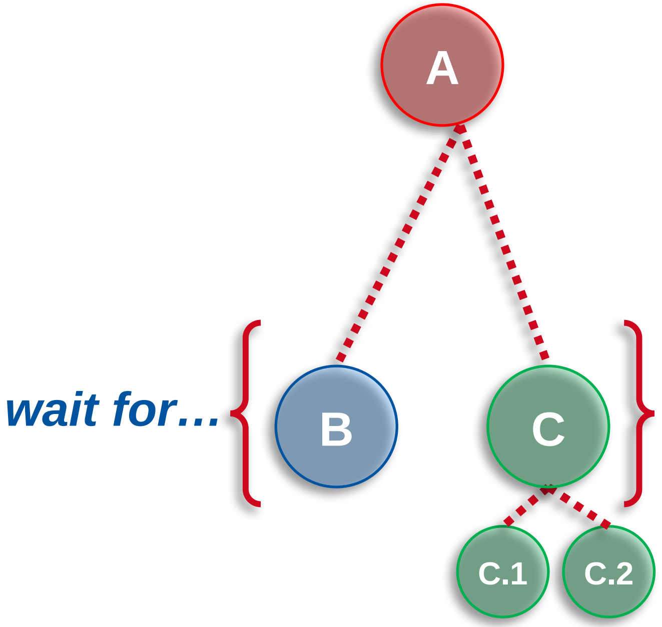
# Task synchronization: taskwait directive

- 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

```
#pragma omp parallel
#pragma omp single
{
  #pragma omp task            :A
  {
     #pragma omp task         :B
     { … }
     #pragma omp task         :C
     { … #C.1; #C.2; …}
     #pragma omp taskwait
  }
} // implicit barrier will wait for C.x
```



# Task synchronization: taskgroup construct

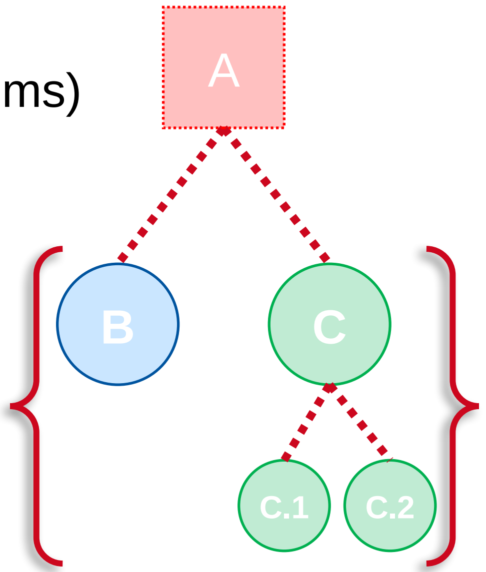
- The taskgroup construct (deep task synchronization)
  - attached to a structured block; completion of all descendants of the current task; TSP at the end

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

- entifier: list-items)

```
#pragma omp parallel
#pragma omp single
{
   #pragma omp taskgroup                 : A
   {
      #pragma omp task                   :B
      { … }
      #pragma omp task                   :C
      { … #C.1; #C.2; …}

   } // end of taskgroup
}
```

*wait for…*



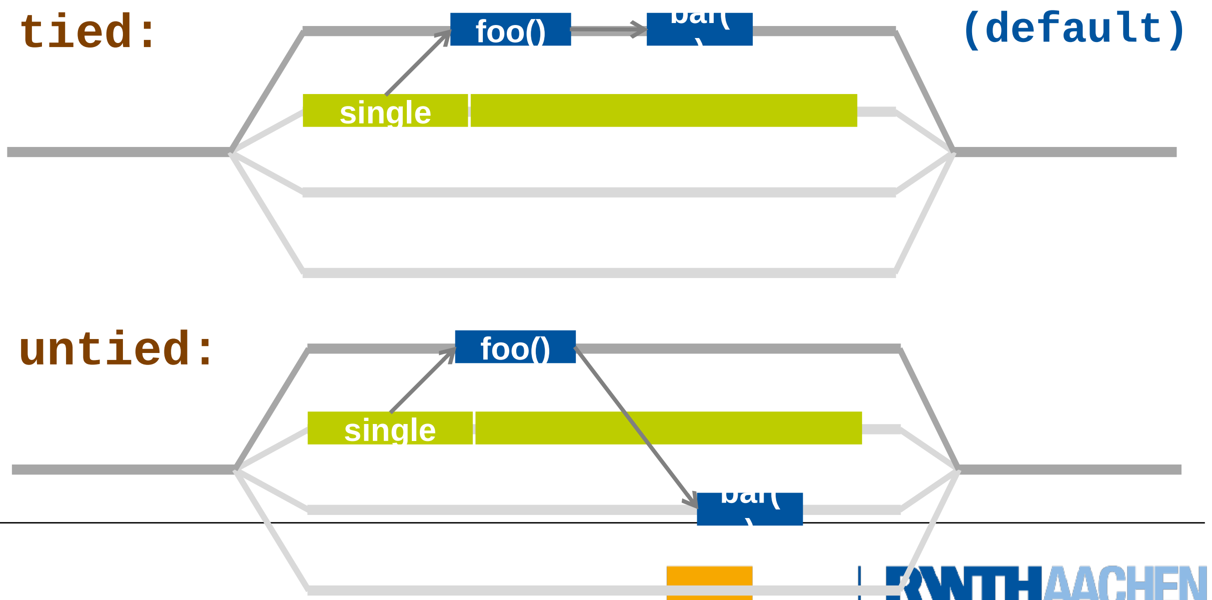
# Task Scheduling

# Task scheduling: taskyield directive

- Task scheduling points (and the taskyield directive)
  - tasks can be suspended/resumed at TSPs 🖂 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 untied
   {
      foo();
      #pragma omp taskyield
      bar()
   }
}
```



# Task reductions (using taskgroup)

- Reduction operation
  - perform some forms of recurrence calculations
  - associative and commutative operators
- The (taskgroup) scoping reduction clause

```
#pragma omp taskgroup task_reduction(op: list)
{structured-block}
```

```
#pragma omp task in_reduction(op: list)
{structured-block}
```

- The (task) in_reduction clause [participating]
  - Task participates in a reduction operation [2]

```
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]
  }
}
```

# 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]

```
#pragma omp task in_reduction(op: list)
{structured-block}
```

  - Task participates in a reduction operation [3]

```
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

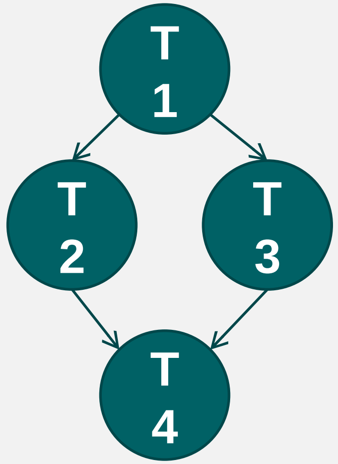
## What's in the spec: sema depend clause (1)

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

- If a task defin
  - the task will dep … of the list items in an out or inout d

- If a task defin
  - the task will dep … of the list items in an in, out or ino

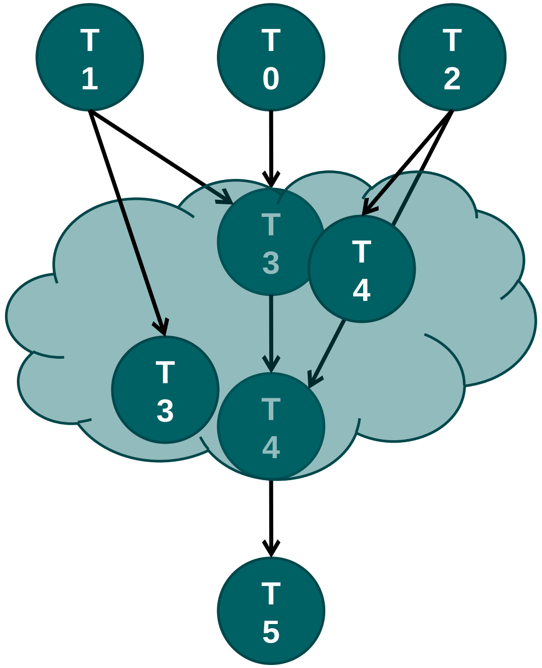
```
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(inout: x) //T1
  { ... }

  #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`

```
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 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`

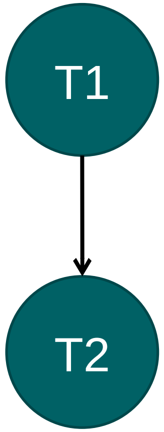
- Adding dependences to the `taskwait` construct
  - Using a `taskwait` construct to explicitly wait for some predecessor tasks
    - Syntactic sugar!

```
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`

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

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




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

# Clauses to optimize Task Scheduling

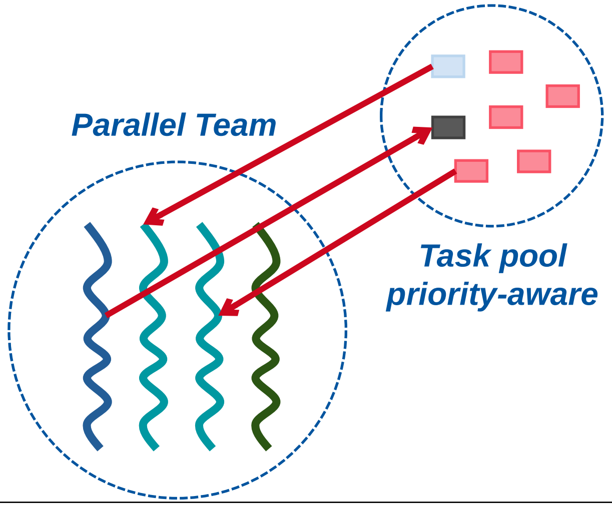
# Task scheduling: programmer's hints

- Programmers may specify a priority value when creating a task

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

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

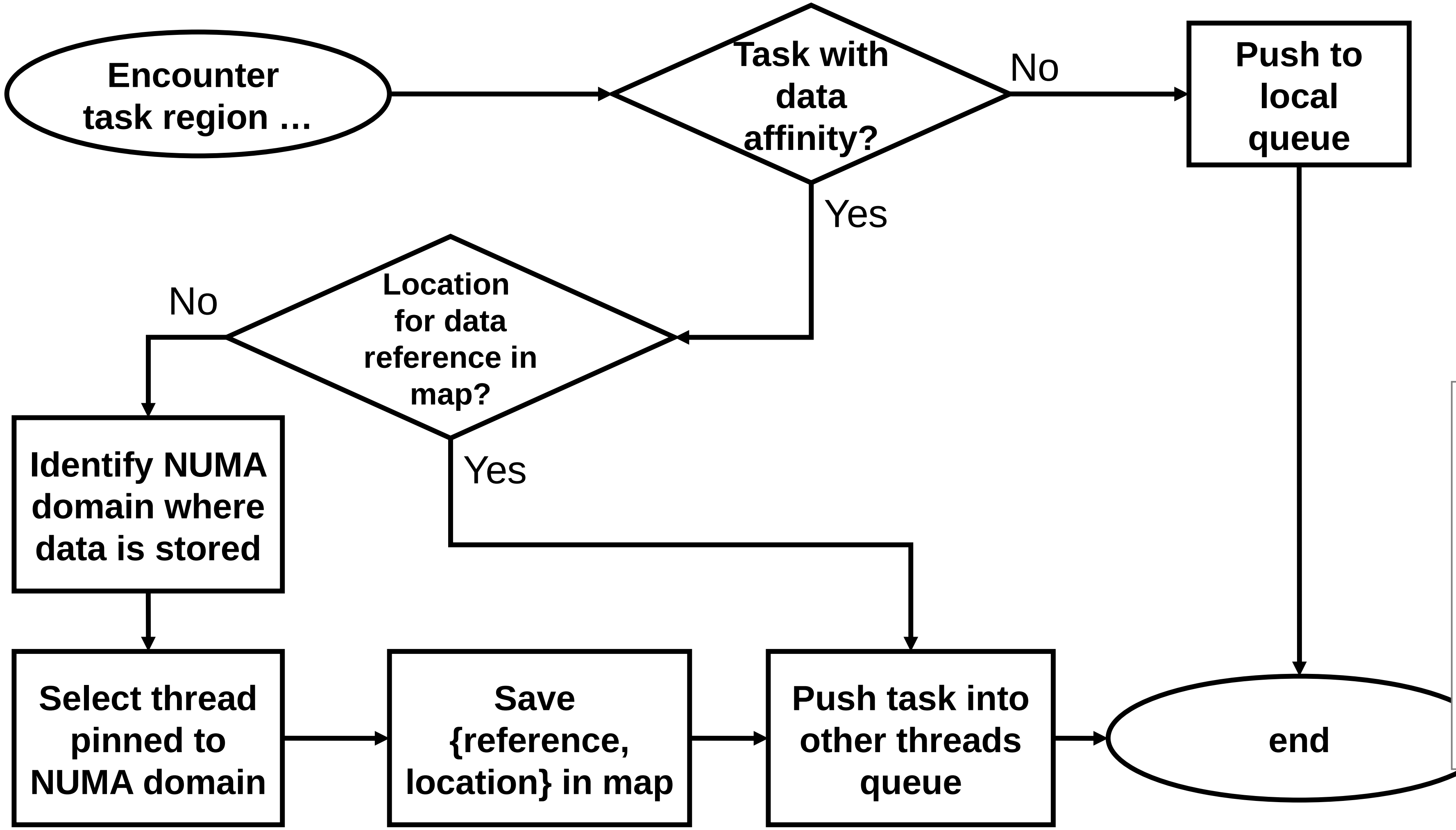
```
#pragma omp parallel
#pragma omp single
{
  for ( i = 0; i < SIZE; i++) {
    #pragma omp task priority(1)
    { code_A; }
  }
  #pragma omp task priority(100)
  { code_B; }
  ...
}
```



# `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

# Selected LLVM implementation details

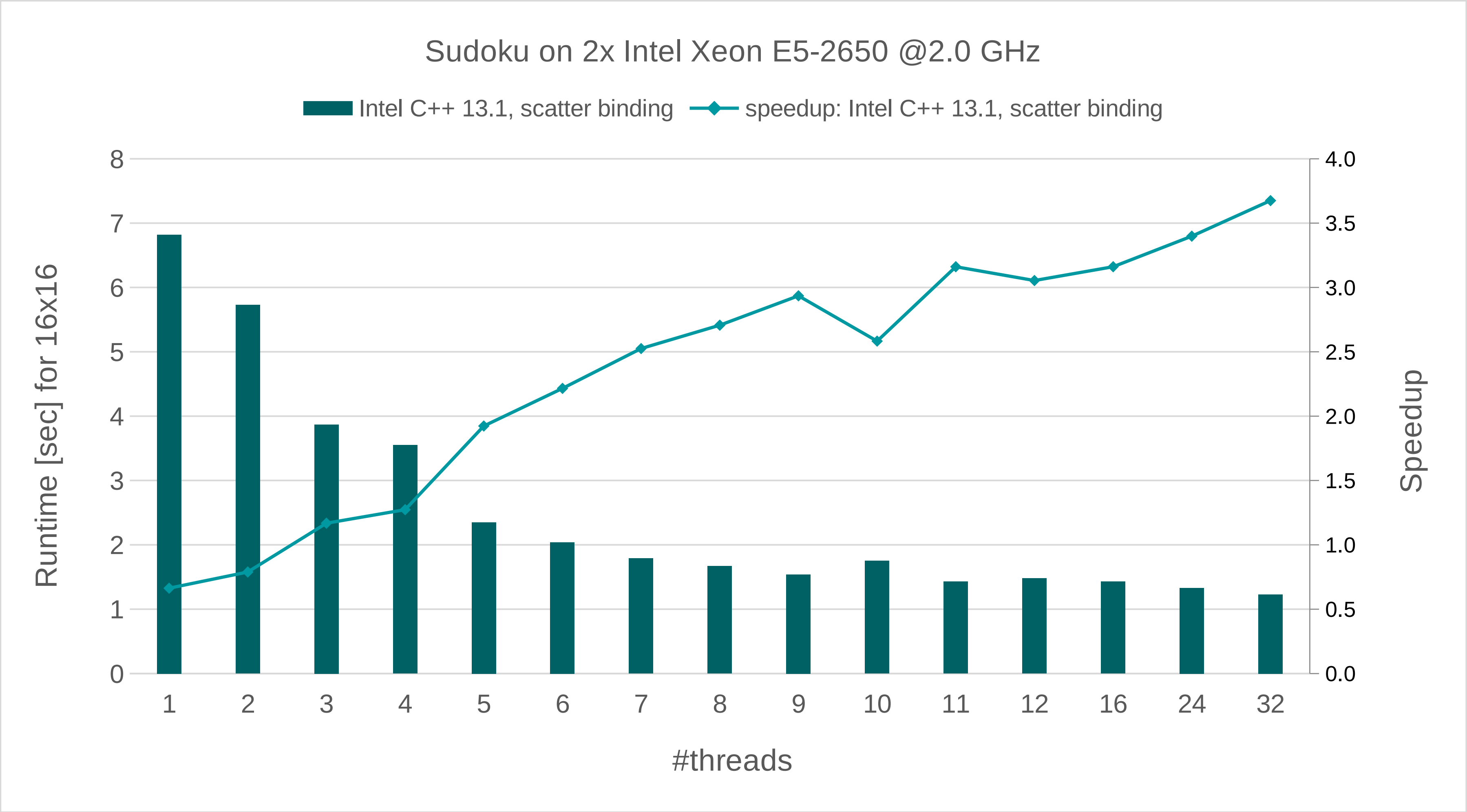



A map is introduced to store location information of data that was previously used

Jannis Klinkenberg, Philipp Samfass, Christian Terboven, Alejandro Duran, Michael Klemm, Xavier Teruel, Sergi Mateo, Stephen L. Olivier, and Matthias S. Müller. **Assessing Task-to-Data Affinity in the LLVM OpenMP Runtime**. Proceedings of the 14th International Workshop on OpenMP, IWOMP 2018. September 26-28, 2018, Barcelona, Spain.

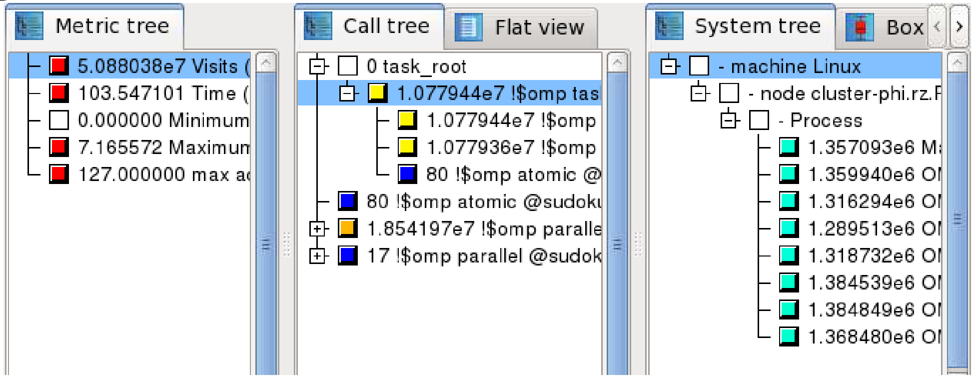
# Do you still remember the Motivation?

# Performance Evaluation

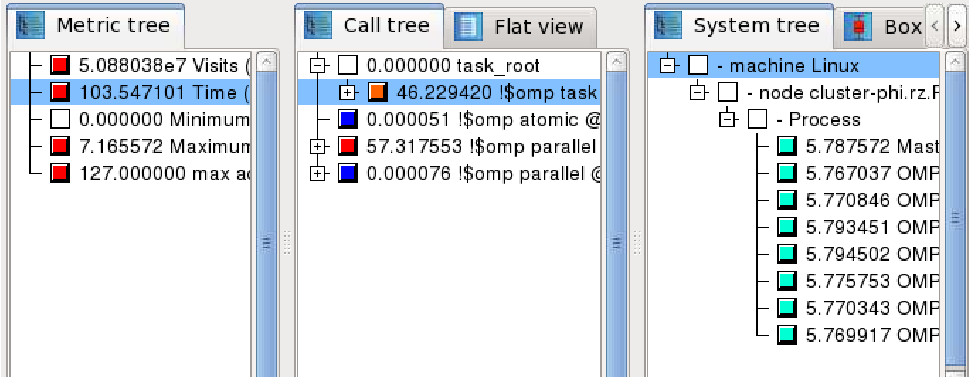


# Performance Analysis

Event-based profiling provides a good overview :




Every thread is executing ~1.3m tasks…

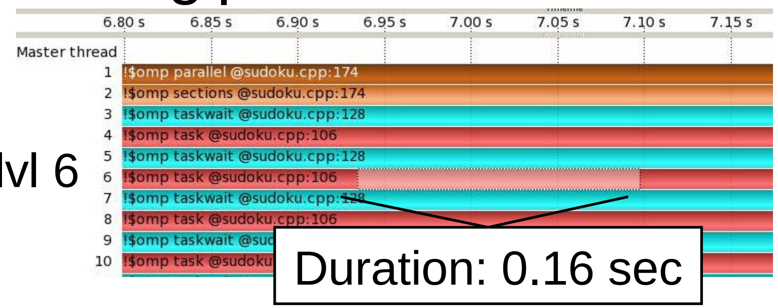



… in ~5.7 seconds.

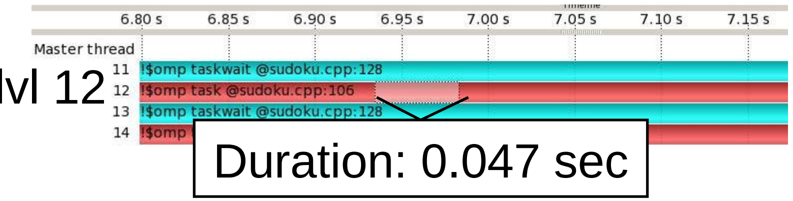
=> average duration of a task is ~4.4 µs

Tracing provides more details:

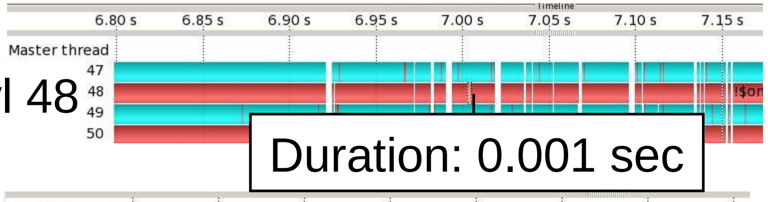
lvl 6



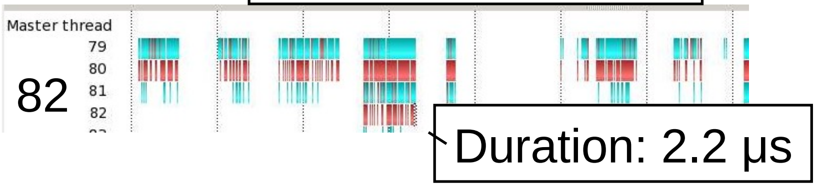

lvl 12




lvl 48




lvl 82

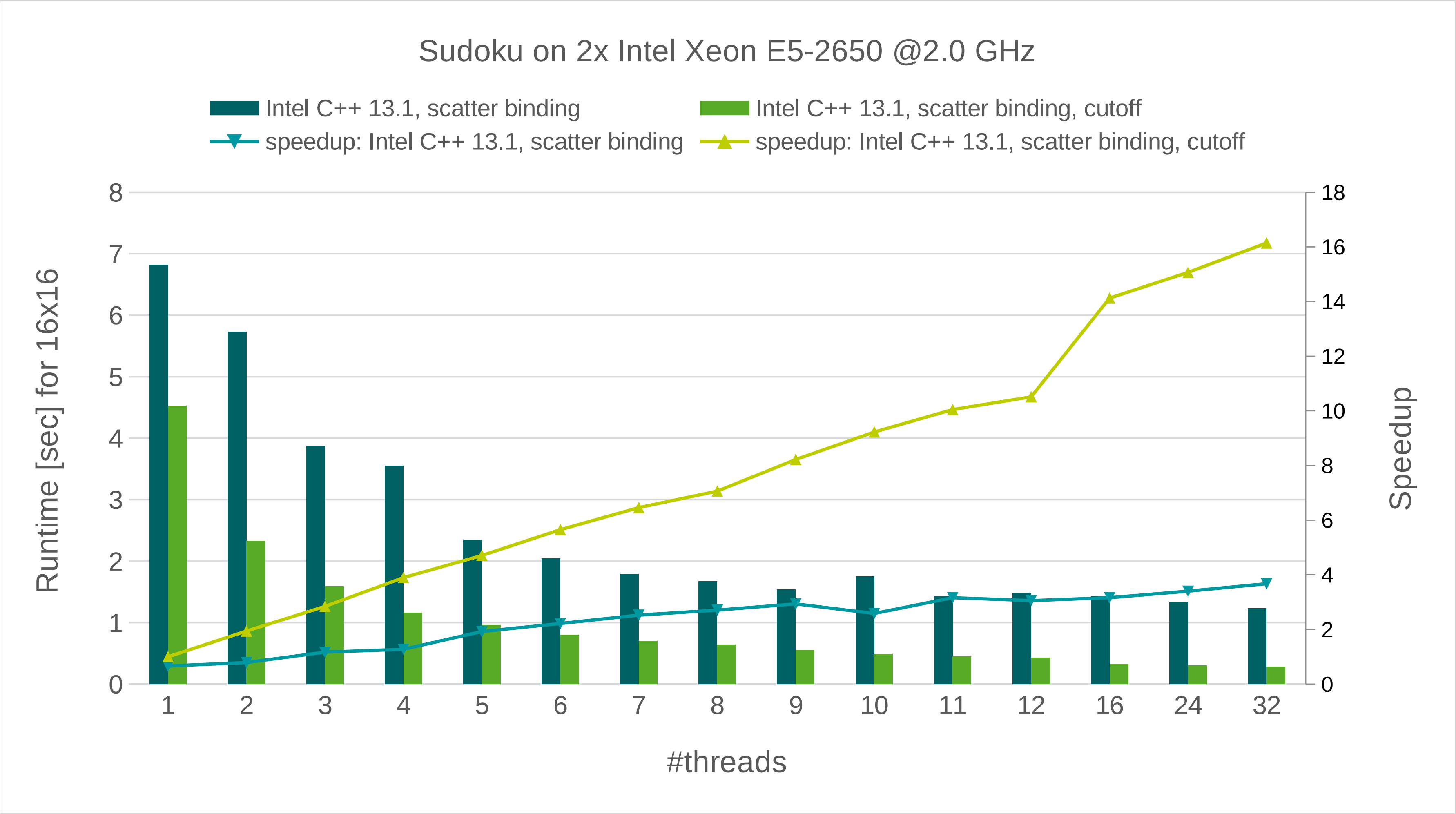



Tasks get much smaller down the call-stack.



IT Center RWTH AACHEN UNIVERSITY

# Performance Evaluation (with cutoff)



# 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?