Progamming the OpenMP API

Introduction to OpenMP Tasks

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

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…

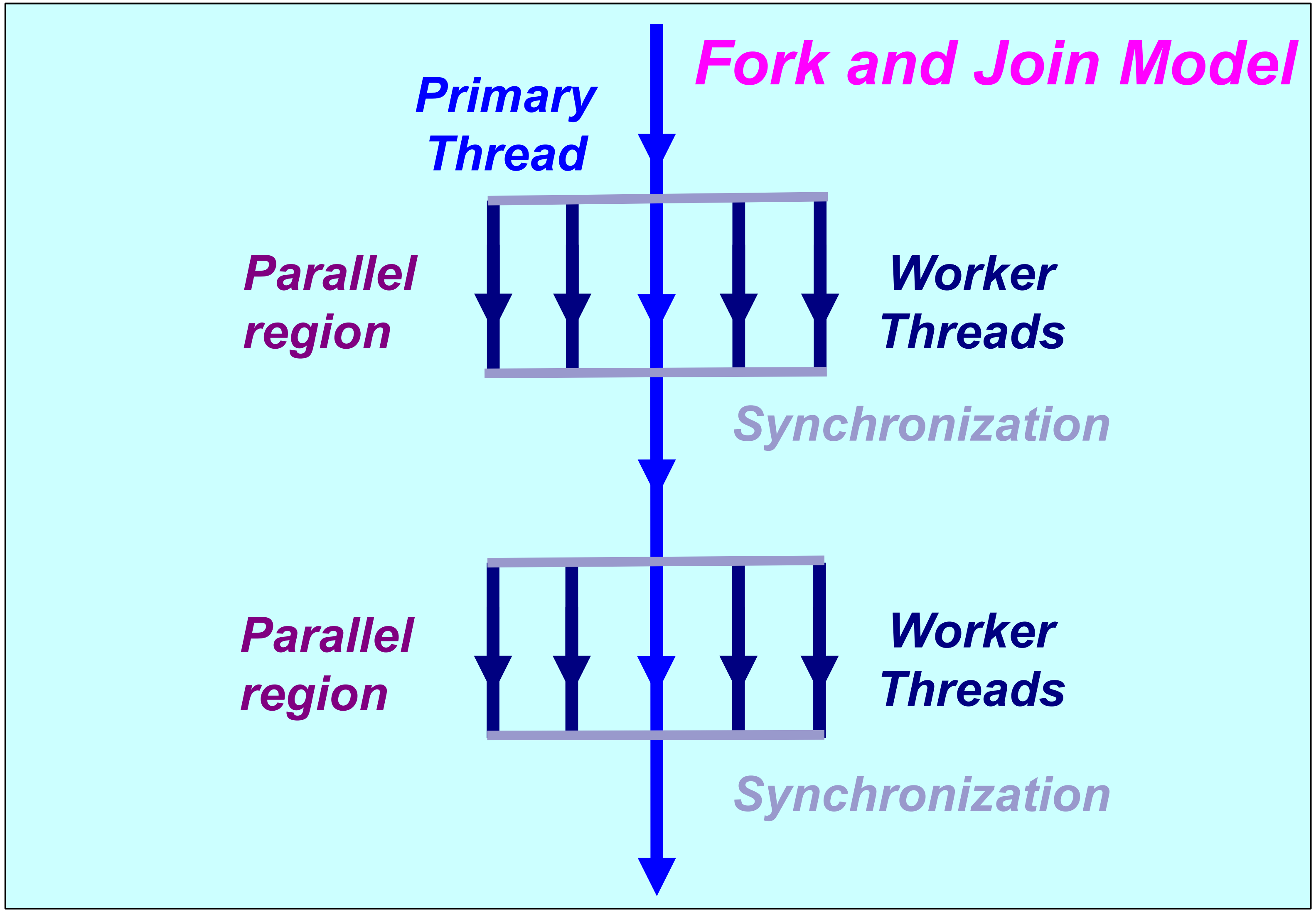
  … when reaching a parallel region → implicit tasks are created (per thread)

  … 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

The OpenMP Execution Model




```
#pragma omp parallel
{
    ....
}
```

```
#pragma omp parallel
{
    ....
}
```

The Single and Masked Directives

- Single: only one thread in the team executes the code enclosed

```
#pragma omp single [private][firstprivate] \
                   [copyprivate][nowait]
{
    <code-block>
}
```

- masked: the primary thread executes the code enclosed

```
#pragma omp masked
{<code-block>}
```

There is no implied barrier on entry or exit !

Tasking 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

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

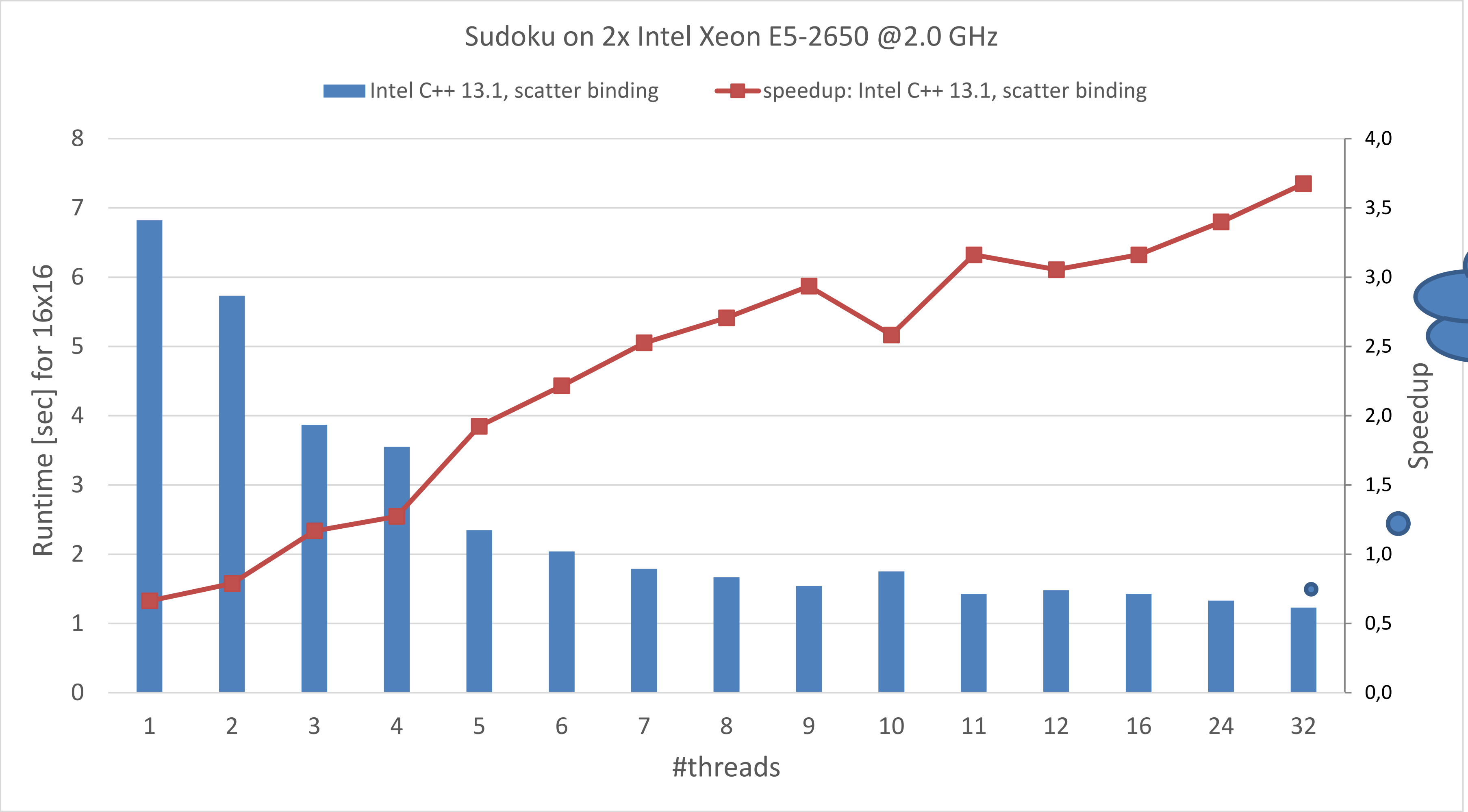
- (2) Try all numbers:
  - (2 a) Check Sudoku
    - If invalid: skip
    - If valid: Go to next field

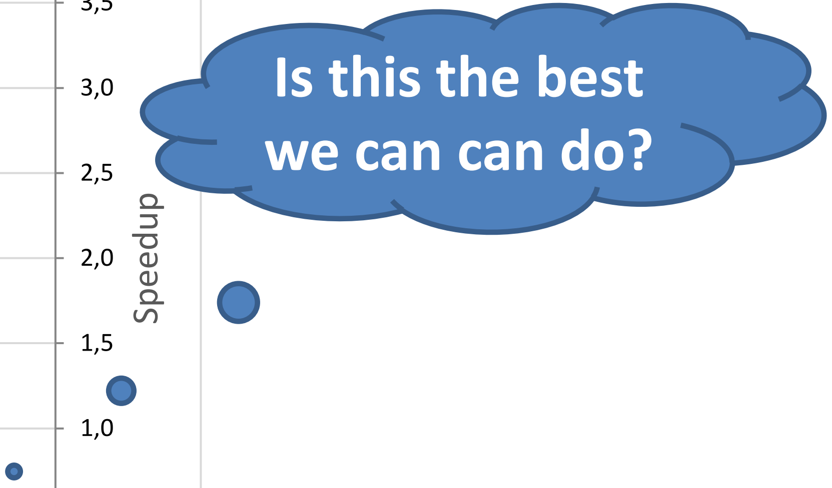
`#pragma omp task`
needs to work on a new copy of the Sudoku board

- Wait for completion

`#pragma omp taskwait`
wait for all child tasks

Performance Evaluation





Tasking execution model

- Supports unstructured parallelism
  - → unbounded loops

```
while ( <expr> ) {
    ...
}
```

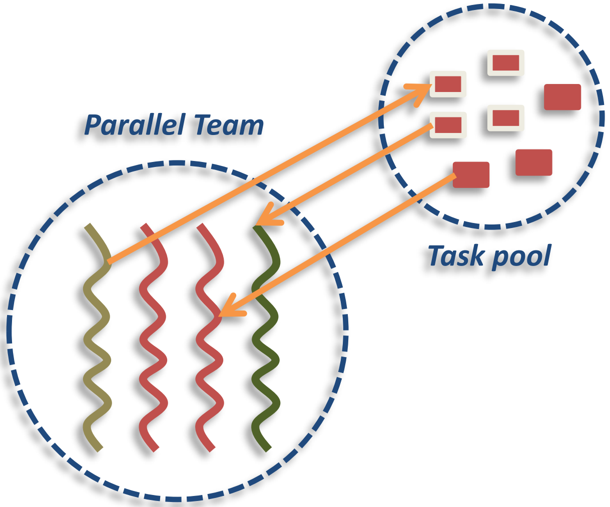
  - → recursive functions

```
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 single
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)

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

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

- Where clause is one of:

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

Task scheduling: tied vs untied tasks

- Tasks are tied by default (when no untied clause present)
  - → tied tasks are executed always by the same thread (not necessarily creator)
  - → tied tasks may run into performance problems
- Programmers may specify tasks to be untied (relax scheduling)

```
#pragma omp task untied
{structured-block}
```

  - → can potentially switch to any thread (of the team)
  - → bad mix with thread based features: thread-id, threadprivate, critical regions...
  - → gives the runtime more flexibility to schedule tasks
  - → but most of OpenMP implementations doesn’t “honor” untied ☹

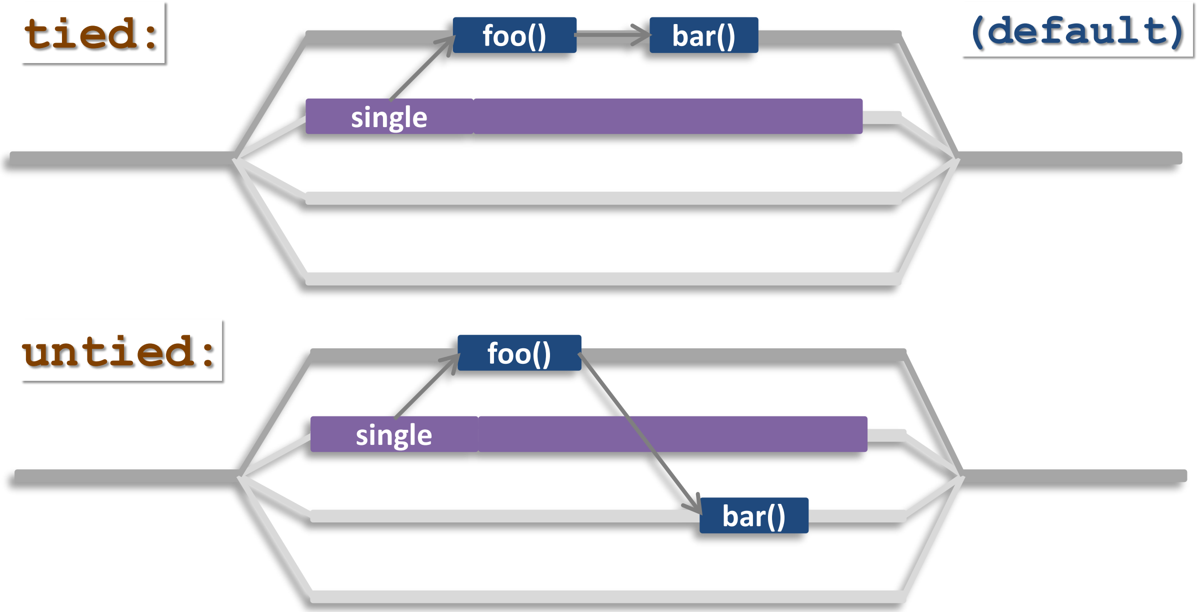
Task scheduling: taskyield directive

- 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
```

- Scheduling [tied/untied] tasks: example

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



Task scheduling: programmer's hints

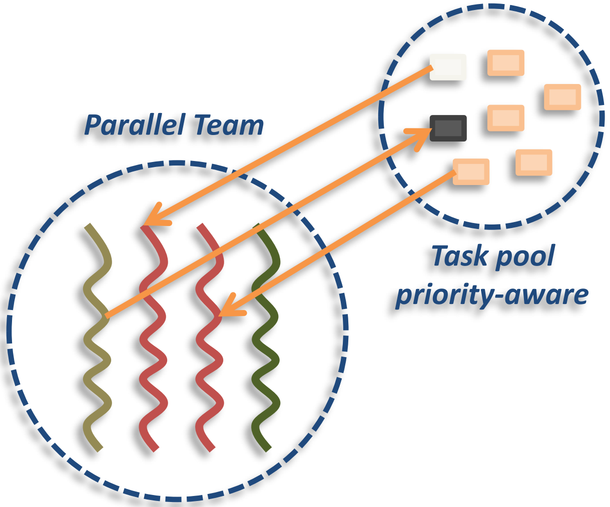
- Programmers may specify a priority value when creating a task

```
#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; }
  ...
}
```



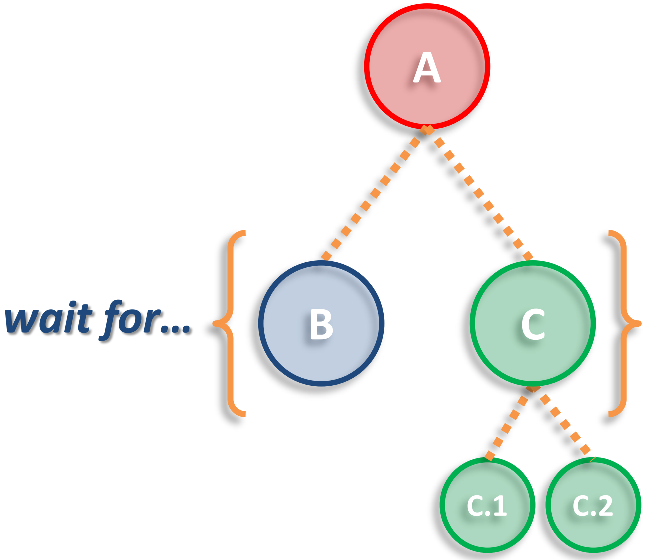
Task synchronization: taskwait directive

- 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)

```
#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: barrier semantics

- OpenMP barrier (implicit or explicit)
  - → All tasks created by any thread of the current team are guaranteed to be completed at barrier exit

```
#pragma omp barrier
```

  - → And all other implicit barriers at parallel, sections, for, single, etc…

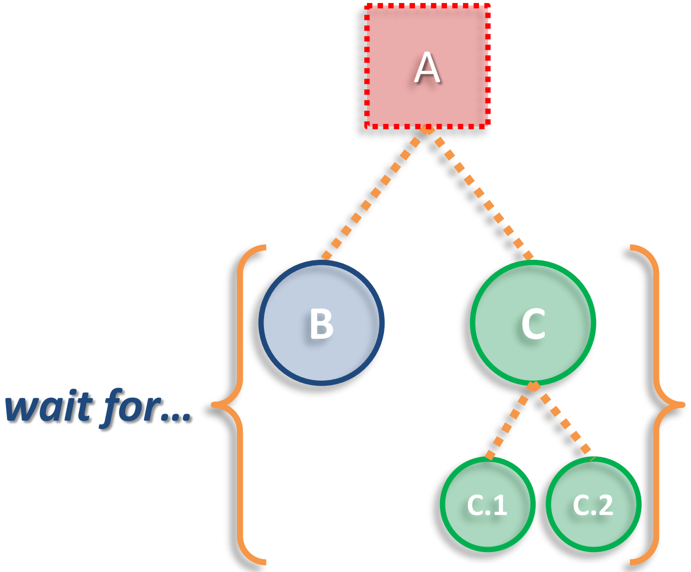
Task synchronization: taskgroup construct

- 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}
```

  - → where clause (could only be): reduction(reduction-identifier: 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
}
```



Data Environment

Explicit data-sharing clauses

- Explicit data-sharing clauses (shared, private and firstprivate)

```
#pragma omp task shared(a)
{
  // Scope of a: shared
}
```

```
#pragma omp task private(b)
{
  // Scope of b: private
}
```

```
#pragma omp task firstprivate(c)
{
  // Scope of c: firstprivate
}
```

- If **default** clause present, what the clause says
  - → shared: data which is not explicitly included in any other data sharing clause will be **shared**
  - → none: compiler will issue an error if the attribute is not explicitly set by the programmer (very useful!!!)

```
#pragma omp task default(shared)
{
 // Scope of all the references, not explicitly
 // included in any other data sharing clause,
 // and with no pre-determined attribute: shared
}
```

```
#pragma omp task default(none)
{
 // Compiler will force to specify the scope for
 // every single variable referenced in the context
}
```

Hint: Use default(none) to be forced to think about every variable if you do not see clearly.

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)…

5
```
#pragma omp task
{
   int x = MN;
   // Scope of x: private
}
```

4
```
#pragma omp task
{
   static int y;
   // Scope of y: shared
}
```

1
```
int A[SIZE];
#pragma omp threadprivate(A)

// ...
#pragma omp task
{
  // A: threadprivate
}
```

2
```
int *p;

p = malloc(sizeof(float)*SIZE);

#pragma omp task
{
   // *p: shared
}
```

3
```
void foo(void){
   static int s = MN;
}

#pragma omp task
{
   foo(); // s@foo(): shared
}
```

Implicit data-sharing attributes (in-practice)

- Implicit data-sharing rules for the task region
  - → the **shared** attribute is lexically inherited
  - → in any other case the variable is **firstprivate**

  - → Pre-determined rules (can not change)
  - → Explicit data-sharing clauses (+ default)
  - → Implicit data-sharing rules

```
int a = 1;
void foo() {
   int b = 2, c = 3;
   #pragma omp parallel private(b)
   {
      int d = 4;
      #pragma omp task
      {
         int e = 5;
         // Scope of a:
         // Scope of b:
         // Scope of c:
         // Scope of d:
         // Scope of e:
      }
   }
}
```

- (in-practice) variable values within the task:
  - → value of a: 1
  - → value of b: x // undefined (undefined in parallel)
  - → value of c: 3
  - → value of d: 4
  - → value of e: 5

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}
```

  - → Register a new reduction at [1]
  - → Computes the final result after [3]
- The (task) in_reduction clause [participating]

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

  - → 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]
```

Improving Tasking Performance: Cutoff clauses and strategies

Example: Sudoku revisited

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

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

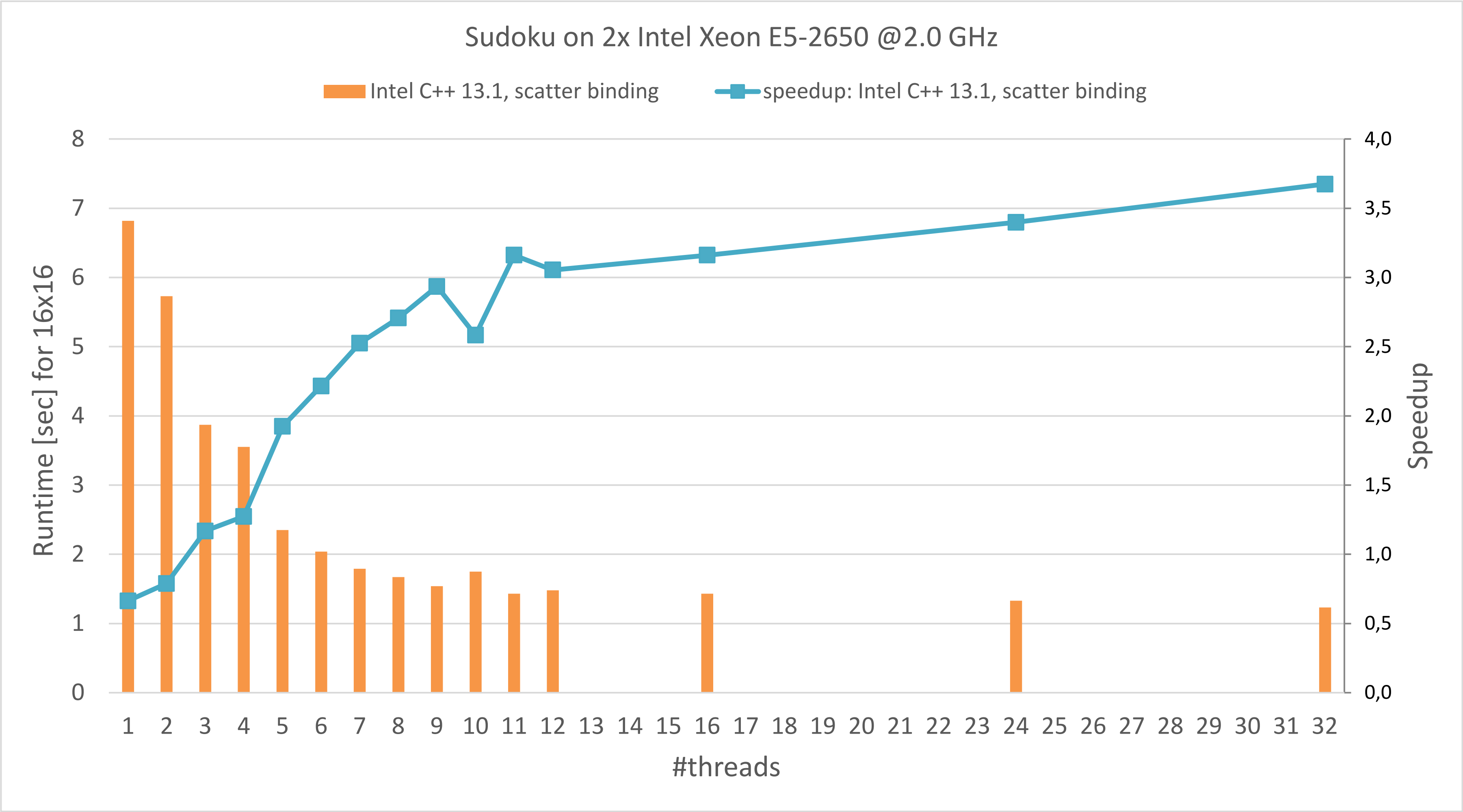
- (2) Try all numbers:
  - (2 a) Check Sudoku
    - If invalid: skip
    - If valid: Go to next field

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

- Wait for completion

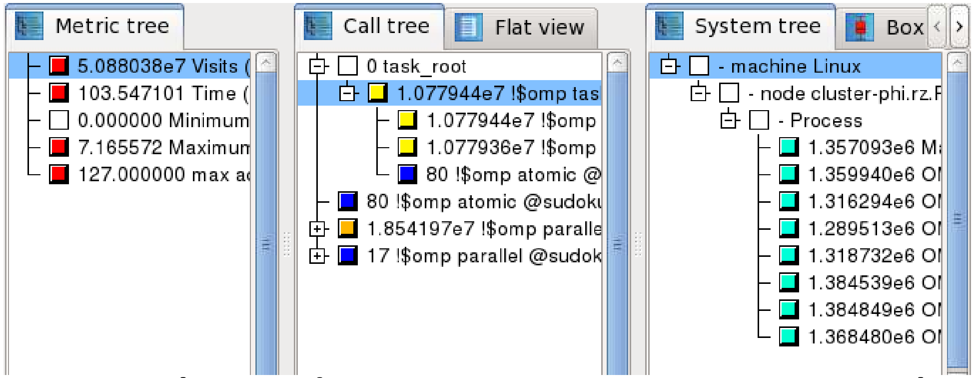
`#pragma omp taskwait`
wait for all child tasks

Performance Evaluation

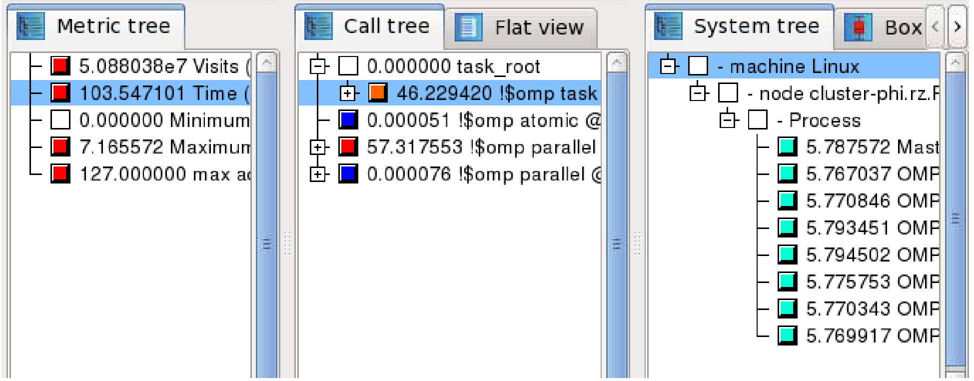


Performance Analysis

Event-based profiling provides a good overview :



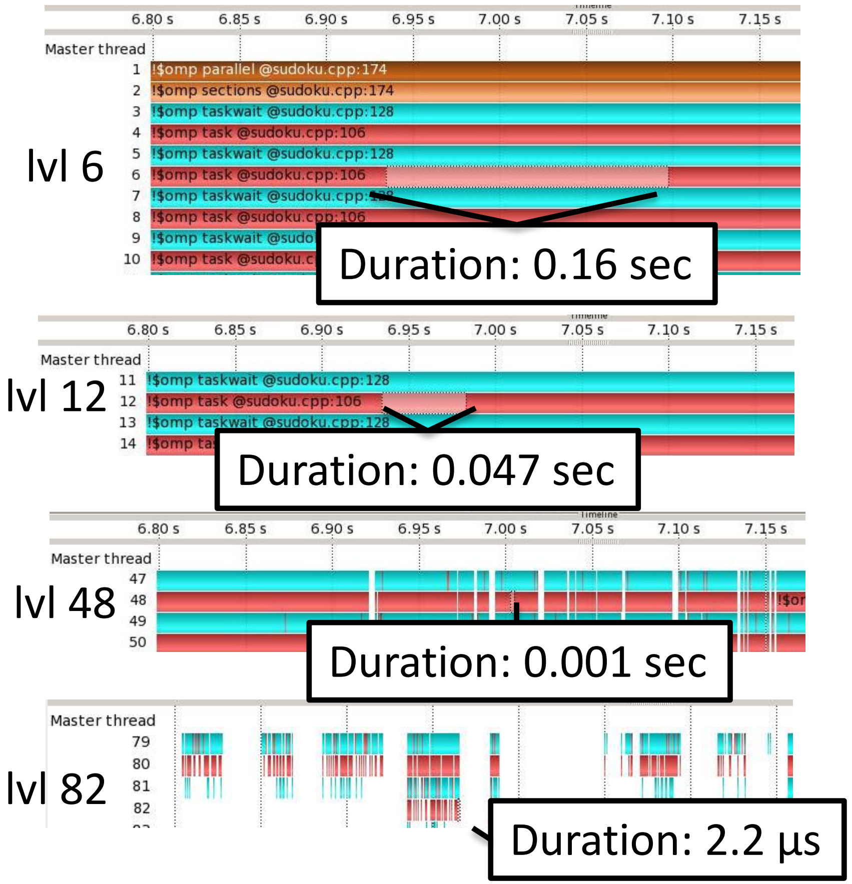
Every thread is executing ~1.3m tasks…



… in ~5.7 seconds.

=> average duration of a task is ~4.4 µs

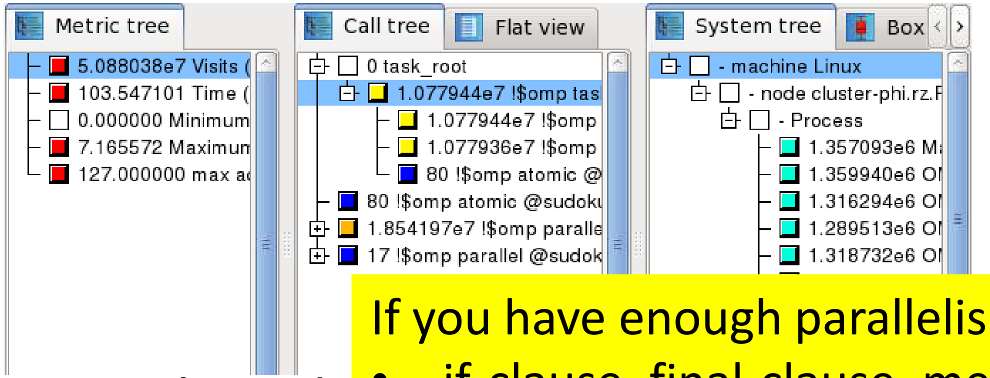
Tracing provides more details:



Tasks get much smaller down the call-stack.

Performance Analysis

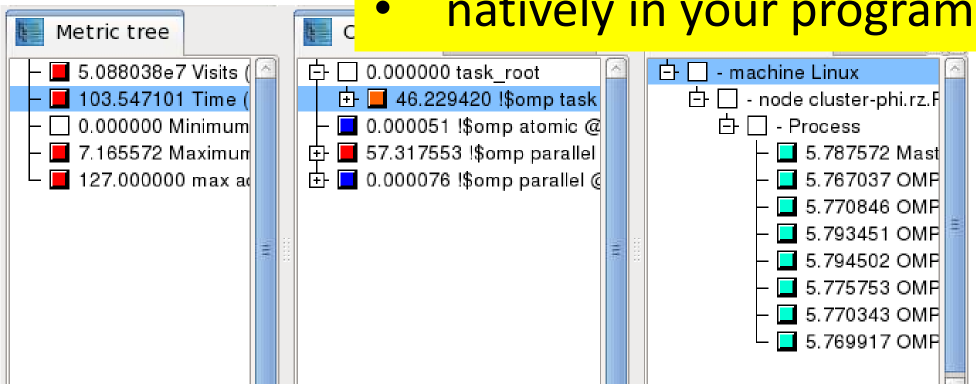
Event-based profiling provides a good overview :



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

- if-clause, final-clause, mergeable-clause
- natively in your program code

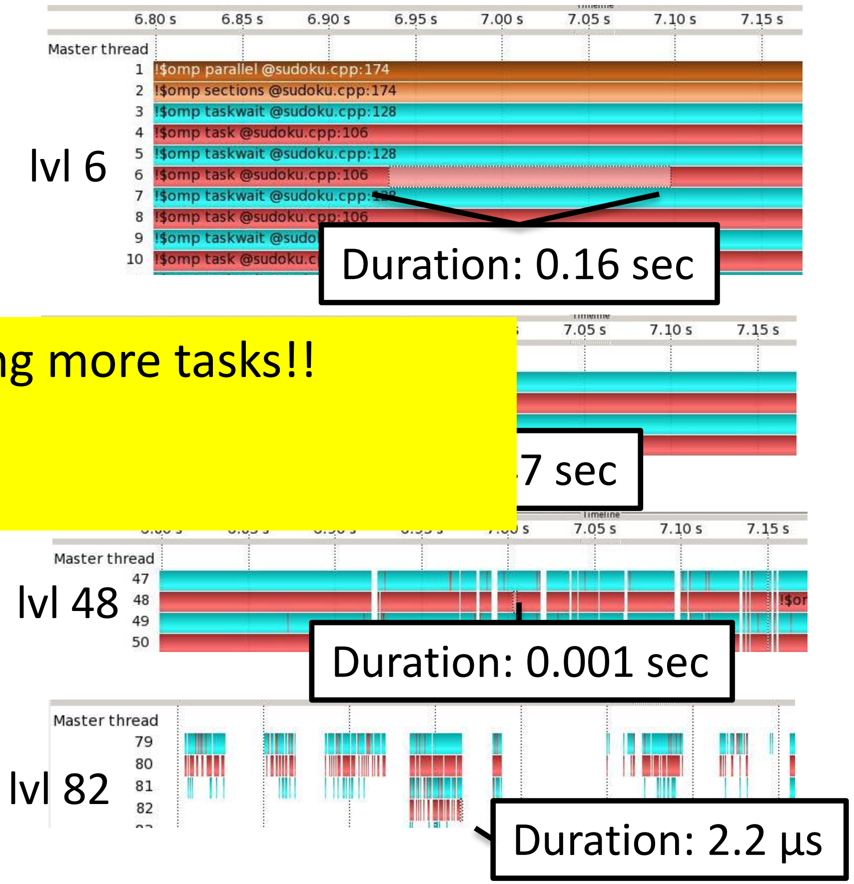
Every thread i



… in ~5.7 seconds.

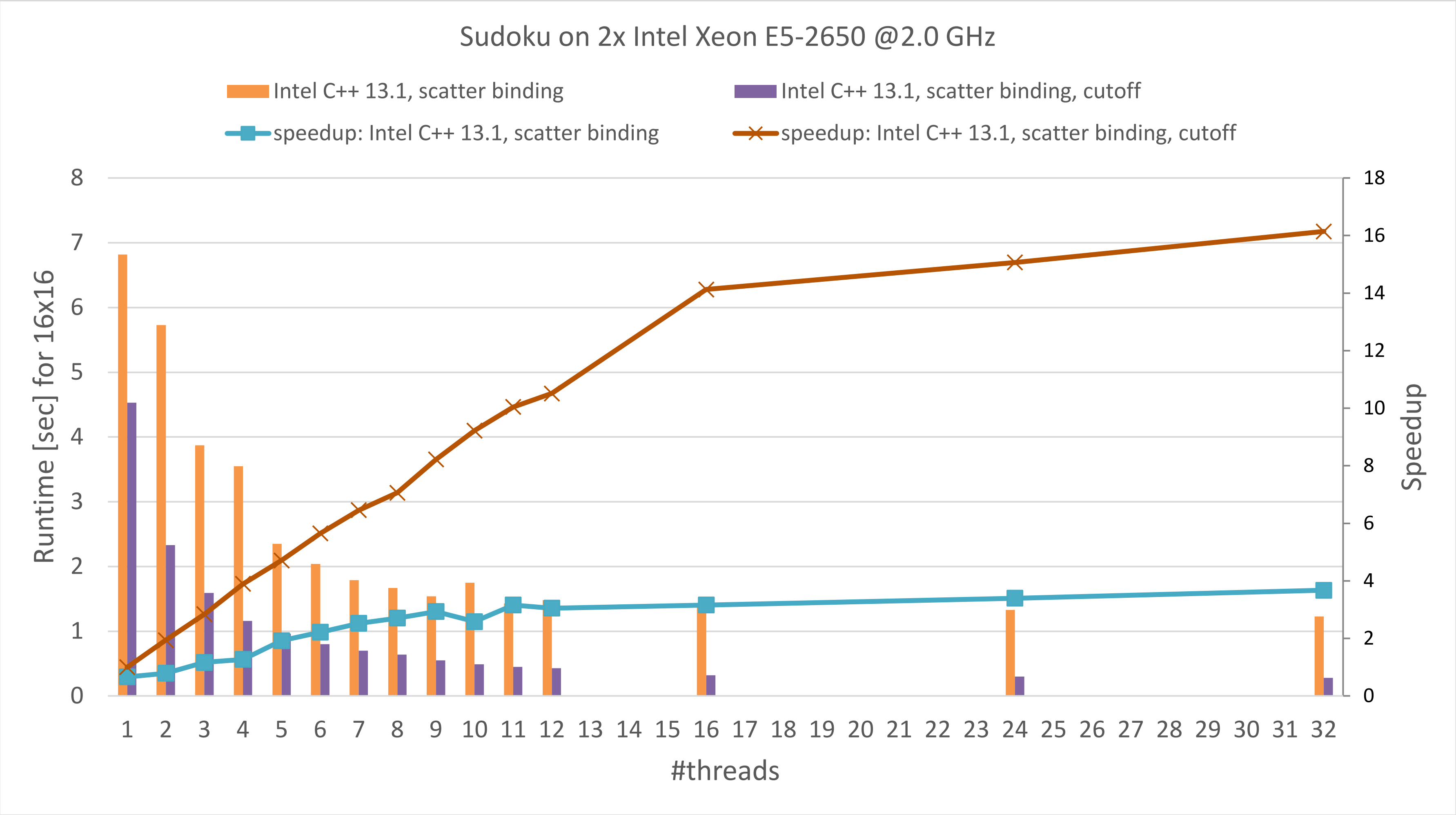
=> average duration of a task is ~4.4 µs

Tracing provides more details:



Tasks get much smaller down the call-stack.

Performance Evaluation (with cutoff)



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*

```
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”);
}
```

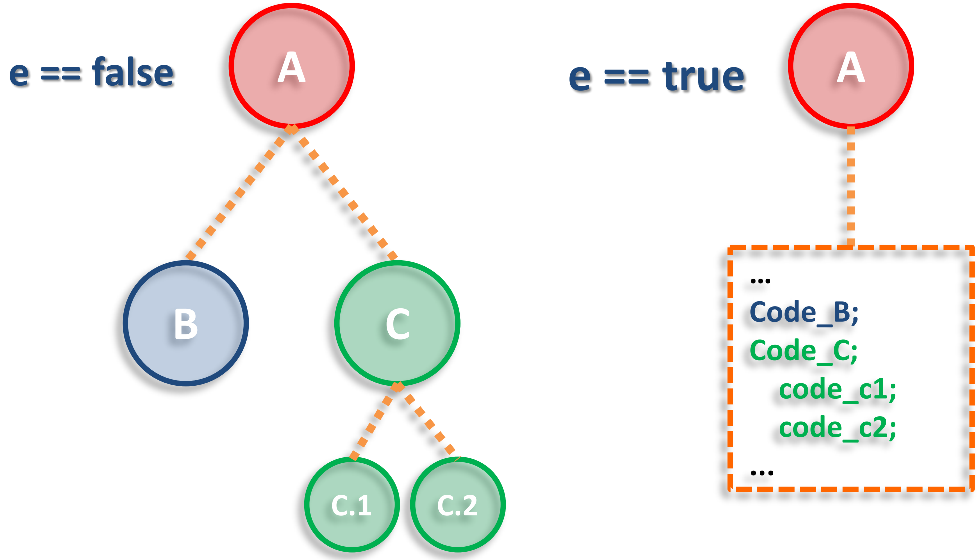
Really useful to debug tasking applications!

- 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*)

```
#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

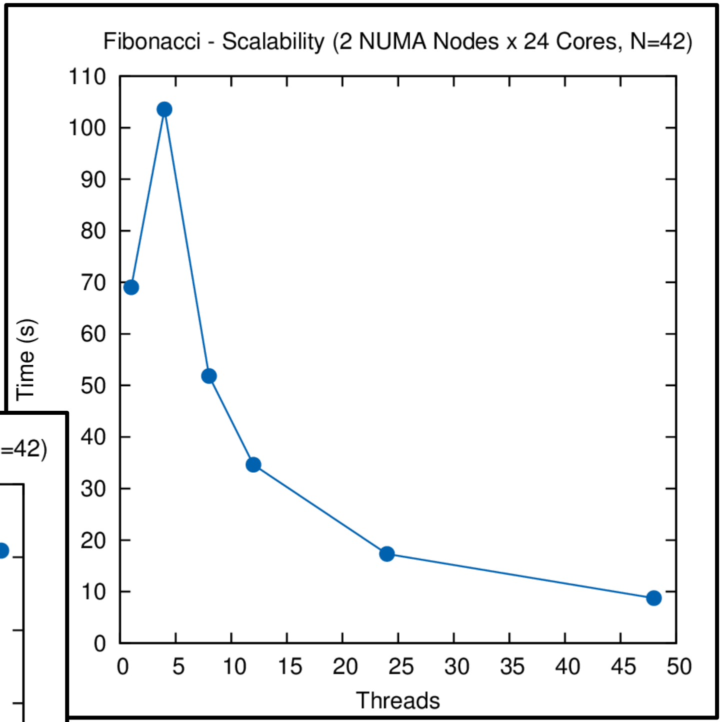
- The `mergeable` clause
  - → Optimization: get rid of “data-sharing clauses are honored”
  - → This optimization can only be applied in *undeferred* or *included tasks*
- A Task that is annotated with the `mergeable` clause is called a *mergeable task*
  - → A task that may be a *merged task* if it 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

Unfortunately, there are no OpenMP commercial implementations taking advantatge of `final` neither `mergeable` =(

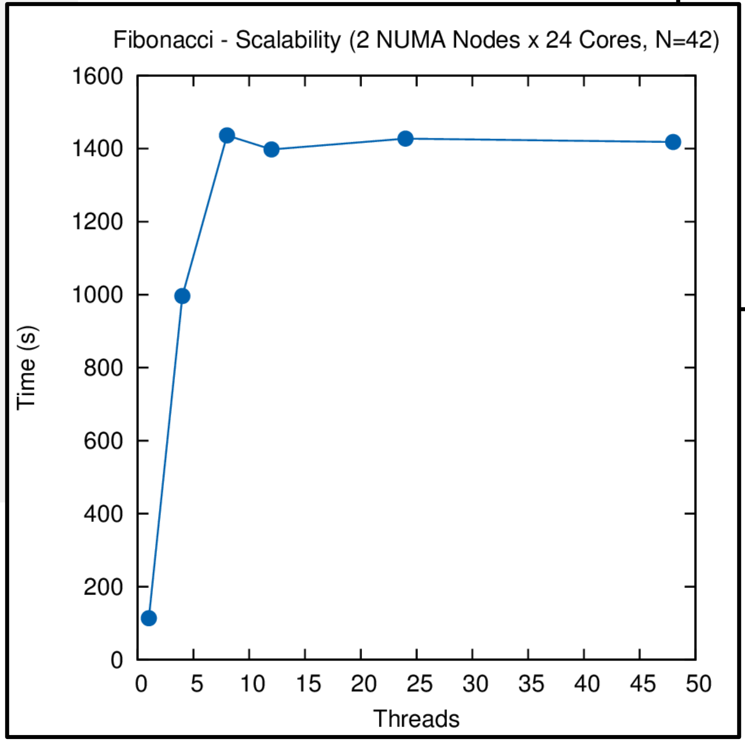
Example: Fibonacci

Fibonacci: without cutoff

```
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;
}
```



icc 2018.0

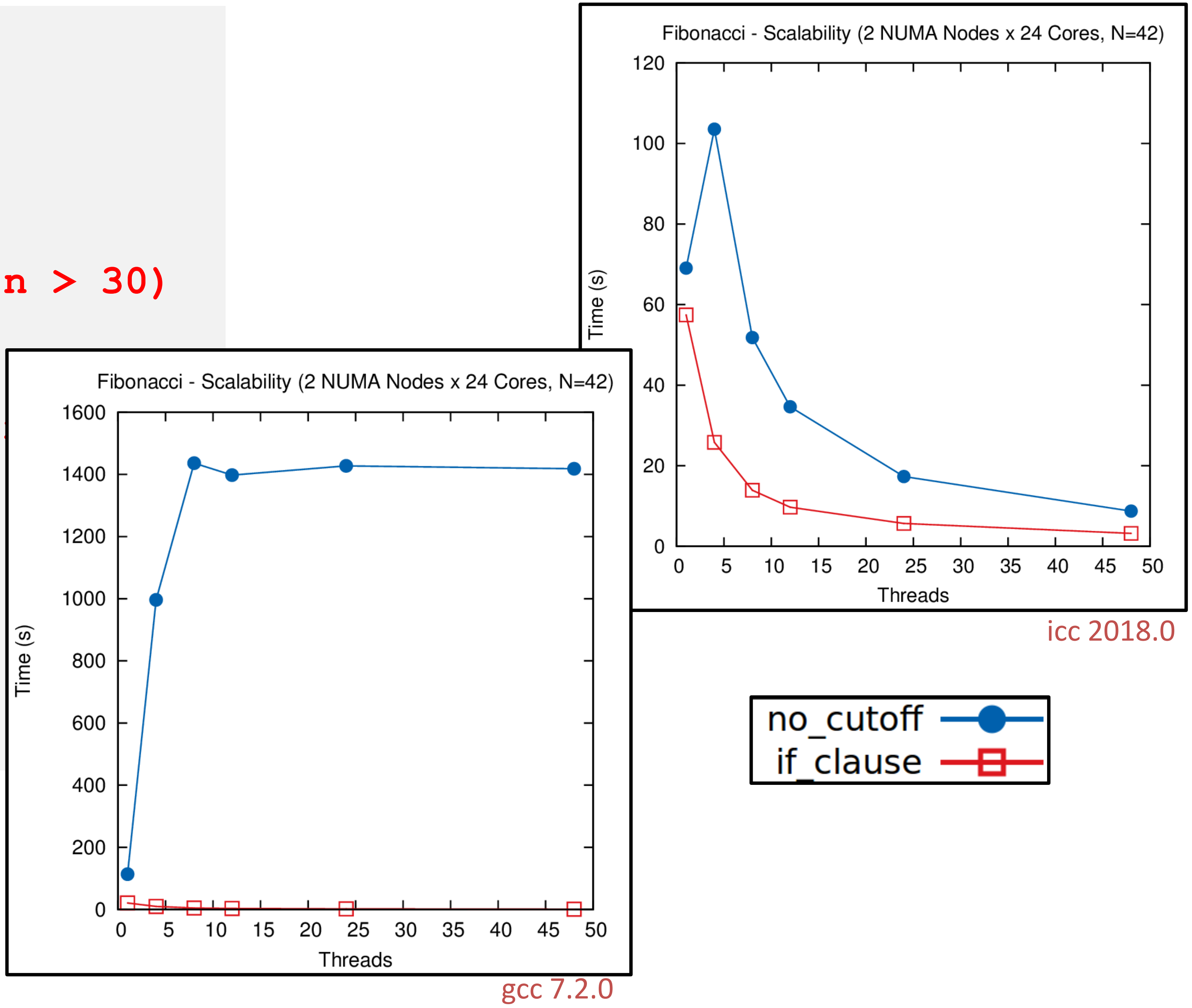


gcc 7.2.0

no_cutoff

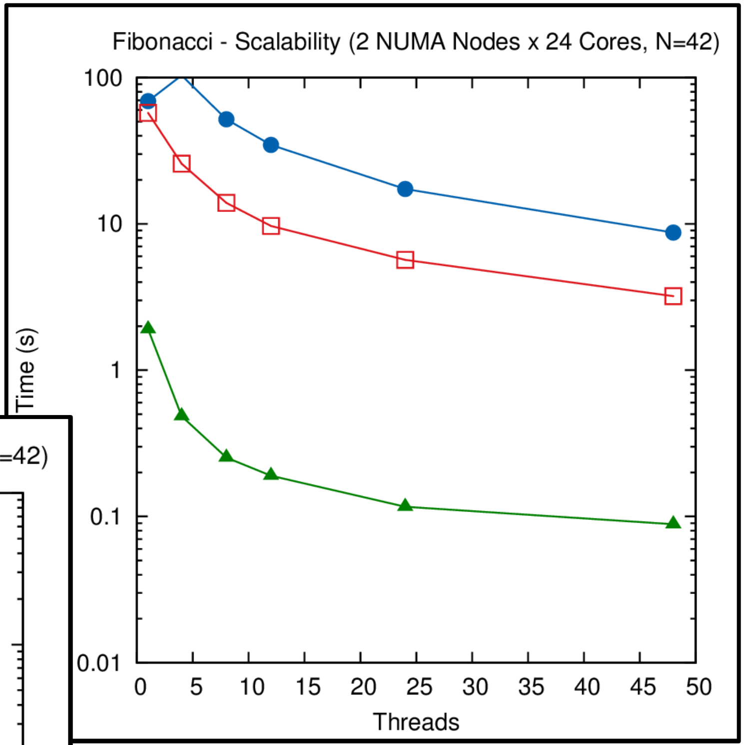
Fibonacci: `if` clause

```
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(
  res2 = fib(n-2);

  #pragma omp taskwait

  return res1 + res2;
}
```

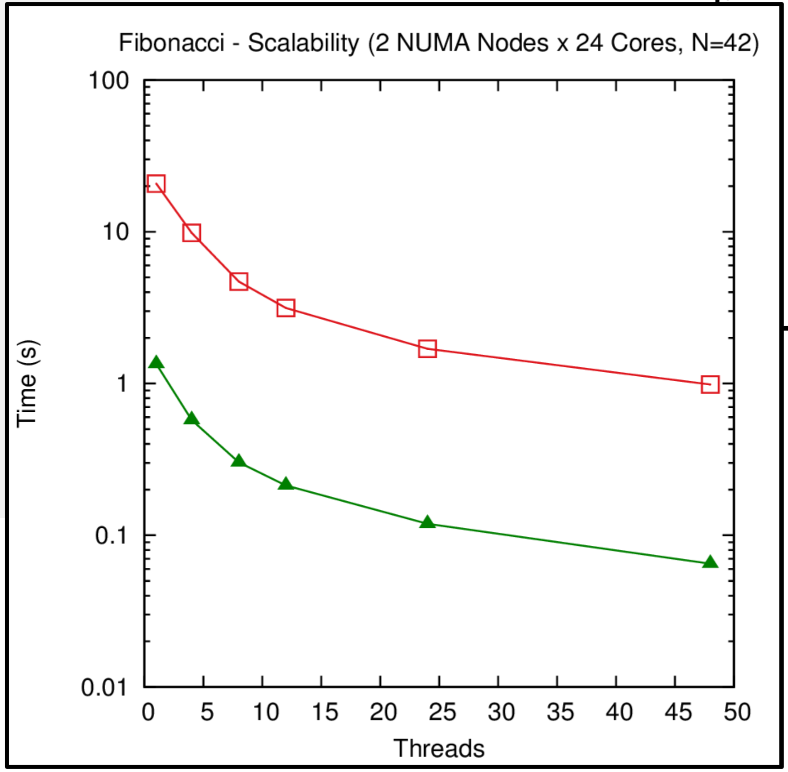


Fibonacci: manual optimization

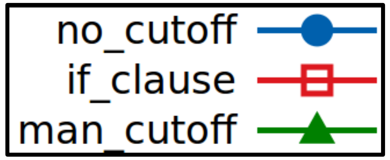
```
int fib(int n) {
  if (n < 30)
    return fib_serial(n);

  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;
}
```



icc 2018.0



gcc 7.2.0



Improving Tasking Performance:

Task Affinity (OpenMP 5.0 feature)

Motivation

- Techniques for process binding & thread pinning available
  - → OpenMP thread level: `OMP_PLACES & OMP_PROC_BIND`
  - → OS functionality: `taskset -c`

OpenMP Tasking:

- In general: Tasks may be executed by any thread in the team
  - → Missing task-to-data affinity may have detrimental effect on performance

OpenMP 5.0:

- `affinity` clause to express affinity to data

`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

Code Example

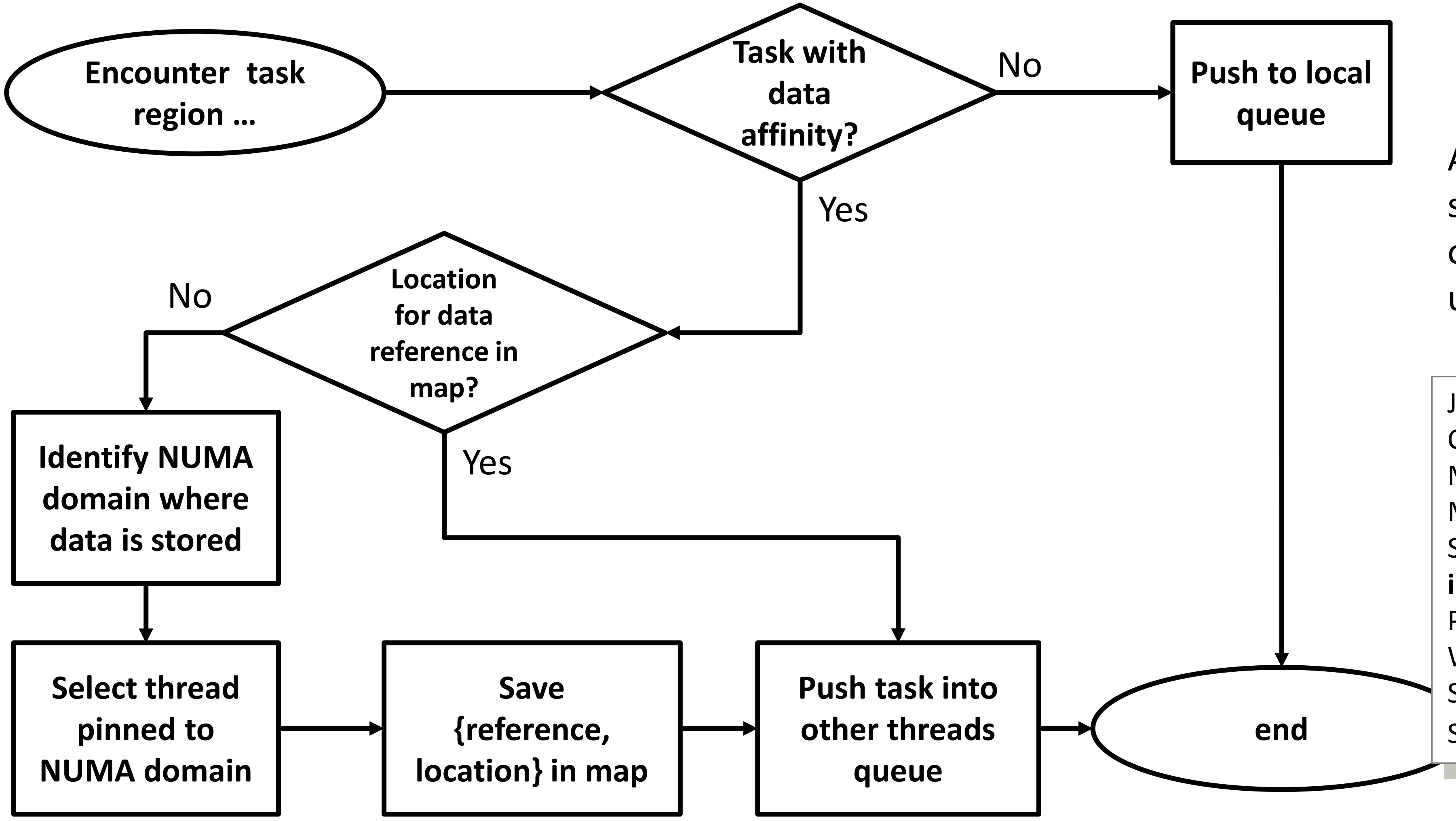
- Excerpt from task-parallel STREAM

```
#pragma omp task \
    shared(a, b, c, scalar) \
    firstprivate(tmp_idx_start, tmp_idx_end) \
    affinity( a[tmp_idx_start] )
{
   int i;
   for(i = tmp_idx_start; i <= tmp_idx_end; i++)
       a[i] = b[i] + scalar * c[i];
}
```

→Loops have been blocked manually (see `tmp_idx_start/end`)

→Assumption: initialization and computation have same blocking and same affinity

Selected LLVM implementation details

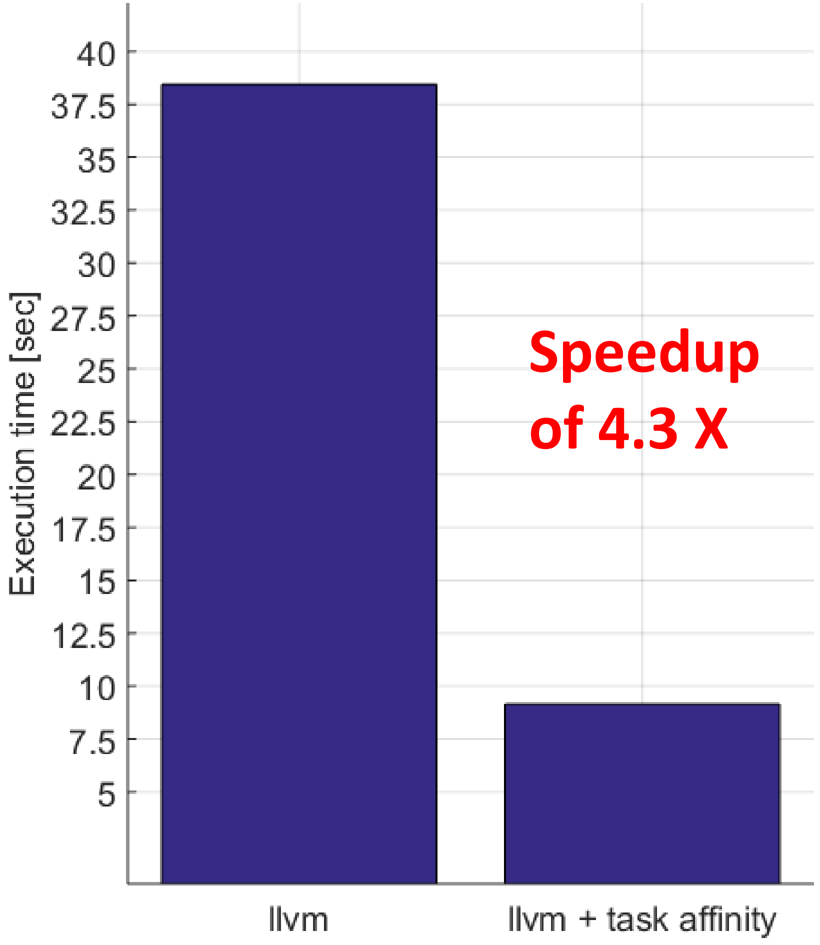


A map is introduced to store location information of data that was previously used

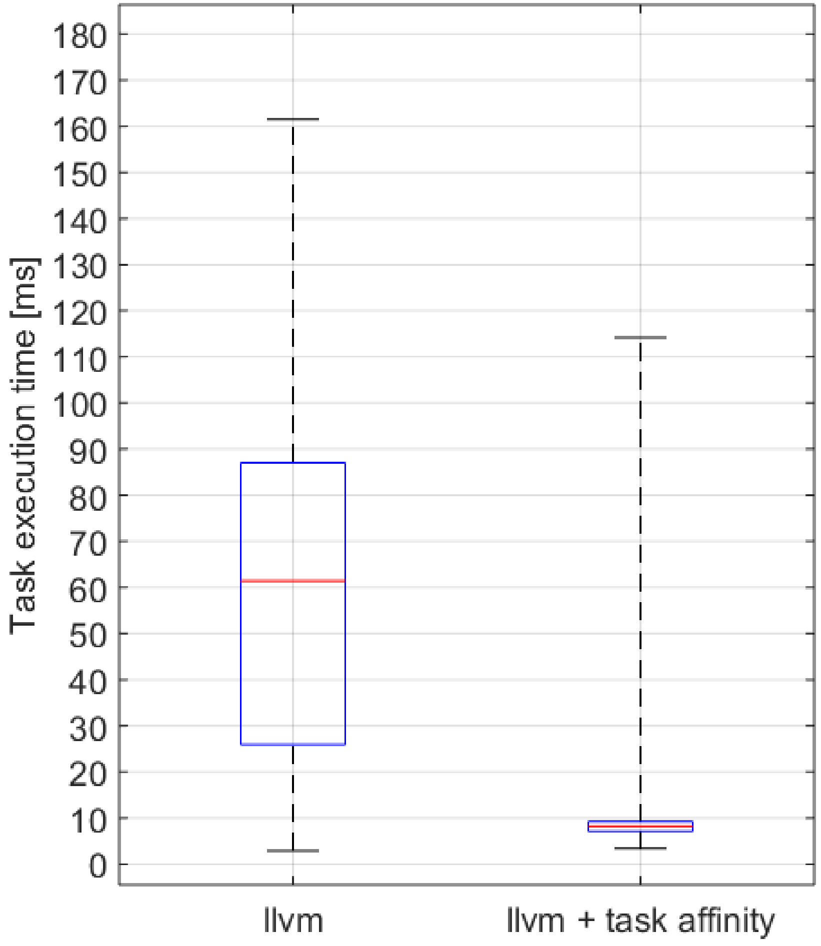
Jannis Klinkenberg, Philipp Samfass, Christian Terboven, Alejandro Duran, Michael Klemm, Xavier Teruel, Sergi Mateo, Stephen L. Olivier, and Matthias S. Müller. **Assessing Task-to-Data Affinity in the LLVM OpenMP Runtime**. Proceedings of the 14th International Workshop on OpenMP, IWOMP 2018. September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

Evaluation (Merge-Sort, from paper above)

Program runtime
Median of 10 runs



Distribution of single task execution times



LIKWID: reduction of remote data volume from 69% to 13%