More on OpenMP Tasking

Christian Terboven

IT Center, RWTH Aachen University

terboven@itc.rwth-aachen.de

Sudoku

Parallel Brute-force Sudoku

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

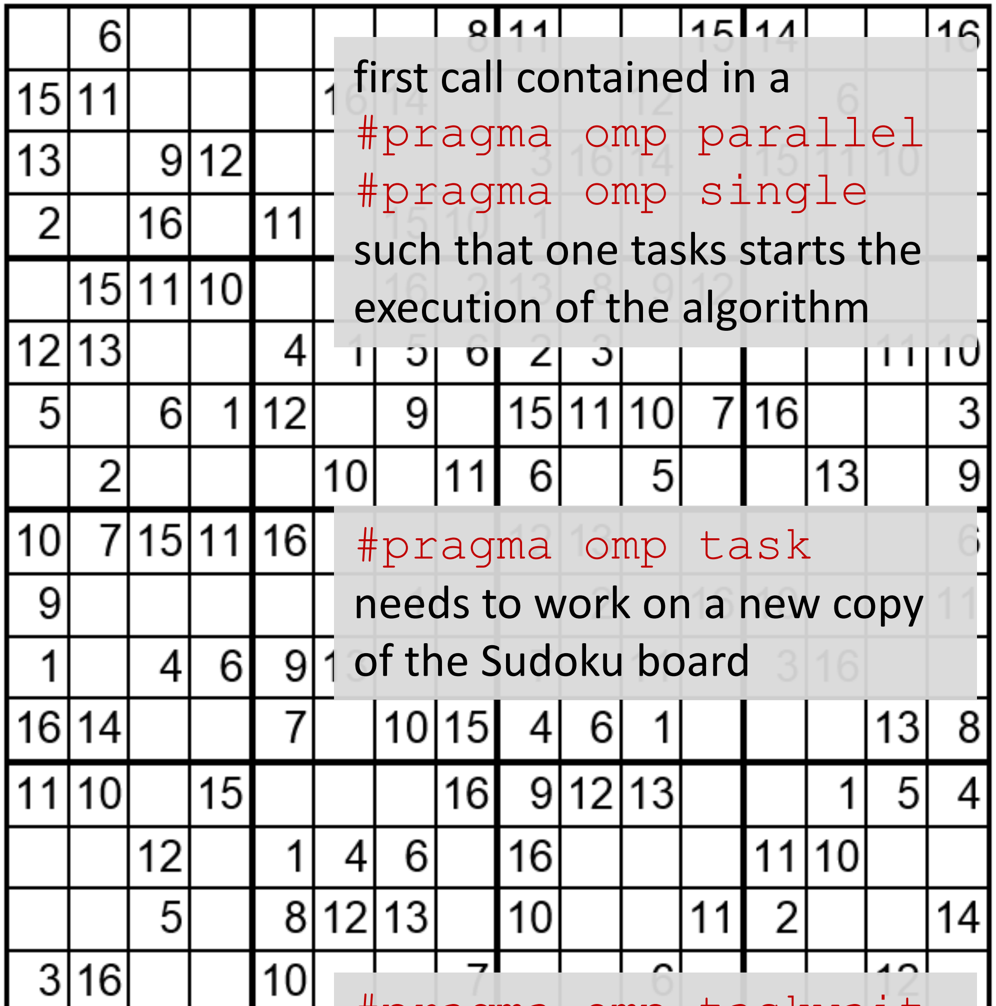
	6						8	11			15	14			16
15	11				16	14				12			6		
13		9	12					3	16	14		15	11	10	
2		16		11		15	10	1							
	15	11	10			16	2	13	8	9	12				
12	13			4	1	5	6	2	3					11	10
5		6	1	12		9		15	11	10	7	16			3
	2				10		11	6		5			13		9
10	7	15	11	16				12	13						6
9						1			2		16	10			11
1		4	6	9	13			7	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

C. Terboven| IT Center der RWTH Aachen University

Parallel Brute-force Sudoku

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



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

- (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

- **OpenMP parallel region creates a team of threads**

```
#pragma omp parallel
{
#pragma omp single
   solve_parallel(0, 0, sudoku2,false);
} // end omp parallel
```

→ Single construct: One thread enters the execution of `solve_parallel`

→ the other threads wait at the end of the `single` …

→ … and are ready to pick up threads „from the work queue“

- **Syntactic sugar (either you like it or you don‘t)**

```
#pragma omp parallel sections
{
   solve_parallel(0, 0, sudoku2,false);
} // end omp parallel
```

Parallel Brute-force Sudoku

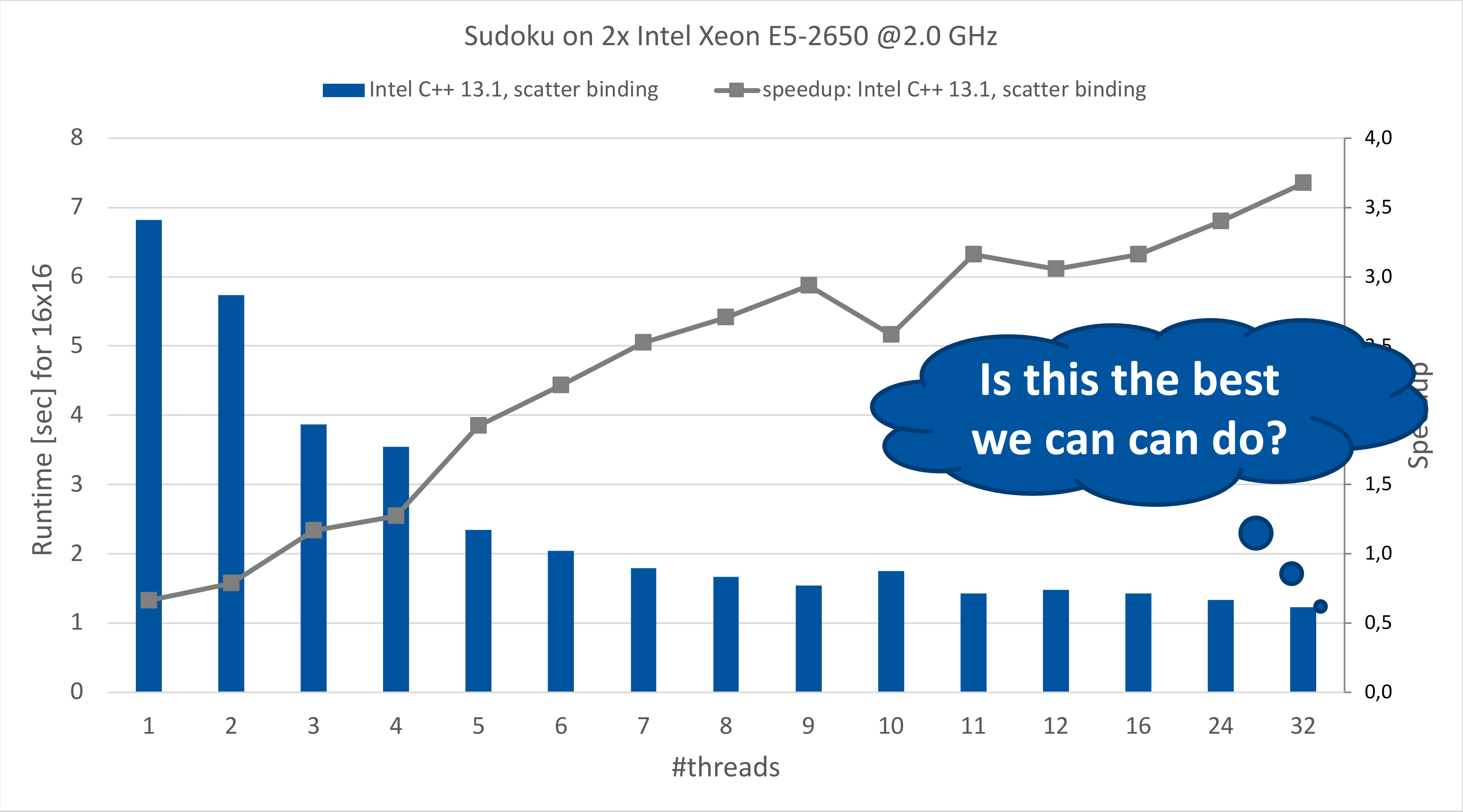
- **The actual implementation**

```
for (int i = 1; i <= sudoku->getFieldSize(); i++) {
   if (!sudoku->check(x, y, i)) {
#pragma omp task firstprivate(i,x,y,sudoku)
{
      // create from copy constructor
      CSudokuBoard new_sudoku(*sudoku);
      new_sudoku.set(y, x, i);
      if (solve_parallel(x+1, y, &new_sudoku)) {
         new_sudoku.printBoard();
      }
} // end omp task
   }
}

#pragma omp taskwait
```

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

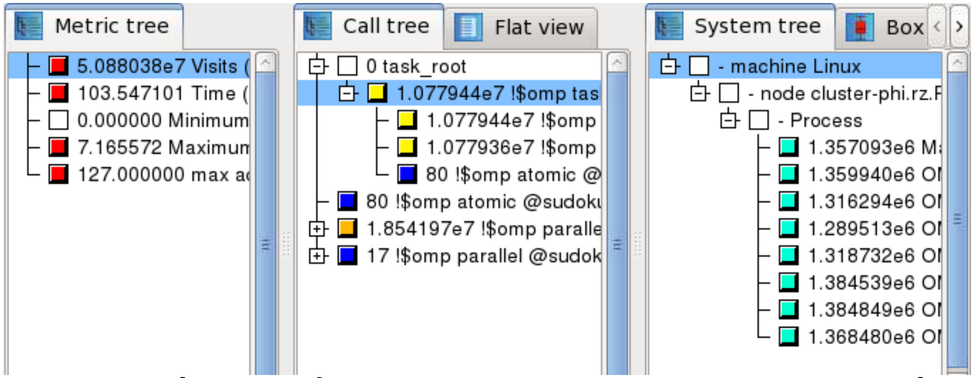
`#pragma omp taskwait`
wait for all child tasks

Performance Evaluation

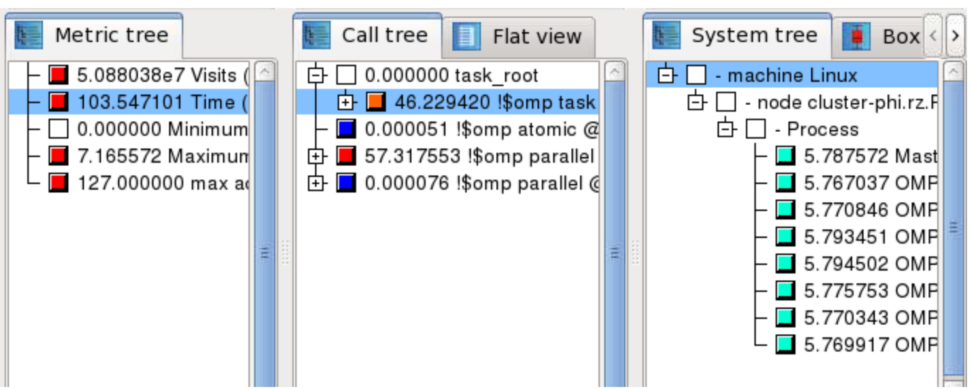


Performance Analysis

Event-based profiling gives a good overview :



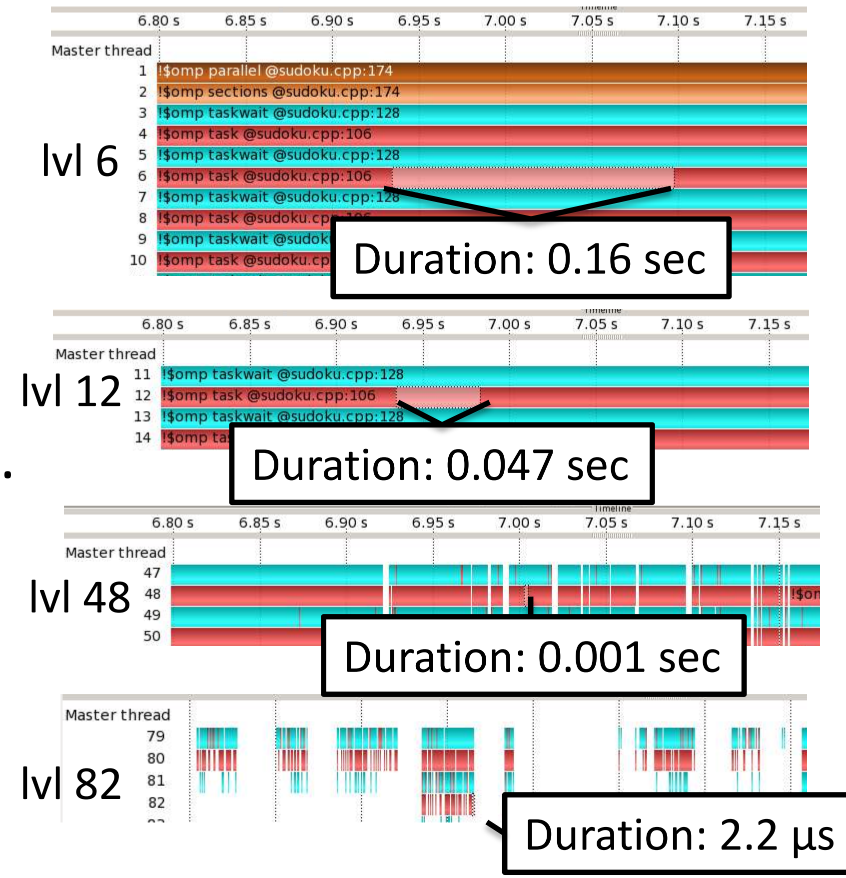
Every thread is executing ~1.3m tasks…



… in ~5.7 seconds.

=> average duration of a task is ~4.4 μs

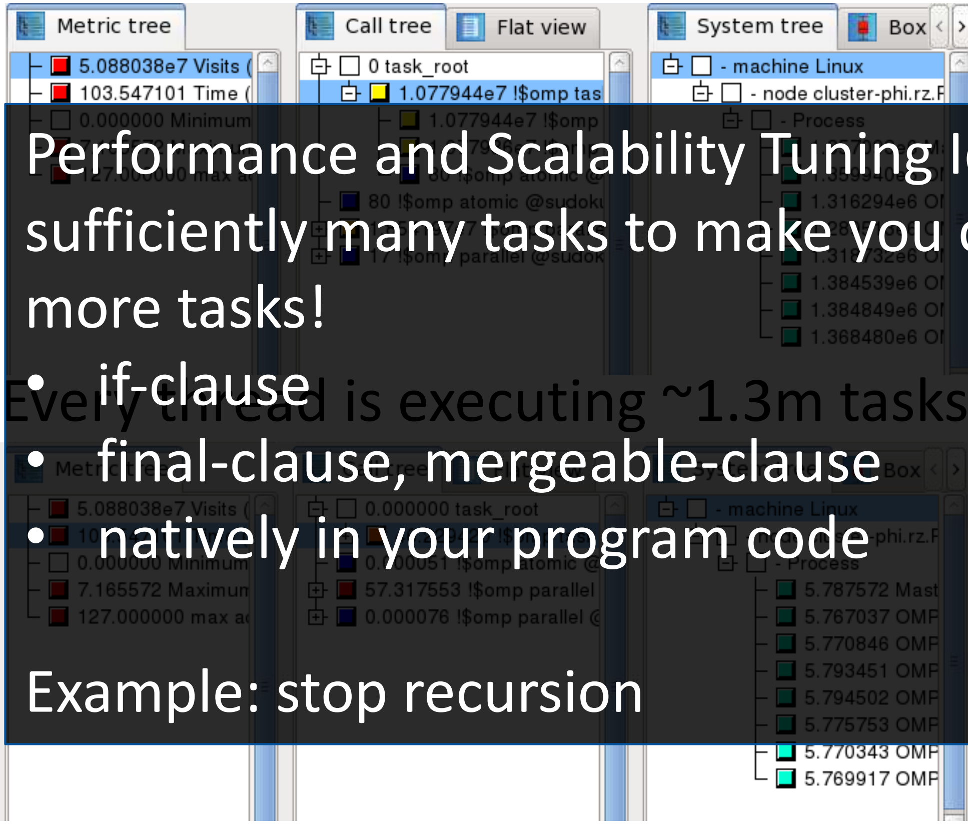
Tracing gives more details:



Tasks get much smaller down the call-stack.

Performance Analysis

Event-based profiling gives a good overview :

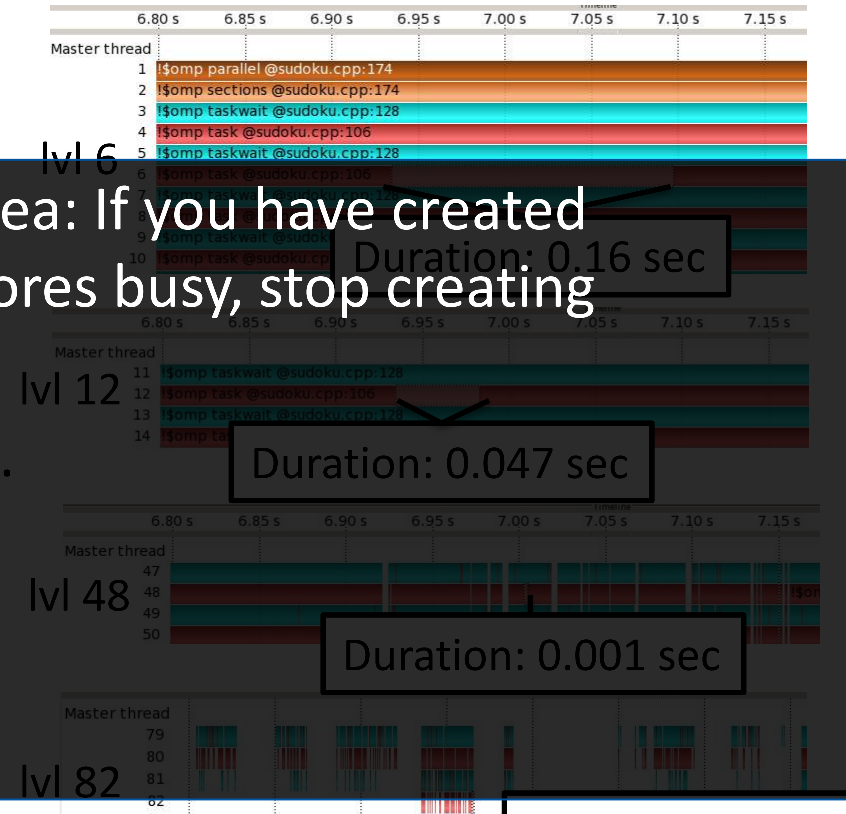


Every thread is executing ~1.3m tasks…

… in ~5.7 seconds.

=> average duration of a task is ~4.4 µs

Tracing gives more details:



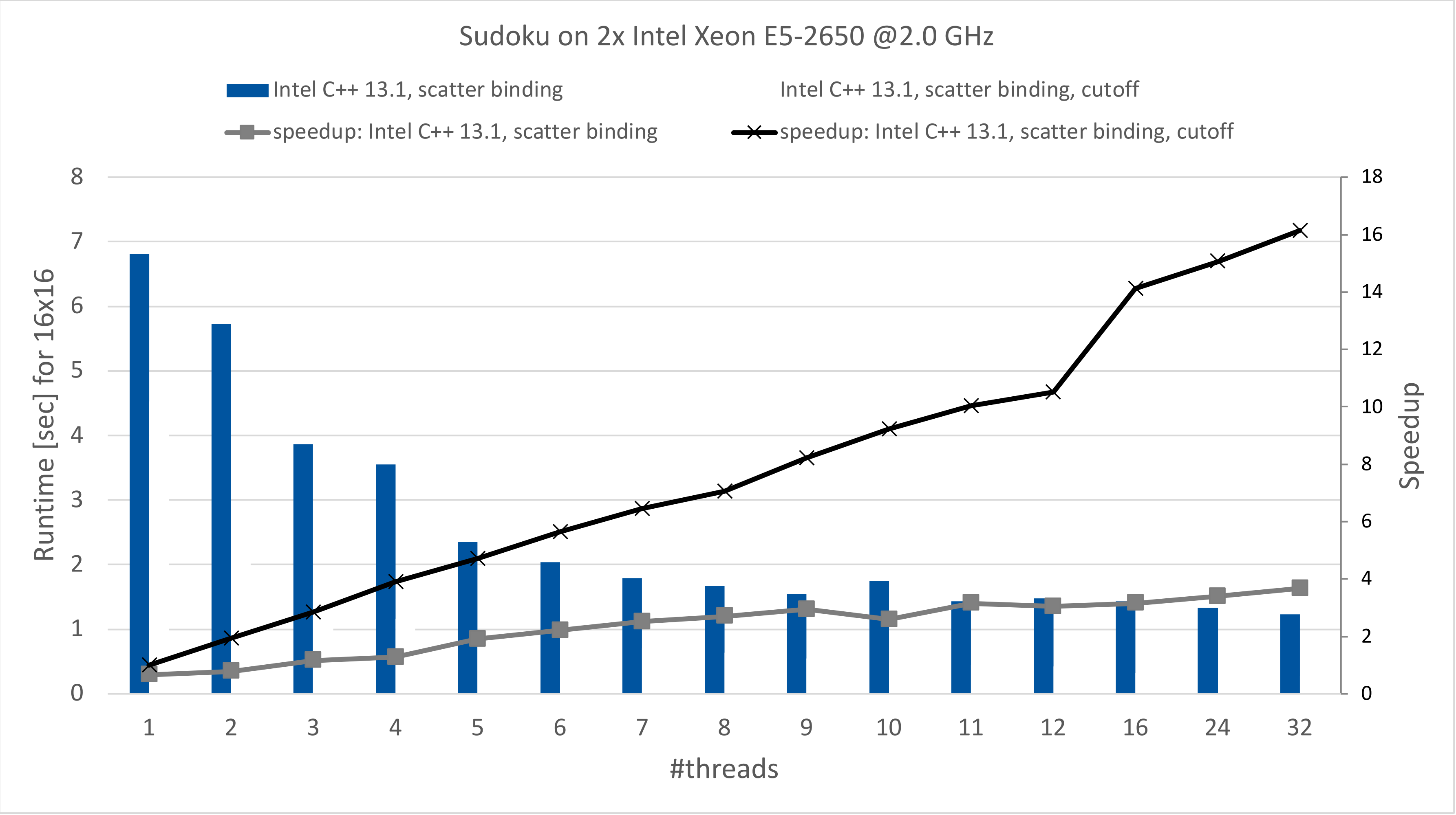
Tasks get much smaller down the call-stack.

Performance and Scalability Tuning Idea: If you have created sufficiently many tasks to make you cores busy, stop creating more tasks!

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

Example: stop recursion

Performance Evaluation



Scheduling

Tasks in OpenMP: Scheduling

- **Default: Tasks are *tied* to the thread that first executes them → not neccessarily the creator. Scheduling constraints:**
  - → Only the thread a task is tied to can execute it
  - → A task can only be suspended at task scheduling points
    - → Task creation, task finish, `taskwait`, `barrier`, `taskyield`
  - → If task is not suspended in a barrier, executing thread can only switch to a direct descendant of all tasks tied to the thread
- **Tasks created with the `untied` clause are never tied**
  - → Resume at task scheduling points possibly by different thread
  - → ~~No scheduling restrictions, e.g., can be suspended at any point~~
  - → But: More freedom to the implementation, e.g., load balancing

Unsafe use of `untied` Tasks

- **Problem: Because untied tasks may migrate between threads at any point, thread-centric constructs can yield unexpected results**

- **Remember when using `untied` tasks:**
  - → Avoid `threadprivate` variable
  - → Avoid any use of thread-ids (i.e., `omp_get_thread_num()`)
  - → Be careful with `critical region` and *locks*

- **Simple Solution:**
  - → Create a tied task region with

```
#pragma omp task if(0)
```

The taskyield Directive

- **The `taskyield` directive specifies that the current task can be suspended in favor of execution of a different task.**
  - → Hint to the runtime for optimization and/or deadlock prevention

C/C++	Fortran
`#pragma omp taskyield`	`!$omp taskyield`

taskyield Example (1/2)

```
#include <omp.h>

void something_useful();
void something_critical();

void foo(omp_lock_t * lock, int n)
{
   for(int i = 0; i < n; i++)
      #pragma omp task
      {
         something_useful();
         while( !omp_test_lock(lock) ) {
            #pragma omp taskyield
         }
         something_critical();
         omp_unset_lock(lock);
      }
}
```

taskyield Example (2/2)

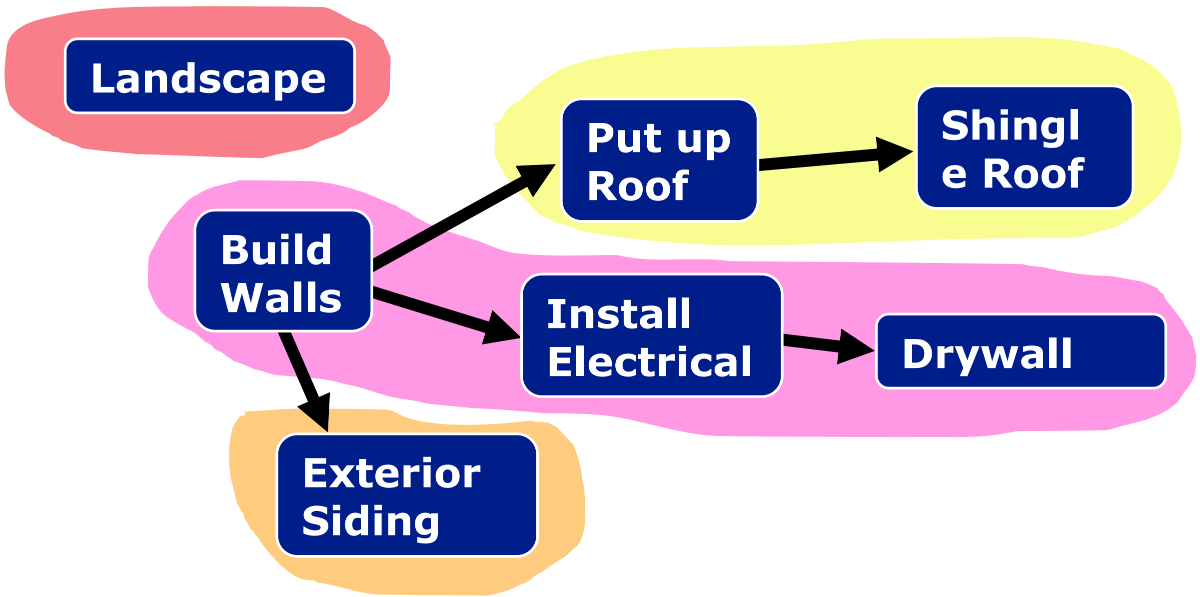
```
#include <omp.h>

void something_useful();
void something_critical();

void foo(omp_lock_t * lock, int n)
{
   for(int i = 0; i < n; i++)
      #pragma omp task
      {
         something_useful();
         while( !omp_test_lock(lock) ) {
            #pragma omp taskyield
         }
         something_critical();
         omp_unset_lock(lock);
      }
}
```

The waiting task may be suspended here and allow the executing thread to perform other work; may also avoid deadlock situations.

Tasks and Dependencies

Tasks and Dependencies

- **Catchy example: Building a house**



Tasks and Dependencies

- **Task dependencies constrain execution order and times for tasks**

- **Fine-grained synchronization of tasks**

Traditional task wait

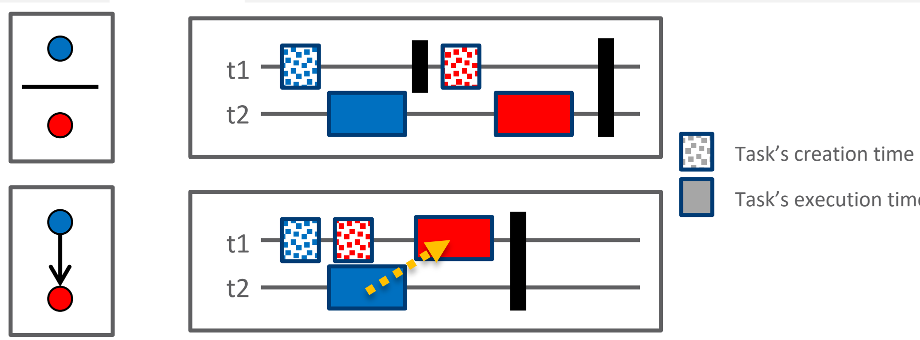
```
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task
  std::cout << x << std::endl;

  #pragma omp taskwait

  #pragma omp task
  x++;
}
```

Dependencies

```
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++;
}
```

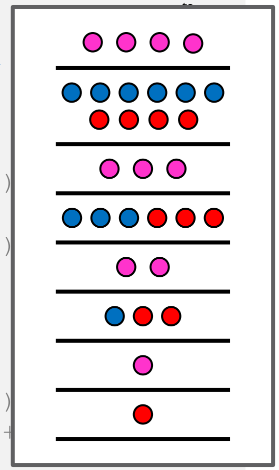
Task wait

Dependencies



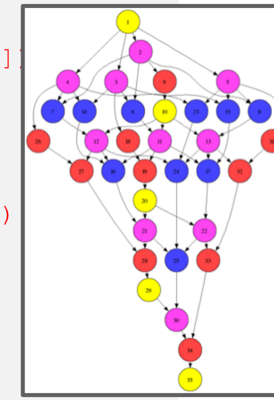
More Complex Example: Cholesky Factorization

```
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
      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
        dgemm(a[k][i], a[k][j], a[j][i], ts, ts);
      }
      #pragma omp task
      syrk(a[k][i], a[i][i], ts, ts);
    }
    #pragma omp taskwait
  }
}
```



Traditional task wait

```
void cholesky(int ts, int nt, double* a[nt][nt]) {
  for (int k = 0; k < nt; k++) {
    // Diagonal Block factorization
    #pragma omp task depend(inout: a[k][k])
    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);
    }

    // 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);
    }
  }
}
```



Dependencies

Questions?