OpenMP Tasking

Christian Terboven

Michael Klemm

Members of the OpenMP Language Committee

Agenda

- Intro by Example: Sudoku
- Scheduling and Dependencies
- Tasking Clauses

Intro by Example: Sudoku

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) Find an empty field

(2) Insert a number

(3) Check Sudoku

(4 a) If invalid:
Delete number,
Insert next number

(4 b) If valid:
Go to next field

The OpenMP Task Construct

C/C++

```
#pragma omp task [clause]
... structured block ...
```

Fortran

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

- Each encountering thread/task creates a new task
  - → Code and data is being packaged up
  - → Tasks can be nested
    - → Into another task directive
    - → Into a Worksharing construct
- Data scoping clauses:
    - → `shared`(*list*)
    - → `private`(*list*) `firstprivate`(*list*)
    - → `default`(*shared* | *none*)

Barrier and Taskwait Constructs

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

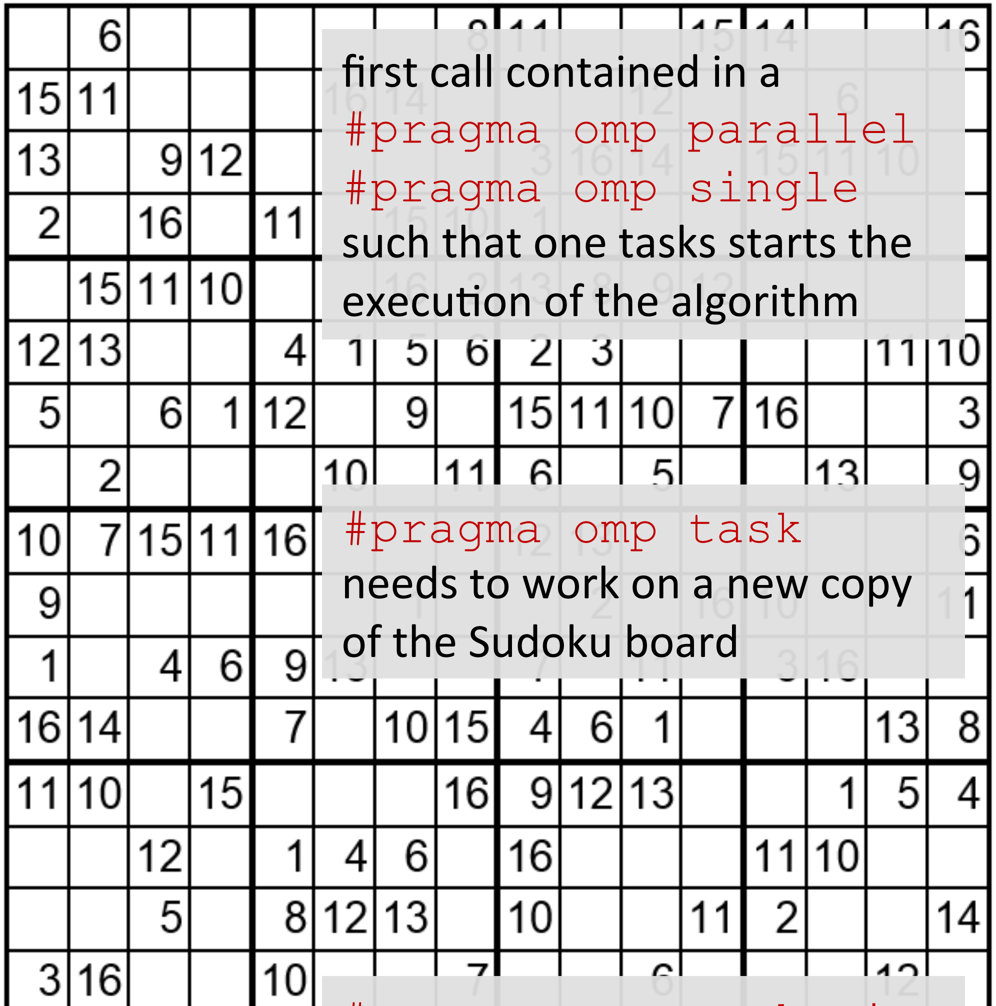
C/C++
```
#pragma omp barrier
```

- Task barrier: `taskwait`
  - → Encountering task is suspended until child tasks complete
    - → Applies only to children, not descendants!

C/C++
```
#pragma omp taskwait
```

Parallel Brute-force Sudoku

- This parallel algorithm finds all valid solutions



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) Insert a number

(3) Check Sudoku

(4 a) If invalid:
Delete number,
Insert next number

(4 b) If valid:
Go to next field

Wait for completion

Parallel Brute-force Sudoku (2/3)

- 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 (3/3)

- The actual implementation

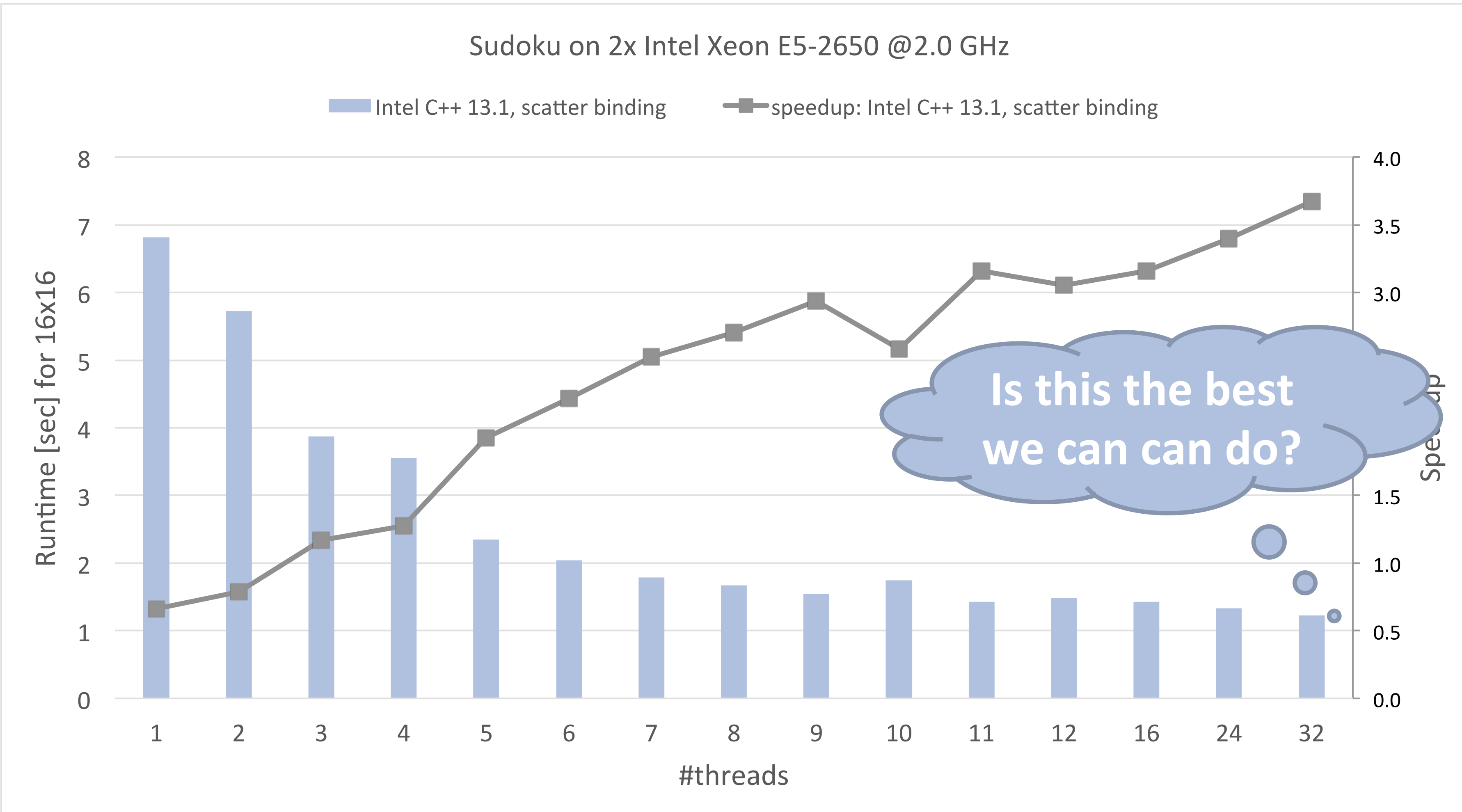
```
for (int i = 1; i <= sudoku->getFieldSize(); i++) {
   if (!sudoku->check(x, y, i)) {
#pragma omp task firstprivate(i,x,y,sudoku)
{
      // create from copy constructor
      CSudokuBoard new_sudoku(*sudoku);
      new_sudoku.set(y, x, i);
      if (solve_parallel(x+1, y, &new_sudoku)) {
      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

#pragma omp taskwait
```

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

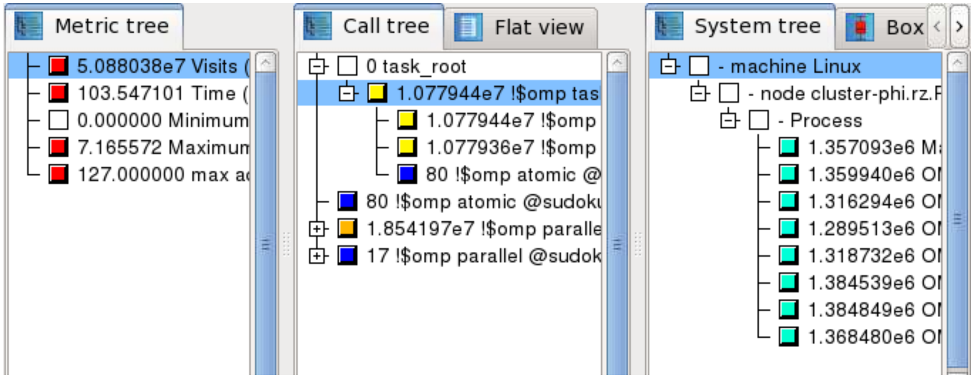
`#pragma omp taskwait`
wait for all child tasks

Performance Evaluation

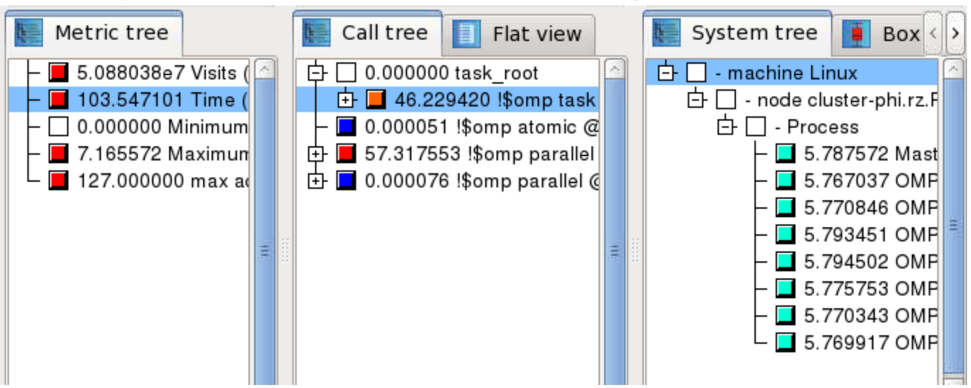


Performance Analysis

Event-based profiling gives a good overview :



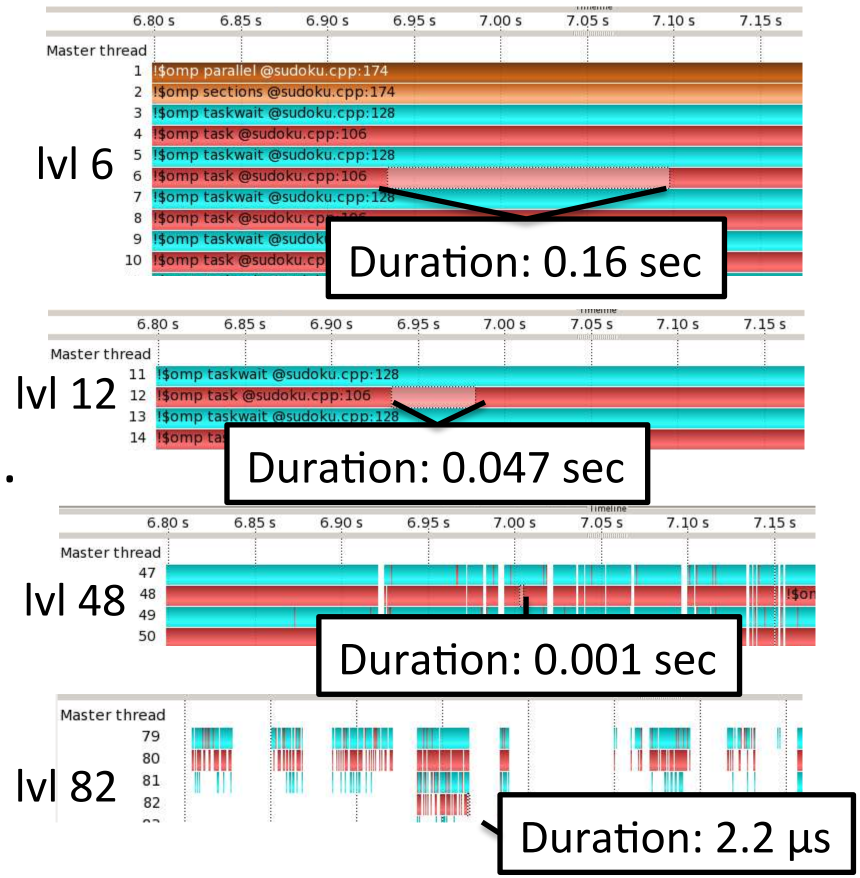
Every thread is executing ~1.3m tasks…



… in ~5.7 seconds.

=> average duration of a task is ~4.4 µs

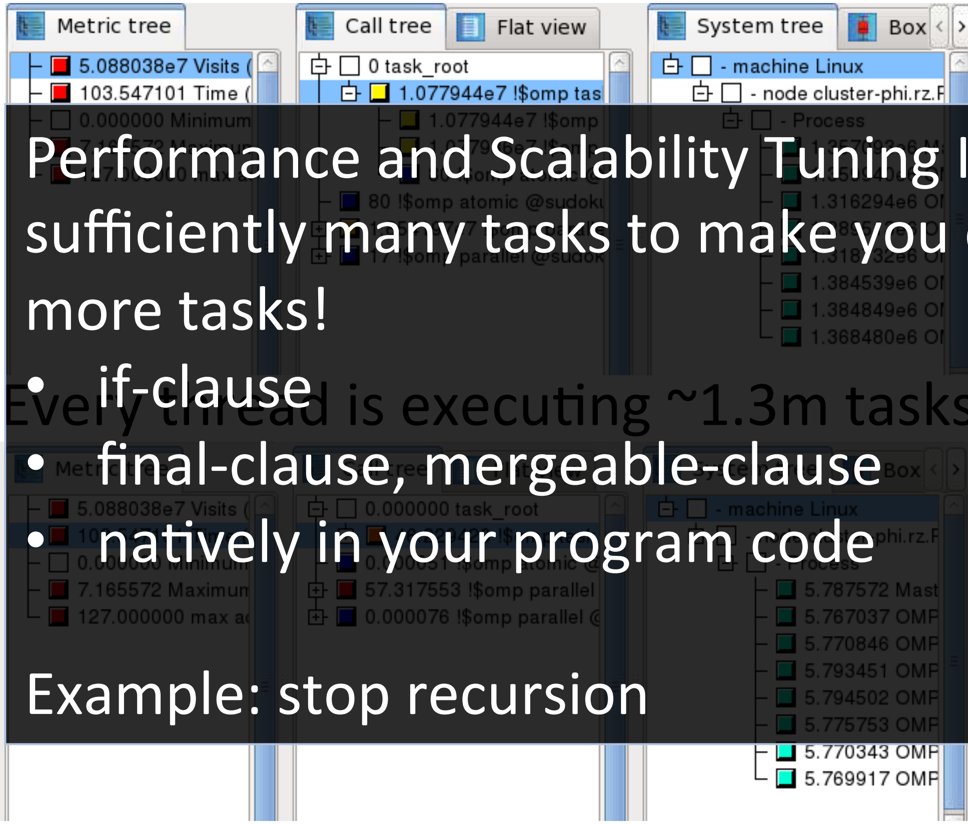
Tracing gives more details:



Tasks get much smaller down the call-stack.

Performance Analysis

Event-based profiling gives a good overview :

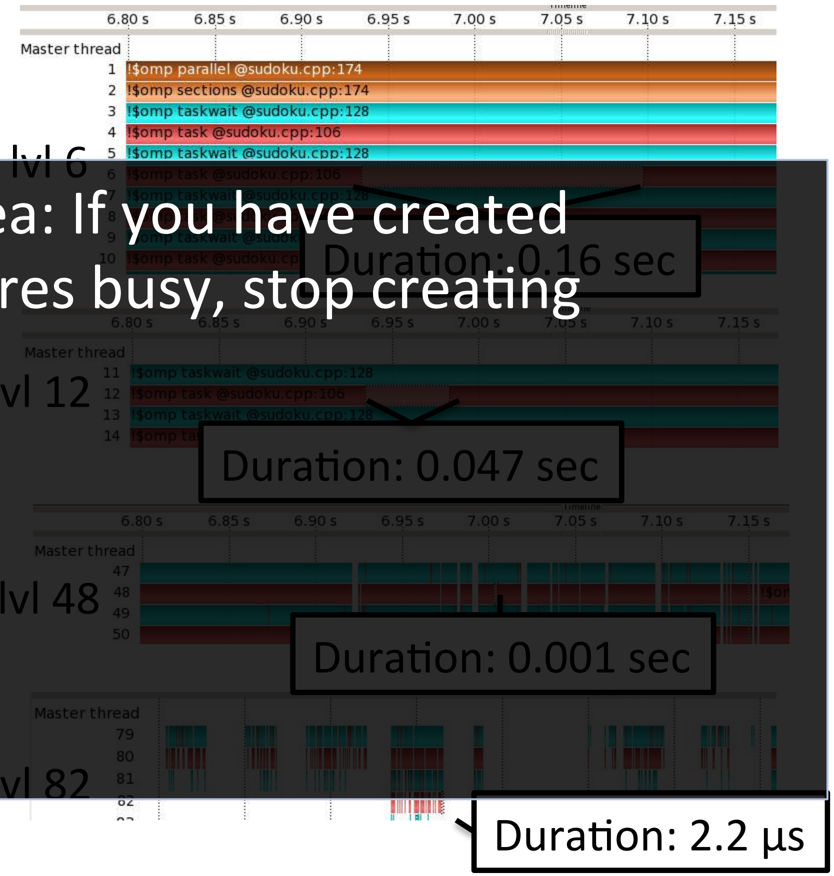


Every thread is executing ~1.3m tasks…

… in ~5.7 seconds.

=> average duration of a task is ~4.4 μs

Tracing gives more details:



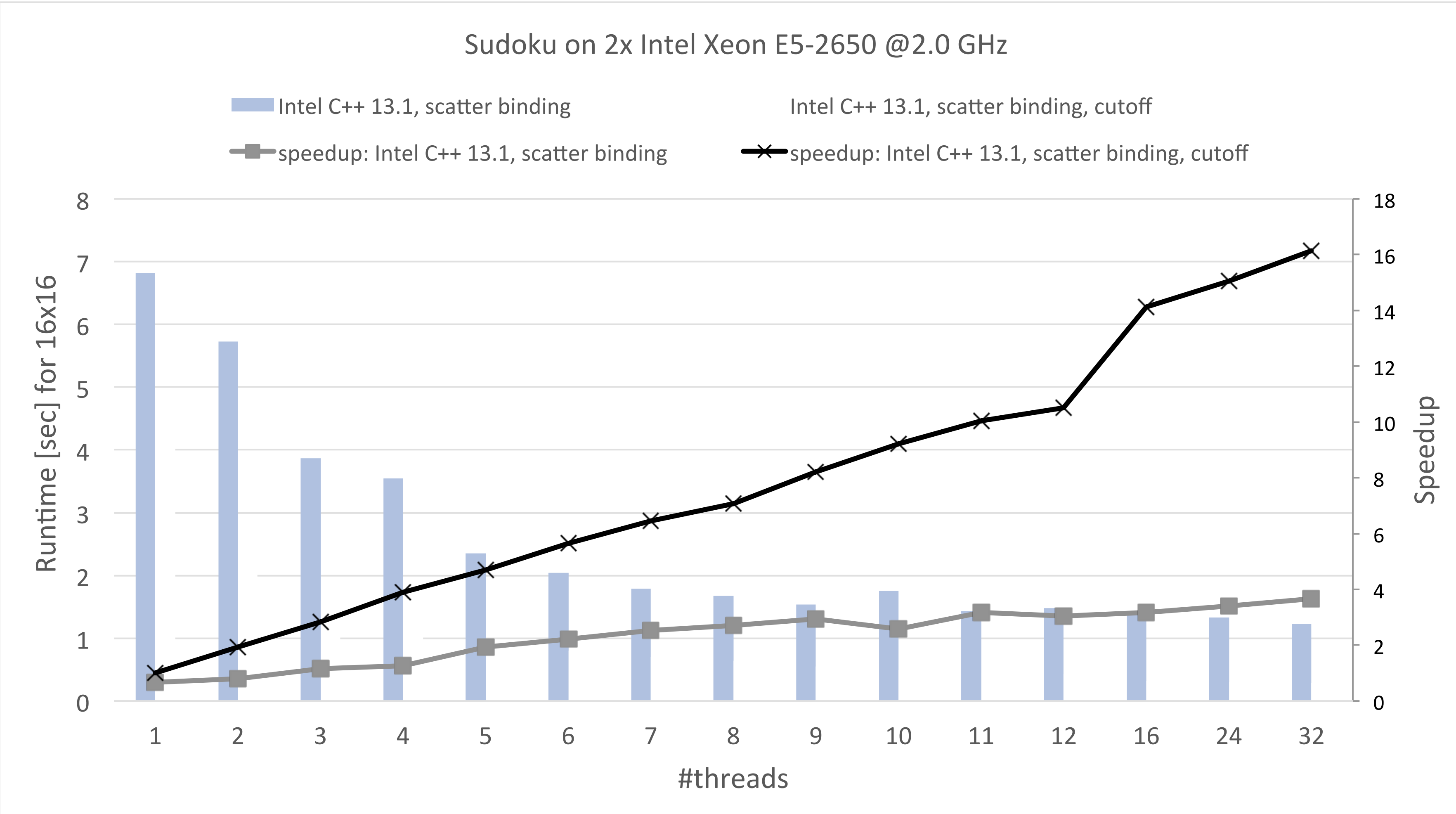
Tasks get much smaller down the call-stack.

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



`if` Clause

- If the expression of an `if` clause on a task evaluates to `false`
  - →The encountering task is suspended
  - →The new task is executed immediately
  - →The parent task resumes when the new task finishes
  - → Used for optimization, e.g., avoid creation of small tasks

Scheduling and Dependencies

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

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

```
#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.

The taskgroup Construct

C/C++

```
#pragma omp taskgroup
... structured block ...
```

Fortran

```
!$omp taskgroup
... structured block ...
!$omp end task
```

- Specifies a wait on completion of child tasks and their descendant tasks
  - → „deeper“ sychronization than `taskwait`, but
  - → with the option to restrict to a subset of all tasks (as opposed to a `barrier`)

The depend Clause

C/C++

```
#pragma omp task depend(dependency-type: list)
... structured block ...
```

- The *task dependence* is fulfilled when the predecessor task has completed
  - → `in` dependency-type: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `out` or `inout` clause.
  - → `out` and `inout` dependency-type: The generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `in`, `out`, or `inout` clause.
  - → The list items in a `depend` clause may include array sections.

Concurrent Execution w/ Dep.

Degree of parallism exploitable in this concrete example:
T2 and T3 (2 tasks), T1 of next iteration has to wait for them

- Note: variables in the depend clause do not necessarily have to indicate the data flow

```
void process_in_parallel() {
   #pragma omp parallel
   #pragma omp single
   {
      int x = 1;
      ...
      for (int i = 0; i < T; ++i) {
         #pragma omp task shared(x, ...) depend(out: x)  // T1
            preprocess_some_data(...);
         #pragma omp task shared(x, ...) depend(in: x)   // T2
            do_something_with_data(...);
         #pragma omp task shared(x, ...) depend(in: x)   // T3
            do_something_independent_with_data(...);
      }
   } // end omp single, omp parallel
}
```

T1 has to be completed before T2 and T3 can be executed.

T2 and T3 can be executed in parallel.

„Real“ Task Dependencies

Jack Dongarra on OpenMP Task Dependencies:

[…] The appearance of DAG scheduling constructs in the OpenMP 4.0 standard offers a particularly important example of this point. Until now, libraries like PLASMA had to rely on custom built task schedulers; […] However, the inclusion of DAG scheduling constructs in the OpenMP standard, along with the rapid implementation of support for them (with excellent multithreading performance) in the GNU compiler suite, throws open the doors to widespread adoption of this model in academic and commercial applications for shared memory. **We view OpenMP as the natural path forward for the PLASMA library and expect that others will see the same advantages to choosing this alternative.**

Full article here:
http://www.hpcwire.com/2015/10/19/numerical-algorithms-and-libraries-at-exascale/

Tasking Clauses

The `taskloop` Construct

- Parallelize a loop using OpenMP tasks
  - → Cut loop into chunks
  - → Create a task for each loop chunk

- Syntax (C/C++)

```
#pragma omp taskloop [simd] [clause[[,] clause],…]
for-loops
```

- Syntax (Fortran)

```
!$omp taskloop[simd] [clause[[,] clause],…]
do-loops
[!$omp end taskloop [simd]]
```

Clauses for `taskloop` Construct

- Taskloop constructs inherit clauses both from worksharing constructs and the `task` construct
  - → `shared`, `private`
  - → `firstprivate`, `lastprivate`
  - → `default`
  - → `collapse`
  - → `final`, `untied`, `mergeable`

- `grainsize(`*`grain-size`*`)`
  Chunks have at least *grain-size* and max 2**grain-size* loop iterations

- `num_tasks(num-tasks)`
  Create *num-tasks* tasks for iterations of the loop

`priority` Clause

C/C++

```
#pragma omp task priority(priority-value)
... structured block ...
```

- The *priority* is a hint to the runtime system for task execution order
- Among all tasks ready to be executed, higher priority tasks are recommended to execute before lower priority ones
  - → priority is non-negative numerical scalar (default: 0)
  - → priority <= max-task-priority ICV
    - → environment variable OMP_MAX_TASK_PRIORITY
- It is not allowed to rely on task execution order being determined by this clause!

`final` Clause

- For recursive problems that perform task decomposition, stopping task creation at a certain depth exposes enough parallelism but reduces overhead.

C/C++	Fortran
`#pragma omp task final(expr)`	`!$omp task final(expr)`

- Merging the data environment may have side-effects

```
void foo(bool arg)
{
   int i = 3;
   #pragma omp task final(arg) firstprivate(i)
      i++;
   printf("%d\n", i);   // will print 3 or 4 depending on expr
}
```

`mergeable` Clause

- If the mergeable clause is present, the implementation might merge the task‘s data environment
  - → if the generated task is undeferred or included
    - → undeferred: if clause present and evaluates to false
    - → included: final clause present and evaluates to true

C/C++	Fortran
`#pragma omp task mergeable`	`!$omp task mergeable`

- Personal Note: As of today, no compiler or runtime exploits final and/or mergeable so that real world applications would profit from using them ☹.