OpenMP Tasking

Dirk Schmidl
IT Center, RWTH Aachen University
Member of the HPC Group
schmidl@itc.rwth-aachen.de

Christian Terboven
IT Center, RWTH Aachen University
Deputy lead of the HPC Group
terboven@itc.rwth-aachen.de

Tasking

The 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*)

Tasks in OpenMP: Data Scoping

- **Some rules from *Parallel Regions* apply:**
  - → Static and Global variables are shared
  - → Automatic Storage (local) variables are private

- **If `shared` scoping is not derived by default:**
  - → Orphaned Task variables are `firstprivate` by default!
  - → Non-Orphaned Task variables inherit the `shared` attribute!
  - → Variables are `firstprivate` unless `shared` in the enclosing context

- **So far no verification tool is available to check Tasking programs for correctness!**

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 are complete
    - → Applies only to direct childs, not descendants!

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

Example: Fibonacci

Recursive approach to compute Fibonacci

```
int main(int argc,
         char* argv[])
{
   [...]
   fib(input);
   [...]
}
```

```
int fib(int n)   {
   if (n < 2) return n;
   int x = fib(n - 1);
   int y = fib(n - 2);
   return x+y;
}
```

- **On the following slides we will discuss three approaches to parallelize this recursive code with Tasking.**

First version parallelized with Tasking (omp-v1)

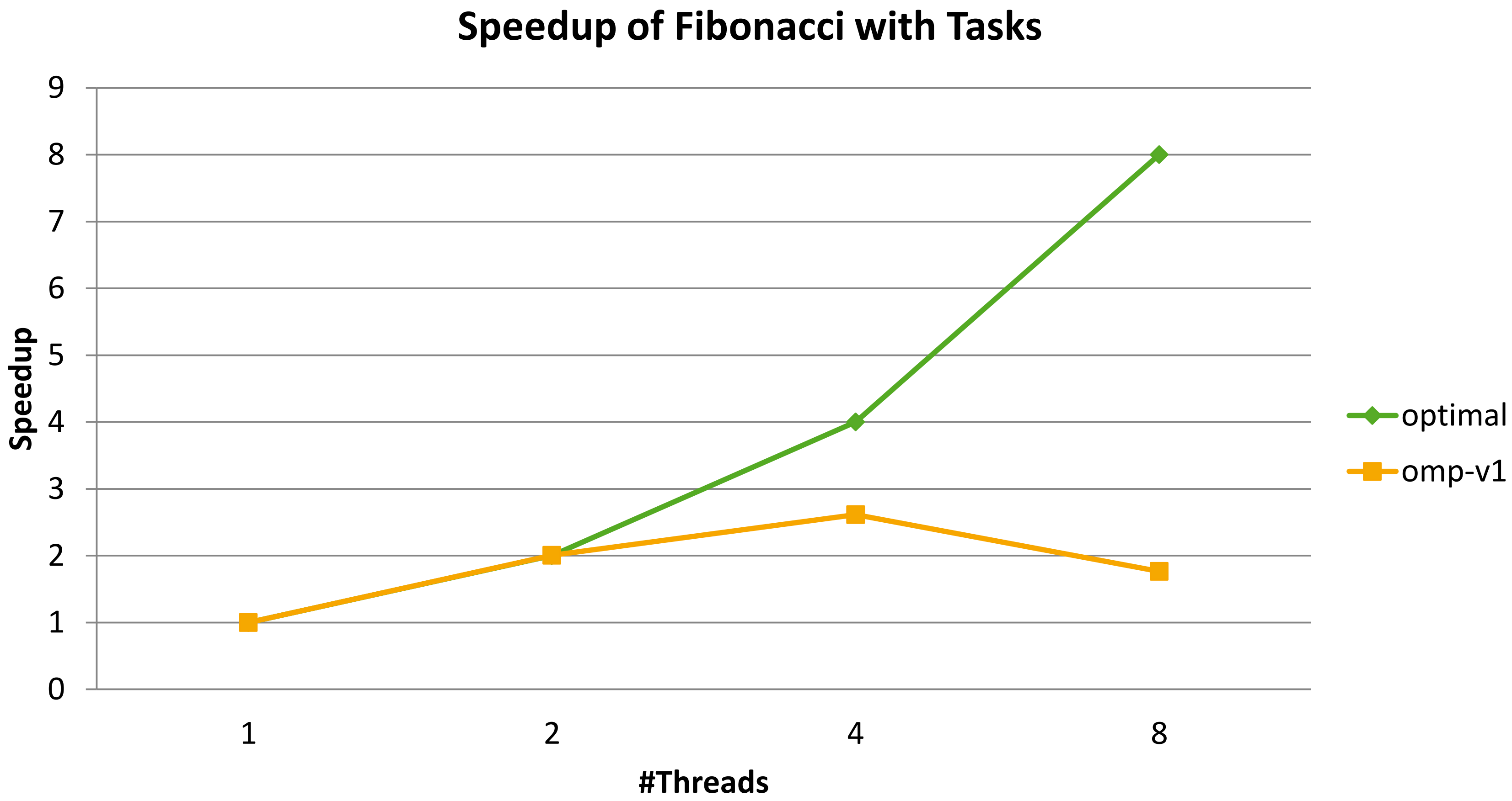
```
int main(int argc,
         char* argv[])
{
   [...]
#pragma omp parallel
{
#pragma omp single
{
   fib(input);
}
}
   [...]
}
```

```
int fib(int n)   {
   if (n < 2) return n;
int x, y;
#pragma omp task shared(x)
{
   x = fib(n - 1);
}
#pragma omp task shared(y)
{
   y = fib(n - 2);
}
#pragma omp taskwait
   return x+y;
}
```

- Only one Task / Thread enters `fib()` from `main()`, it is responsable for creating the two initial work tasks
- Taskwait is required, as otherwise `x` and `y` would be lost

Scalability measurements (1/3)

- **Overhead of task creation prevents better scalability!**



Improved parallelization with Tasking (omp-v2)

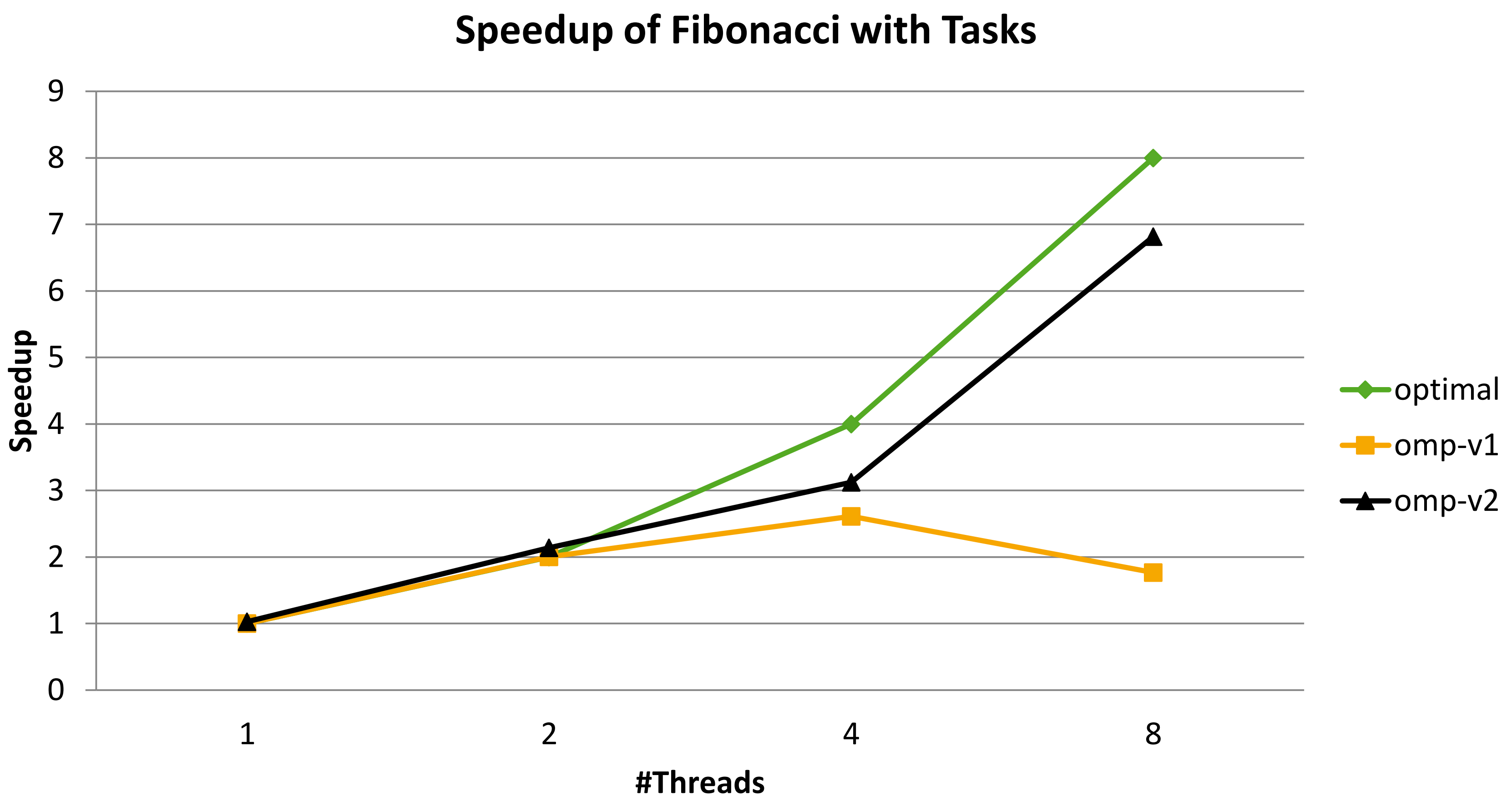
- **Improvement: Don‘t create yet another task once a certain (small enough) `n` is reached**

```
int main(int argc,
         char* argv[])
{
   [...]
#pragma omp parallel
{
#pragma omp single
{
   fib(input);
}
}
   [...]
}
```

```
int fib(int n)   {
   if (n < 2) return n;
int x, y;
#pragma omp task shared(x) \
   if(n > 30)
{
   x = fib(n - 1);
}
#pragma omp task shared(y) \
   if(n > 30)
{
   y = fib(n - 2);
}
#pragma omp taskwait
   return x+y;
}
```

Scalability measurements (2/3)

- **Speedup is ok, but we still have some overhead when running with 4 or 8 threads**



Improved parallelization with Tasking (omp-v3)

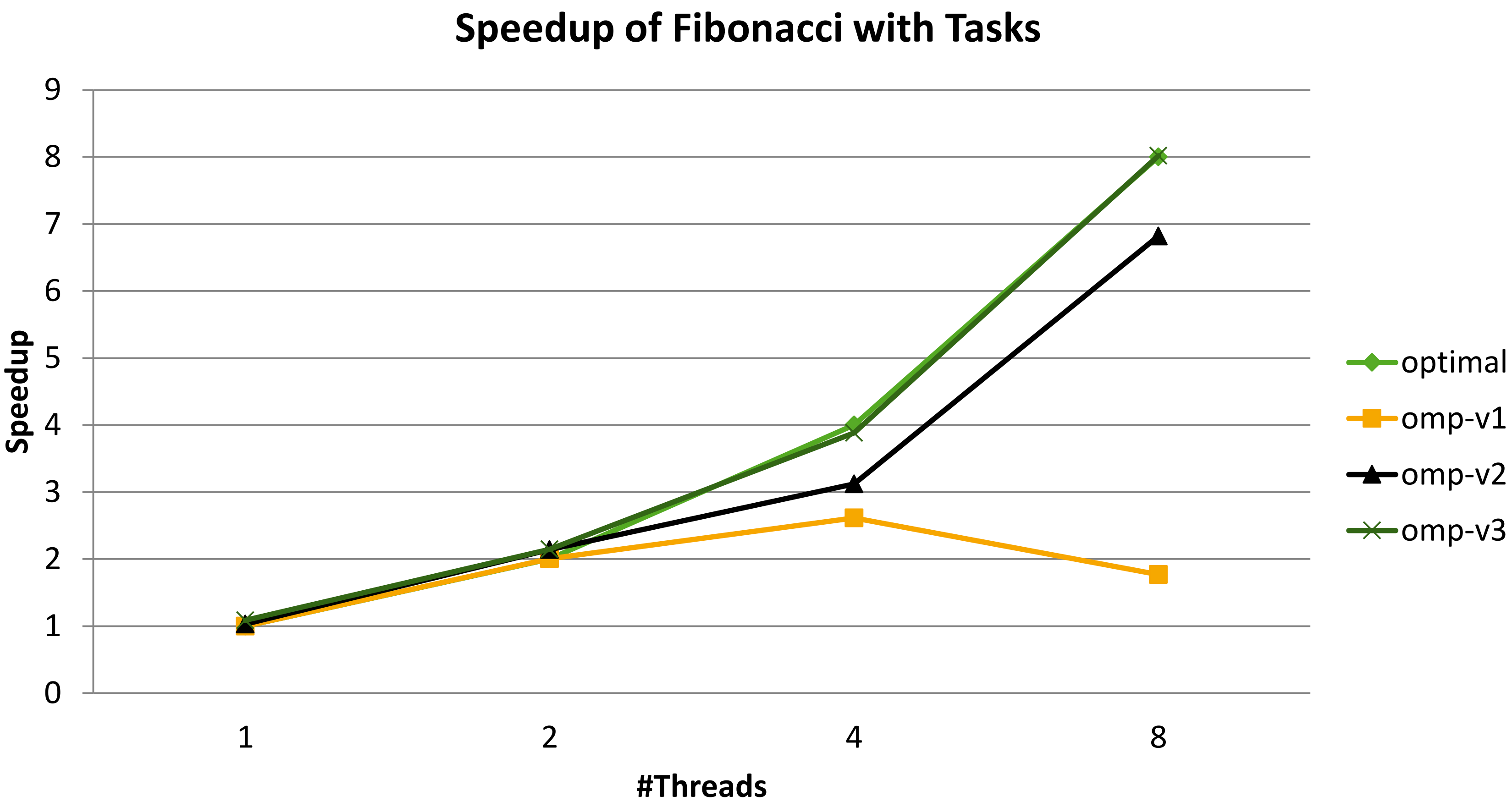
- **Improvement: Skip the OpenMP overhead once a certain `n` is reached (no issue w/ production compilers)**

```
int main(int argc,
         char* argv[])
{
   [...]
#pragma omp parallel
{
#pragma omp single
{
   fib(input);
}
}
   [...]
}
```

```
int fib(int n)   {
   if (n < 2) return n;
   if (n <= 30)
      return serfib(n);
int x, y;
#pragma omp task shared(x)
{
   x = fib(n - 1);
}
#pragma omp task shared(y)
{
   y = fib(n - 2);
}
#pragma omp taskwait
   return x+y;
}
```

Scalability measurements (3/3)

- **Everything ok now ☺**



Example: Data Scoping

Data Scoping Example (1/7)

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b)
   #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:
} } }
```

Data Scoping Example (2/7)

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

                // Scope of a: shared
                // Scope of b:
                // Scope of c:
                // Scope of d:
                // Scope of e:
} } }
```

Data Scoping Example (3/7)

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

               // Scope of a: shared
               // Scope of b: firstprivate
               // Scope of c:
               // Scope of d:
               // Scope of e:
} } }
```

Data Scoping Example (4/7)

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

                // Scope of a: shared
                // Scope of b: firstprivate
                // Scope of c: shared
                // Scope of d:
                // Scope of e:
} } }
```

Data Scoping Example (5/7)

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

                // Scope of a: shared
                // Scope of b: firstprivate
                // Scope of c: shared
                // Scope of d: firstprivate
                // Scope of e:
} } }
```

Data Scoping Example (6/7)

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

                // Scope of a: shared
                // Scope of b: firstprivate
                // Scope of c: shared
                // Scope of d: firstprivate
                // Scope of e: private
} } }
```

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

Data Scoping Example (7/7)

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

         // Scope of a: shared,         value of a: 1
         // Scope of b: firstprivate,   value of b: 0 / undefined
         // Scope of c: shared,         value of c: 3
         // Scope of d: firstprivate,   value of d: 4
         // Scope of e: private,        value of e: 5
} } }
```

Task Scheduling

- **Task Synchronization explained:**

```
#pragma omp parallel num_threads(np)
{
#pragma omp task
    function_A();
#pragma omp barrier
#pragma omp single
    {
#pragma omp task
        function_B();
    }
}
```

np Tasks created here, one for each thread

All Tasks guaranteed to be completed here

1 Task created here

B-Task guaranteed to be completed here

OpenMP Task 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

`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 new tasks finishes
  - → Used for optimization, e.g. avoid creation of small tasks

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. This may also avoid deadlock situations.

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

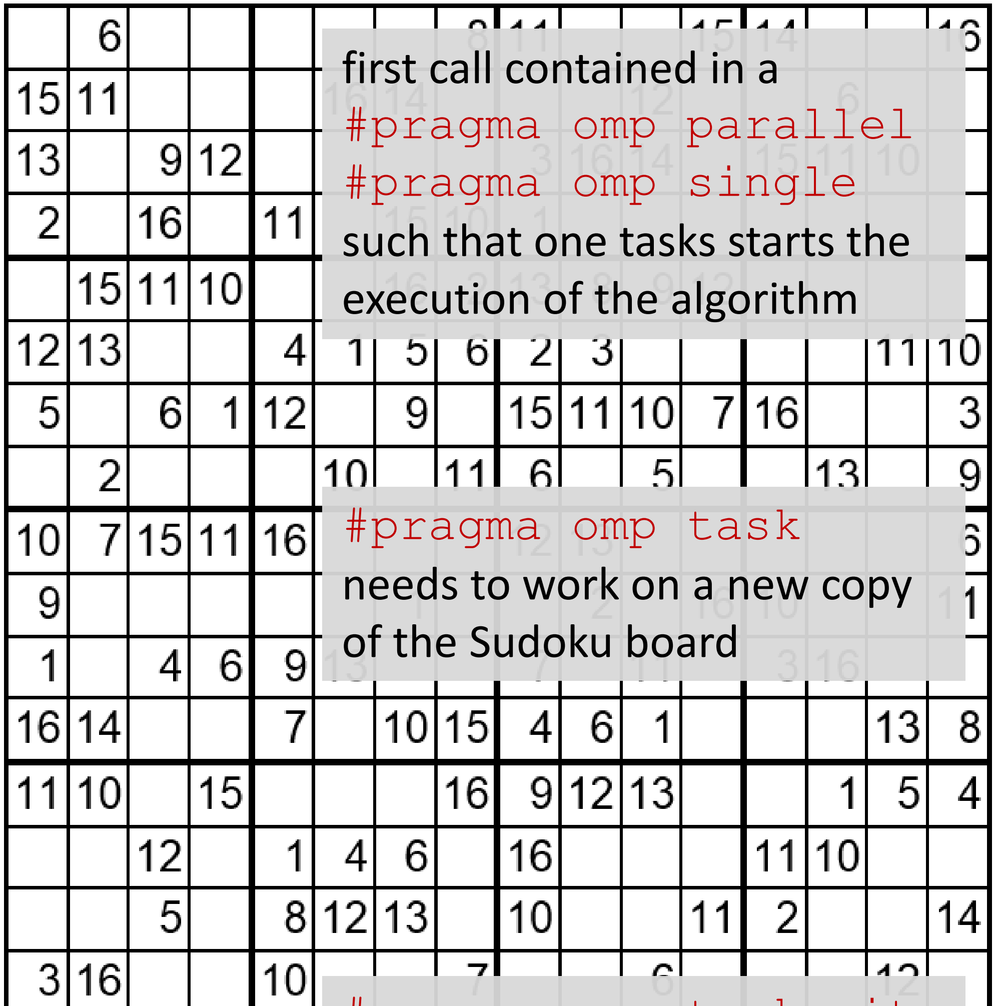
(3) Check Sudoku

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

(4 b) If valid:
Go to next field

Parallel Brute-force Sudoku (1/3)

- **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

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 do not)**

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

Parallel Brute-force Sudoku (3/3)

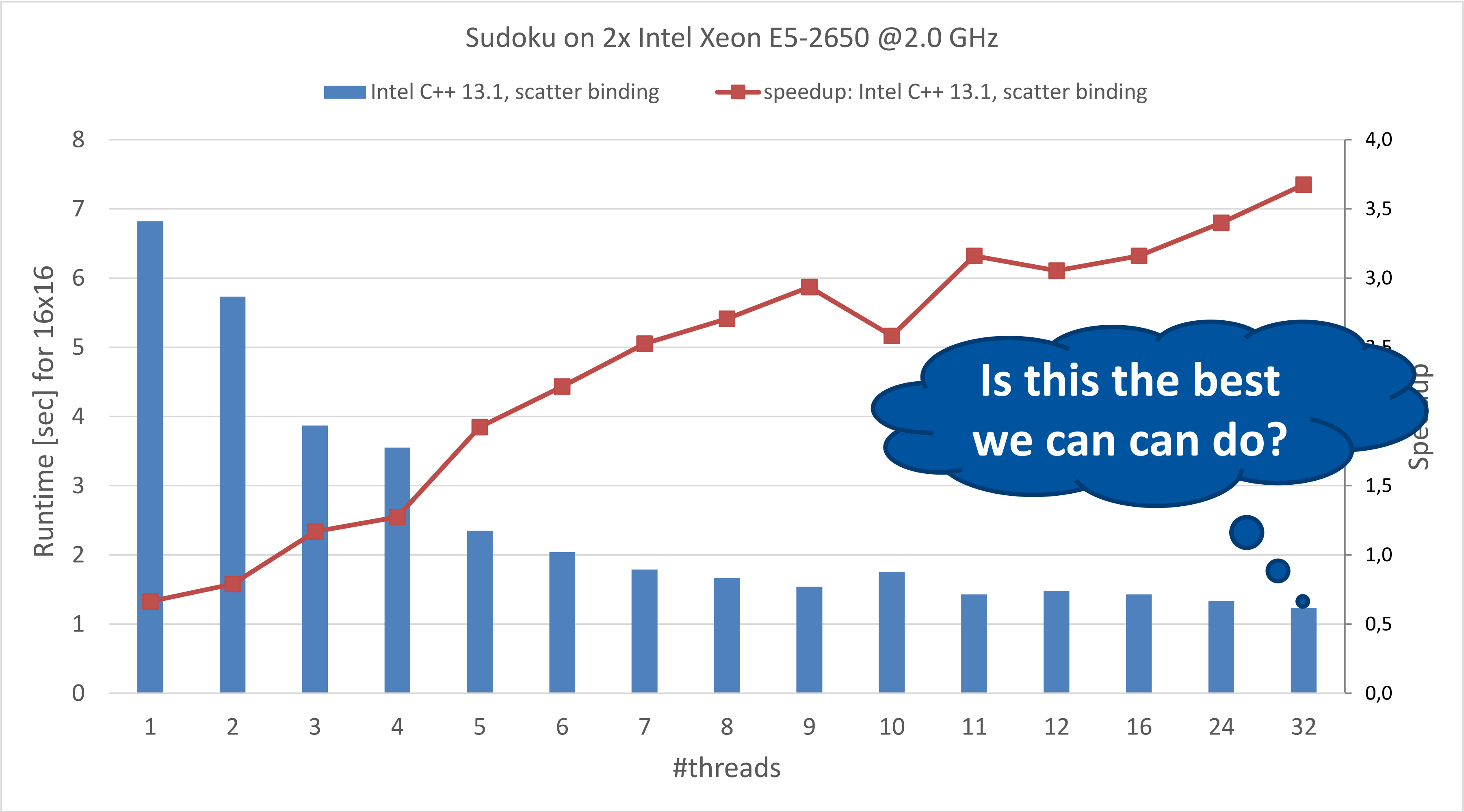
- **The actual implementation**

```
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`
needs to work on a new copy of the Sudoku board

`#pragma omp taskwait`
wait for all child tasks

Performance Evaluation



taskgroup Construct

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 descendent tasks**
  - → „deeper“ sychronization than `taskwait`, but
  - → with the option to restrict to a subset of all tasks (as opposed to a `barrier`)
- **Main use case for now in OpenMP 4.0: Cancellation…**

Questions?