OpenMP Tutorial

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

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

Tasking in OpenMP

IWOMP Tutorial: October 5th, 2016

Christian Terboven

Members of the OpenMP Language Committee

Agenda

- Intro by Example: Sudoku
- Data Scoping
- Example: Fibonacci
- Scheduling and Dependencies
- Taskloops

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
```

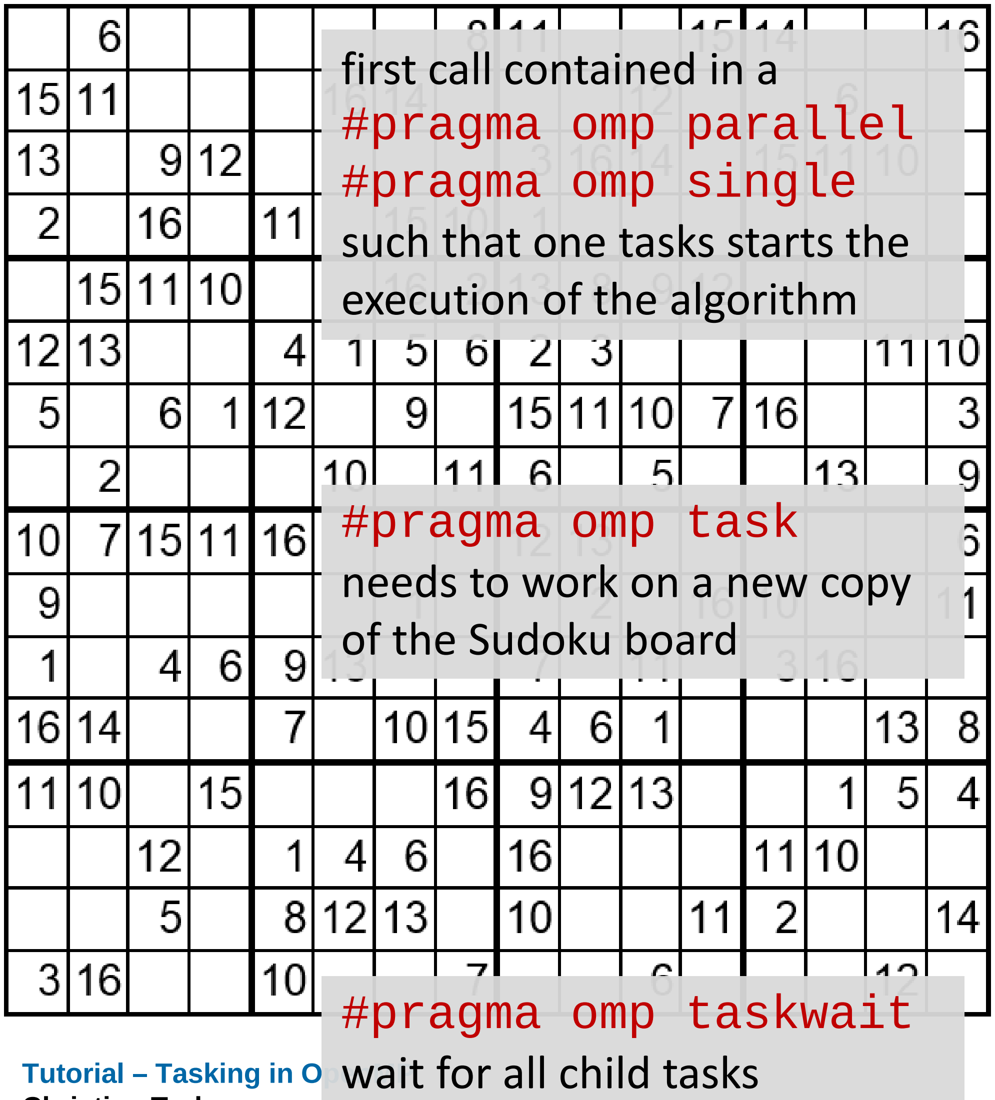
- Task barrier: `taskwait`
  - → Encountering task is suspended until child tasks are complete
    - → Applies only to direct childs, not descendants!

C/C++

```
#pragma omp taskwait
```

Parallel Brute-force Sudoku

- This parallel algorithm finds all valid solutions



(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

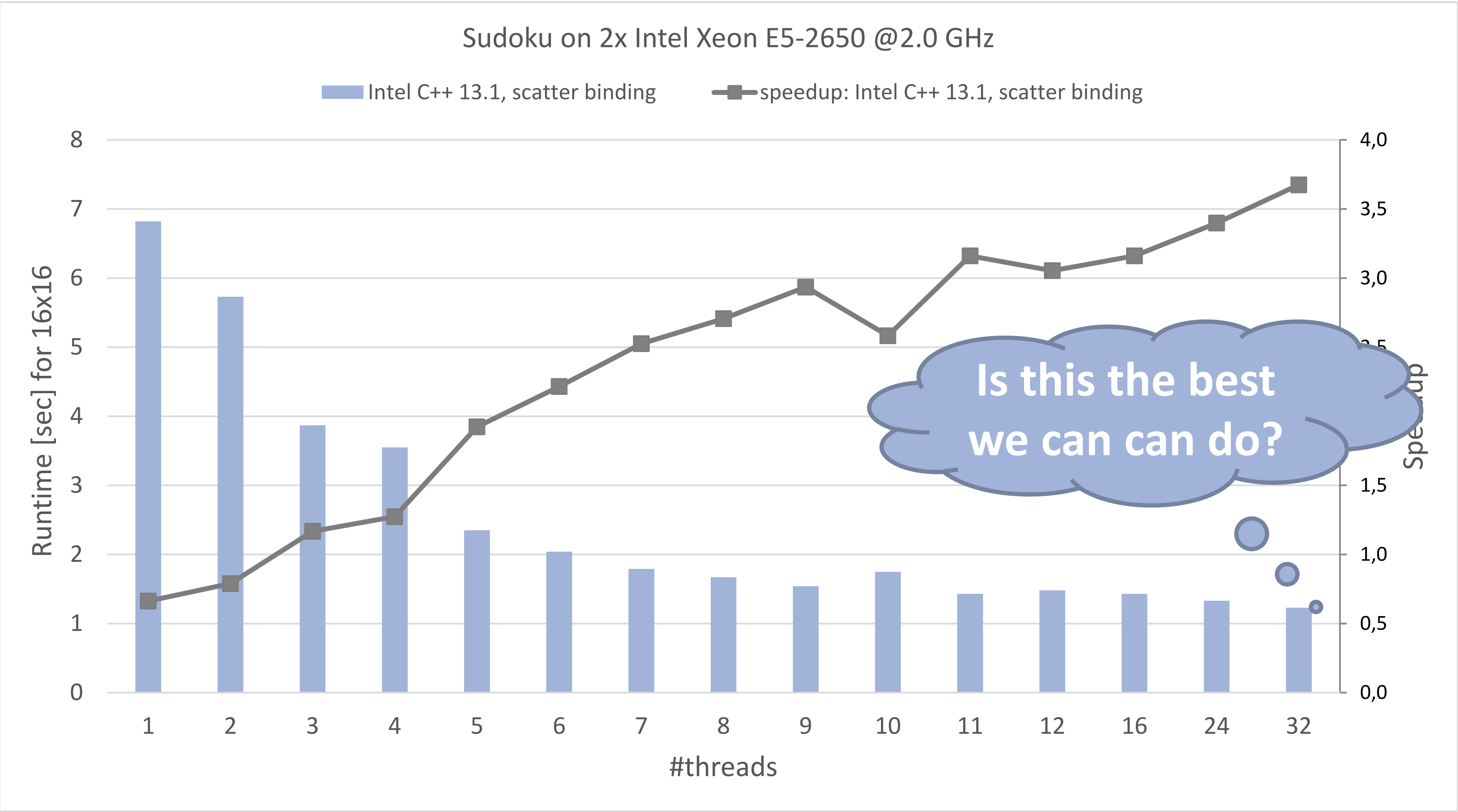
```
for (int i = 1; i <= sudoku->getFieldSize(); i++) {
   if (!sudoku->check(x, y, i)) {
#pragma omp task firstprivate(i,x,y,sudoku)
{
      // create from copy constructor
      CSudokuBoard new_sudoku(*sudoku);
      new_sudoku.set(y, x, i);
      if (solve_parallel(x+1, y, &new_sudoku)) {
      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

#pragma omp taskwait
```

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

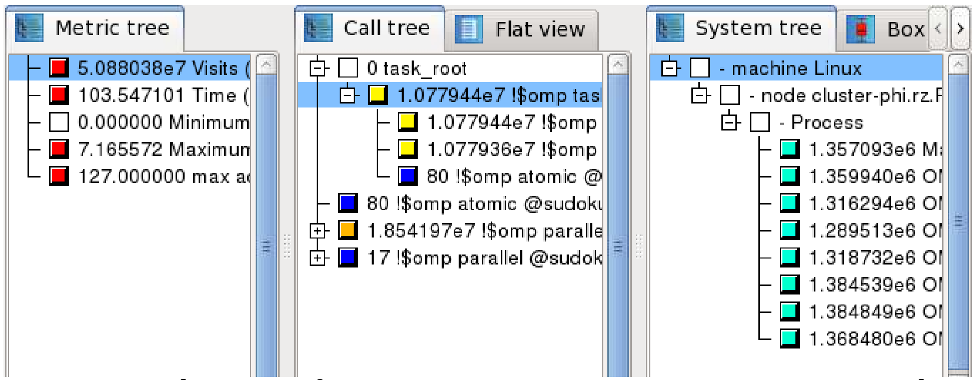
`#pragma omp taskwait`
wait for all child tasks

Performance Evaluation

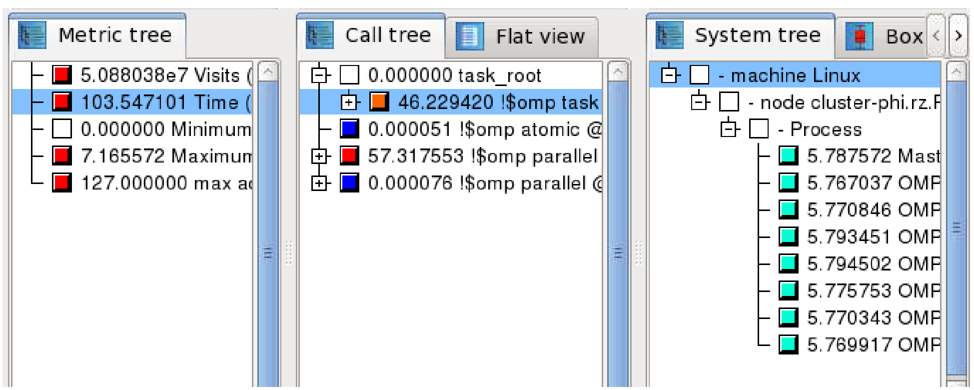


Performance Analysis

Event-based profiling gives a good overview :



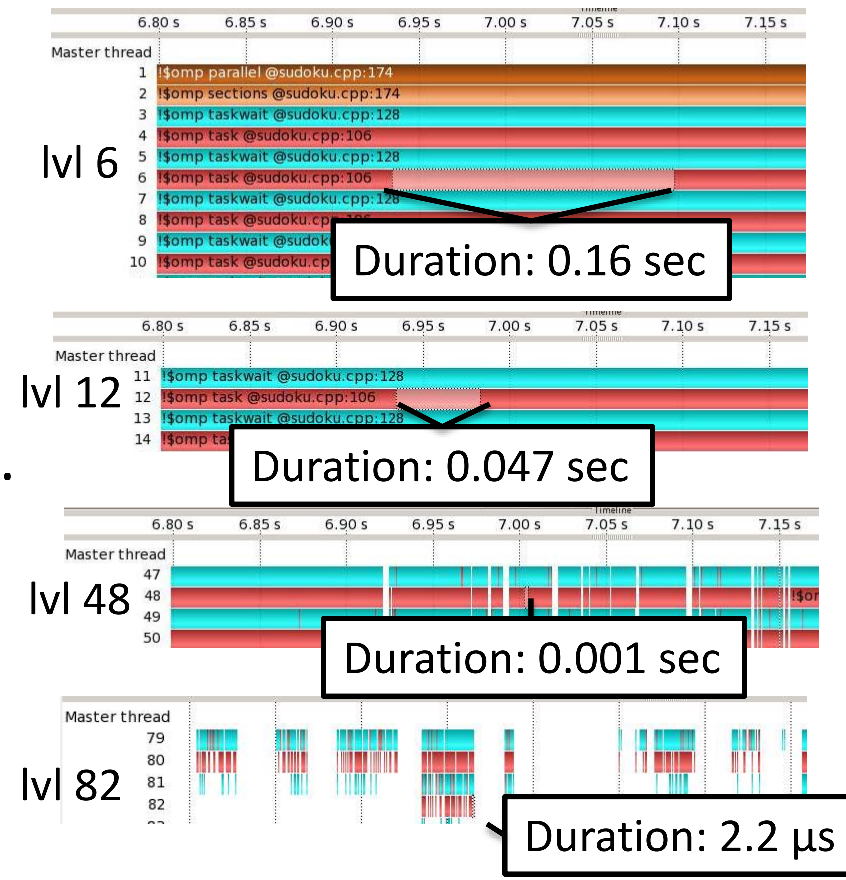
Every thread is executing ~1.3m tasks…



… in ~5.7 seconds.

=> average duration of a task is ~4.4 μs

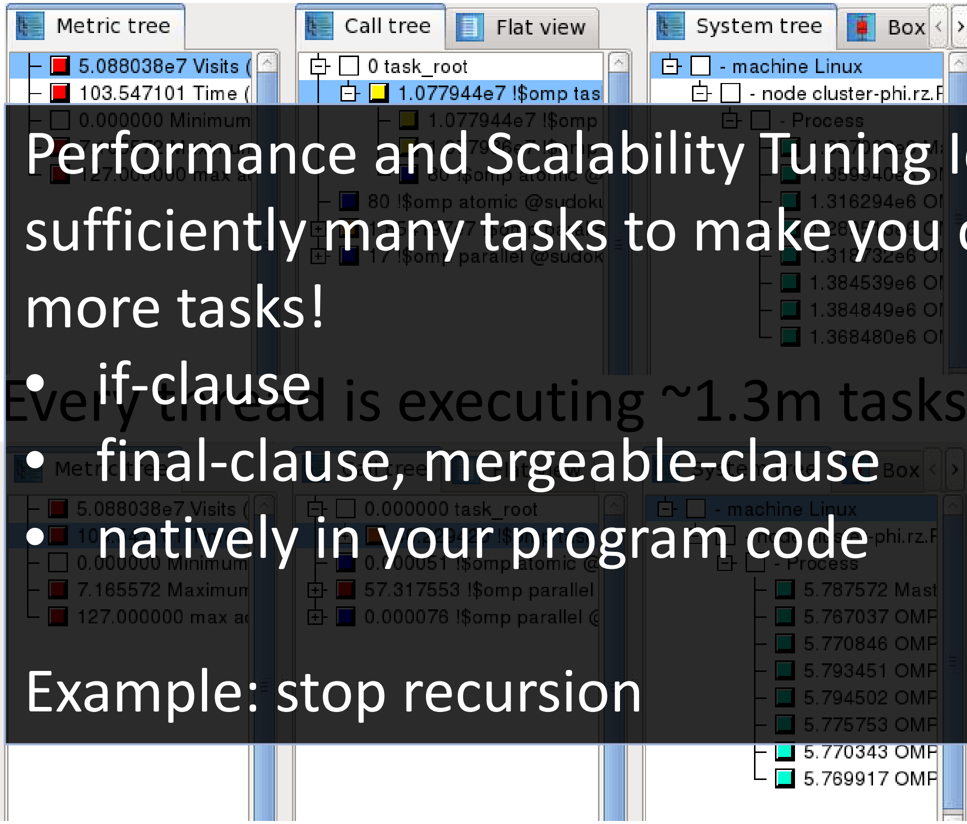
Tracing gives more details:



Tasks get much smaller down the call-stack.

Performance Analysis

Event-based profiling gives a good overview :

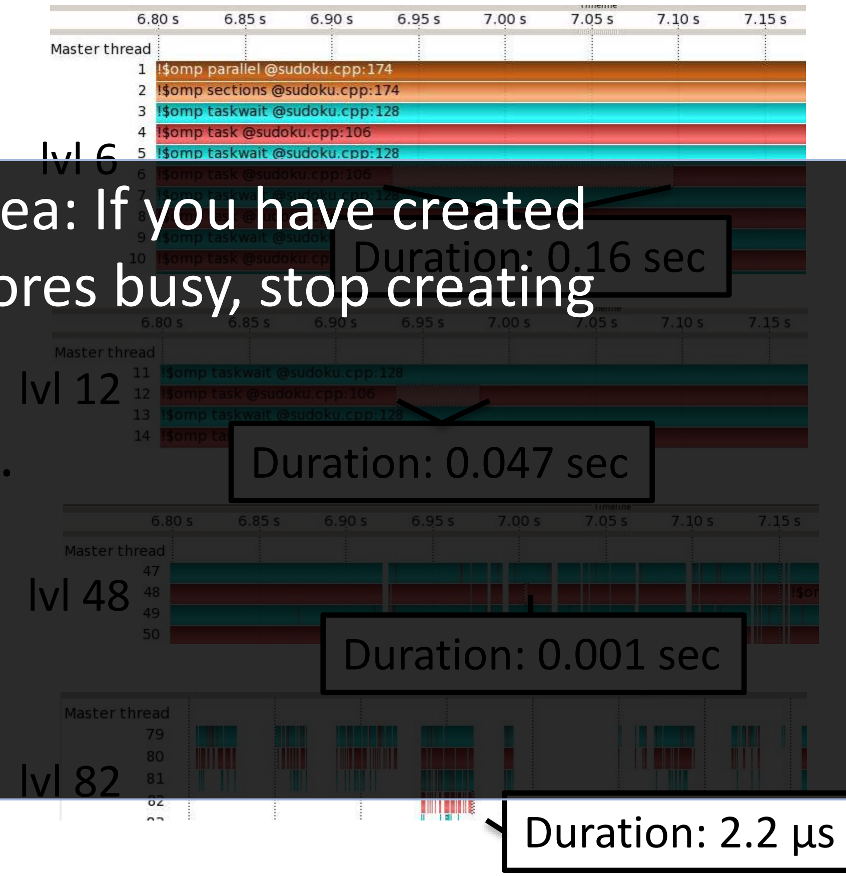


Every thread is executing ~1.3m tasks…

… in ~5.7 seconds.

=> average duration of a task is ~4.4 µs

Tracing gives more details:



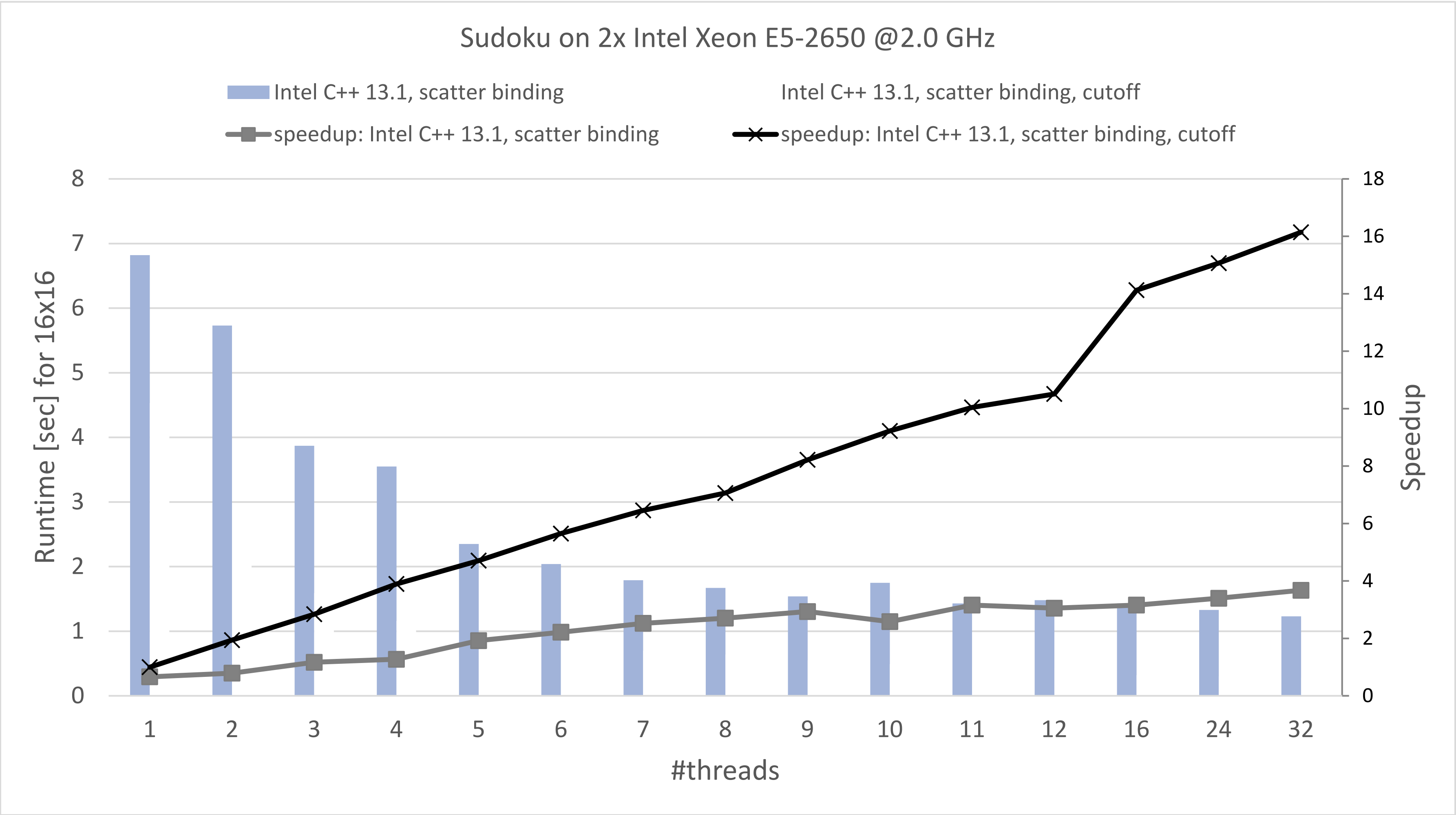
Tasks get much smaller down the call-stack.

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



Data Scoping

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 inherited:
  - → Orphaned Task variables are `firstprivate` by default!
  - → Non-Orphaned Task variables inherit the `shared` attribute!
  - → Variables are `firstprivate` unless `shared` in the enclosing context

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
} } }
```

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
} } }
```

Use default(none)!

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

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b) default(none)
   #pragma omp parallel private(b) default(none)
   {
      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
} } }
```

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)

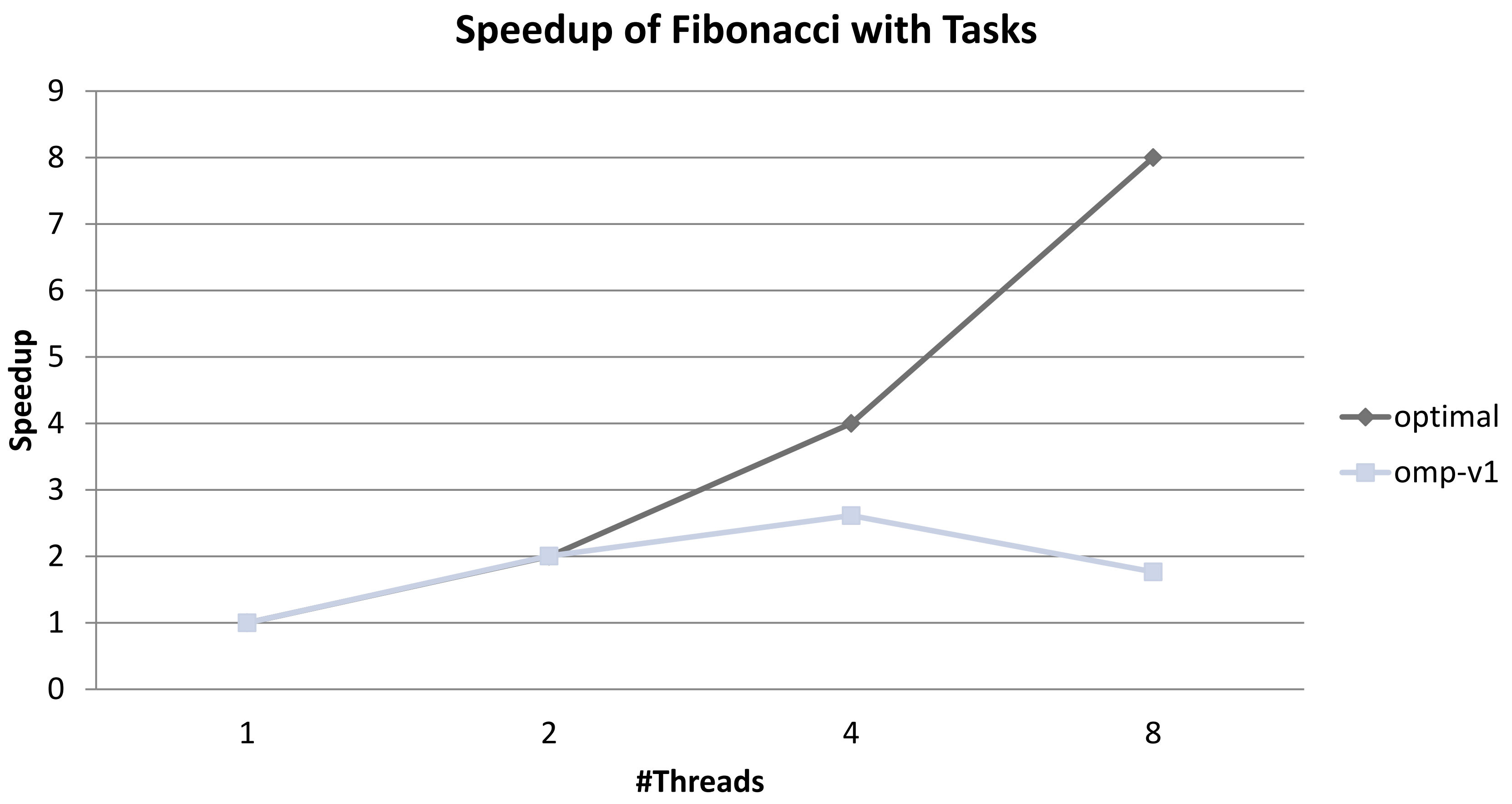
```
int main(int argc,
         char* argv[])
{
   [...]
#pragma omp parallel
{
#pragma omp single
{
   fib(input);
}
}
   [...]
}
```

```
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)

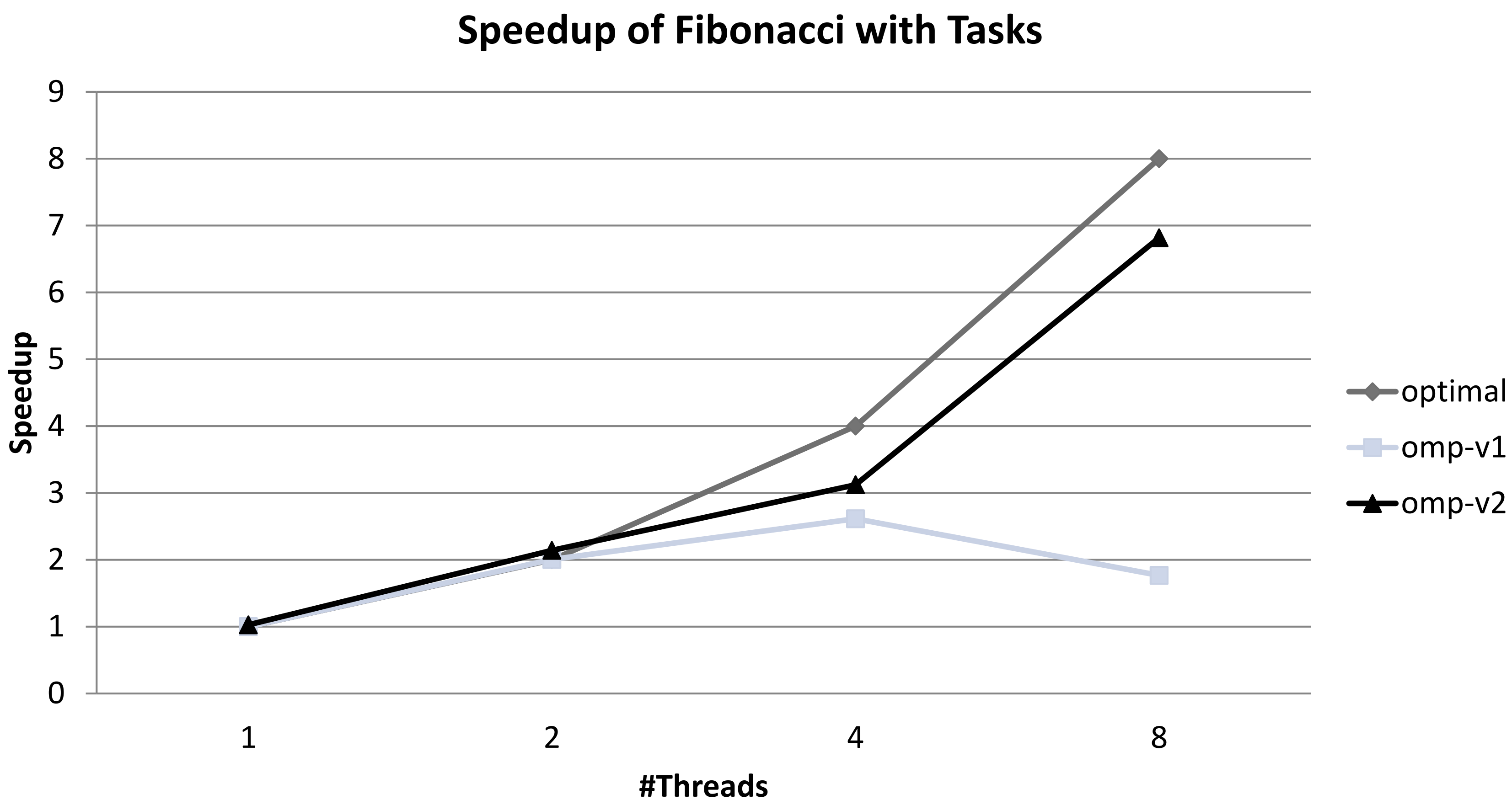
- **Improvement: Don‘t create yet another task once a certain (small enough) n is reached**

```
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)

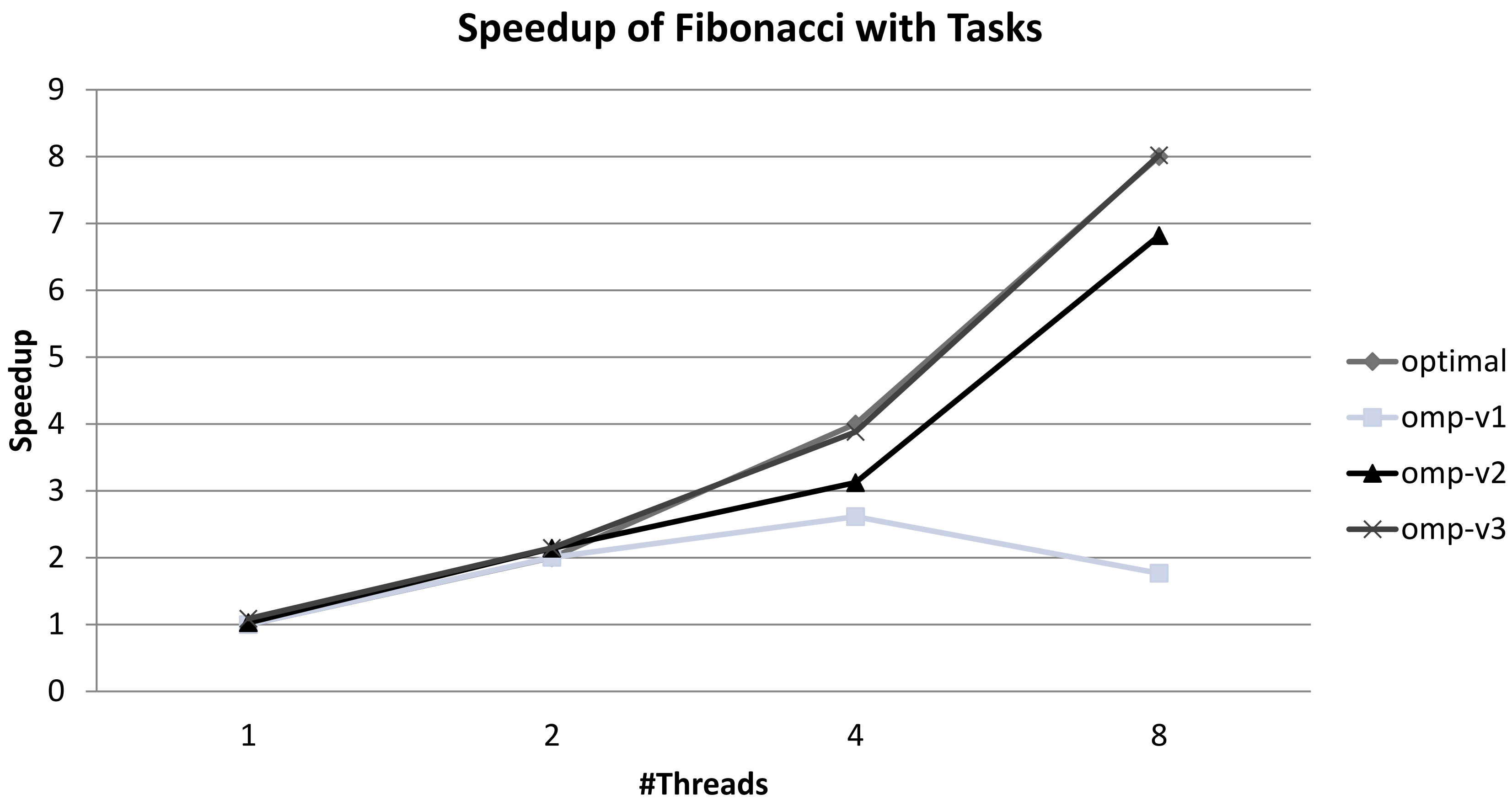
- **Improvement: Skip the OpenMP overhead once a certain n is reached (no issue w/ production compilers)**

```
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 ☺**



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

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)
```

`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.

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.

Concurrent Execution w/ Dep.

- The following allows for more parallelism, as there is one i per thread. Hence, two tasks my be active per thread.

```
void process_in_parallel() {
   #pragma omp parallel
   {
      #pragma omp for
      for (int i = 0; i < T; ++i) {
         #pragma omp task depend(out: i)
            preprocess_some_data(...);
         #pragma omp task depend(in: i)
            do_something_with_data(...);
         #pragma omp task depend(in: i)
            do_something_independent_with_data(...);
      }
   } // end omp parallel
}
```

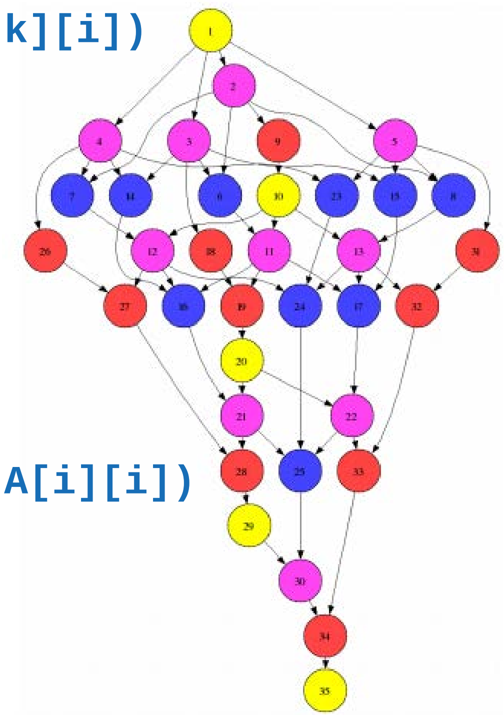
Concurrent Execution w/ Dep.

- The following allows for even more parallelism, as there now can be two tasks active per thread per i-th iteration.

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

„Real“ Task Dependencies

```
void blocked_cholesky( int NB, float A[NB][NB] ) {
   int i, j, k;
   for (k=0; k<NB; k++) {
     #pragma omp task depend(inout:A[k][k])
          spotrf (A[k][k]) ;
     for (i=k+1; i<NT; i++)
       #pragma omp task depend(in:A[k][k]) depend(inout:A[k][i])
            strsm (A[k][k], A[k][i]);
       // update trailing submatrix
       for (i=k+1; i<NT; i++) {
         for (j=k+1; j<i; j++)
           #pragma omp task depend(in:A[k][i],A[k][j])
                            depend(inout:A[j][i])
                sgemm( A[k][i], A[k][j], A[j][i]);
         #pragma omp task depend(in:A[k][i]) depend(inout:A[i][i])
              ssyrk (A[k][i], A[i][i]);
       }
   }
}
```



* image from BSC

The `taskloop` Construct

OpenMP
4.5

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

Example: Sparse CG

```
for (iter = 0; iter < sc->maxIter; iter++) {
    precon(A, r, z);
    vectorDot(r, z, n, &rho);
    beta = rho / rho_old;
    xpay(z, beta, n, p);
    matvec(A, p, q);
    vectorDot(p, q, n, &dot_pq);
    alpha = rho / dot_pq;
    axpy(alpha, p, n, x);
    axpy(-alpha, q, n, r);
    sc->residual = sqrt(rho) * b
    if (sc->residual <= sc->tole
        break;
    rho_old = rho;
}
```

```
void matvec(Matrix *A, double *x, double *y) {
    // ...
#pragma omp parallel for \
            private(i,j,is,ie,j0,y0) \
            schedule(static)
for (i = 0; i < A->n; i++) {
        y0 = 0;
        is = A->ptr[i];
        ie = A->ptr[i + 1];
        for (j = is; j < ie; j++) {
            j0 = index[j];
            y0 += value[j] * x[j0];
        }
        y[i] = y0;
    }
    // ...
}
```

Example: Sparse CG

```
#pragma omp parallel
#pragma omp single
for (iter = 0; iter < sc->maxIter; iter++) {
    precon(A, r, z);
    vectorDot(r, z, n, &rho);
    beta = rho / rho_old;
    xpay(z, beta, n, p);
    matvec(A, p, q);
    vectorDot(p, q, n, &dot_pq);
    alpha = rho / dot_pq;
    axpy(alpha, p, n, x);
    axpy(-alpha, q, n, r);
    sc->residual = sqrt(rho) * b
    if (sc->residual <= sc->tole
        break;
    rho_old = rho;
}
```

```
void matvec(Matrix *A, double *x, double *y) {
    // ...

#pragma omp taskloop private(j,is,ie,j0,y0) \
            grain_size(500)
    for (i = 0; i < A->n; i++) {
        y0 = 0;
        is = A->ptr[i];
        ie = A->ptr[i + 1];
        for (j = is; j < ie; j++) {
            j0 = index[j];
            y0 += value[j] * x[j0];
        }
        y[i] = y0;
    }
    // ...
}
```

One last task...

Yeah!!!

The last slide...

```
#pragma omp parallel for num_threads(2)
for (int i = 0; i < iwomp.num_attendees(); i++)
{
#pragma omp task
   {
      attendee.get_coffee();
   }
}
```