OpenMP Tasking In-depth

Christian Terboven

IT Center, RWTH Aachen University
terboven@itc.rwth-aachen.de

Loops with Tasks

The taskloop Construct

- **Task generating construct: decompose a loop into chunks, create a task for each loop chunk**

```
#pragma omp taskloop [clause[[,] clause]…]
{structured-for-loops}
```

- Where clause is one of:

→ shared(list) → private(list) → firstprivate(list) → lastprivate(list) → default(sh | *pr* | *fp* | none) → reduction(r-id: list) → in_reduction(r-id: list)	**Data Environment**
→ grainsize(grain-size) → num_tasks(num-tasks)	**Chunks/Grain**
→ if(scalar-expression) → final(scalar-expression) → mergeable	**Cutoff Strategies**
→ untied → priority(priority-value)	**Scheduler (R/H)**
→ collapse(n) → nogroup → allocate([allocator:] list)	**Miscellaneous**

Worksharing vs. taskloop constructs (1/2)

```
subroutine worksharing
    integer :: x
    integer :: i
    integer, parameter :: T = 16
    integer, parameter :: N = 1024

    x = 0
!$omp parallel shared(x) num_threads(T)

!$omp do
    do i = 1,N
!$omp atomic
        x = x + 1
!$omp end atomic
    end do
!$omp end do

!$omp end parallel
    write (*,'(A,I0)') 'x = ', x
end subroutine
```

Result: x = 1024

```
subroutine taskloop
    integer :: x
    integer :: i
    integer, parameter :: T = 16
    integer, parameter :: N = 1024

    x = 0
!$omp parallel shared(x) num_threads(T)

!$omp taskloop
    do i = 1,N
!$omp atomic
        x = x + 1
!$omp end atomic
    end do
!$omp end taskloop

!$omp end parallel
    write (*,'(A,I0)') 'x = ', x
end subroutine
```

Result: x = 16384

Worksharing vs. taskloop constructs (2/2)

```
subroutine worksharing
    integer :: x
    integer :: i
    integer, parameter :: T = 16
    integer, parameter :: N = 1024

    x = 0
!$omp parallel shared(x) num_threads(T)

!$omp do
    do i = 1,N
!$omp atomic
        x = x + 1
!$omp end atomic
    end do
!$omp end do

!$omp end parallel
    write (*,'(A,I0)') 'x = ', x
end subroutine
```

Result: x = 1024

```
subroutine taskloop
    integer :: x
    integer :: i
    integer, parameter :: T = 16
    integer, parameter :: N = 1024

    x = 0
!$omp parallel shared(x) num_threads(T)
!$omp single
!$omp taskloop
    do i = 1,N
!$omp atomic
        x = x + 1
!$omp end atomic
    end do
!$omp end taskloop
!$omp end single
!$omp end parallel
    write (*,'(A,I0)') 'x = ', x
end subroutine
```

Result: x = 1024

Taskloop decomposition approaches

- **Clause: grainsize(grain-size)**
  - → Chunks have at least grain-size iterations
  - → Chunks have maximum 2x grain-size

```
int TS = 4 * 1024;
#pragma omp taskloop grainsize(TS)
for ( i = 0; i<SIZE; i+=1) {
   A[i]=A[i]*B[i]*S;
}
```

- Clause: num_tasks(num-tasks)
  - → Create num-tasks chunks
  - → Each chunk must have at least one iteration

```
int NT = 4 * omp_get_num_threads();
#pragma omp taskloop num_tasks(NT)
for ( i = 0; i<SIZE; i+=1) {
   A[i]=A[i]*B[i]*S;
}
```

- If none of previous clauses is present, the *number of chunks* and the *number of iterations per chunk* is implementation defined
- Additional considerations:
  - → The order of the creation of the loop tasks is unspecified
  - → Taskloop creates an implicit taskgroup region; **nogroup** → no implicit taskgroup region is created

Sudoku

Parallel Brute-force Sudoku

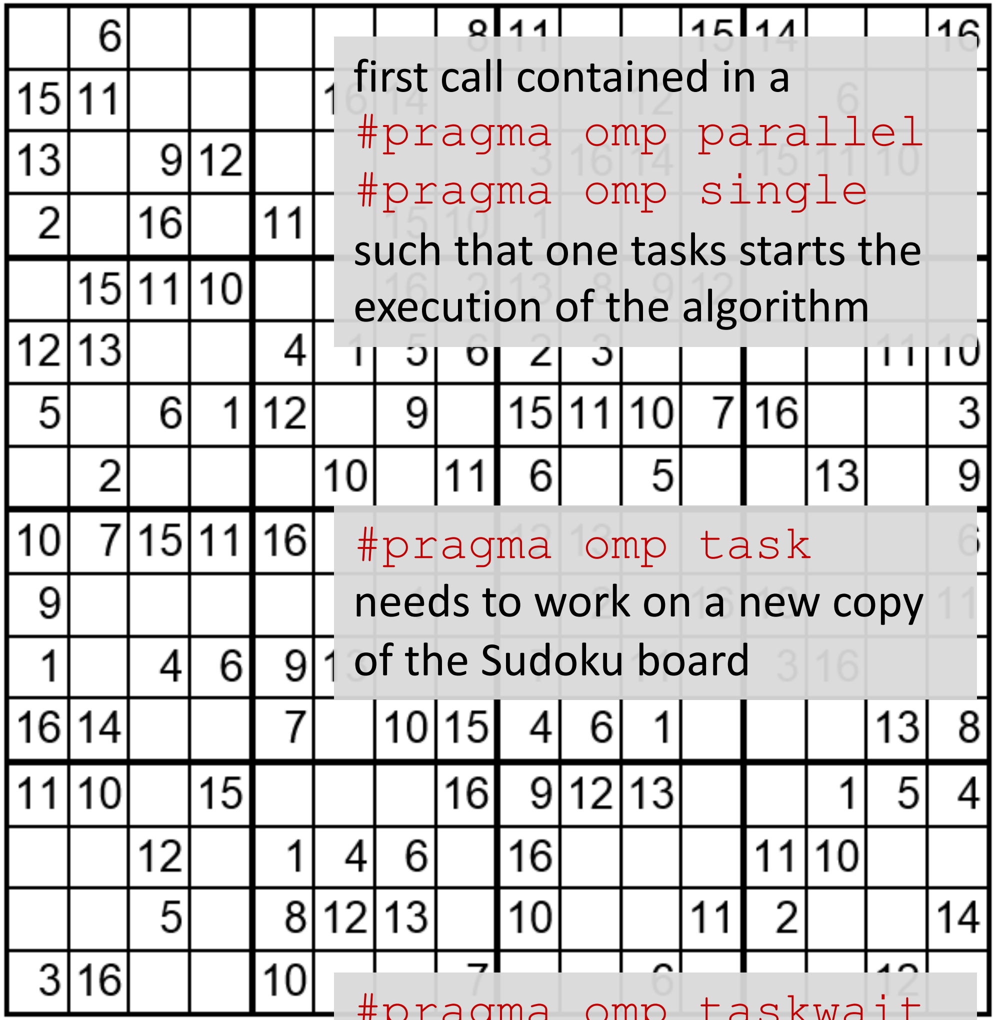
- **Lets solve Sudoku puzzles with brute multi-core force**

	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

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

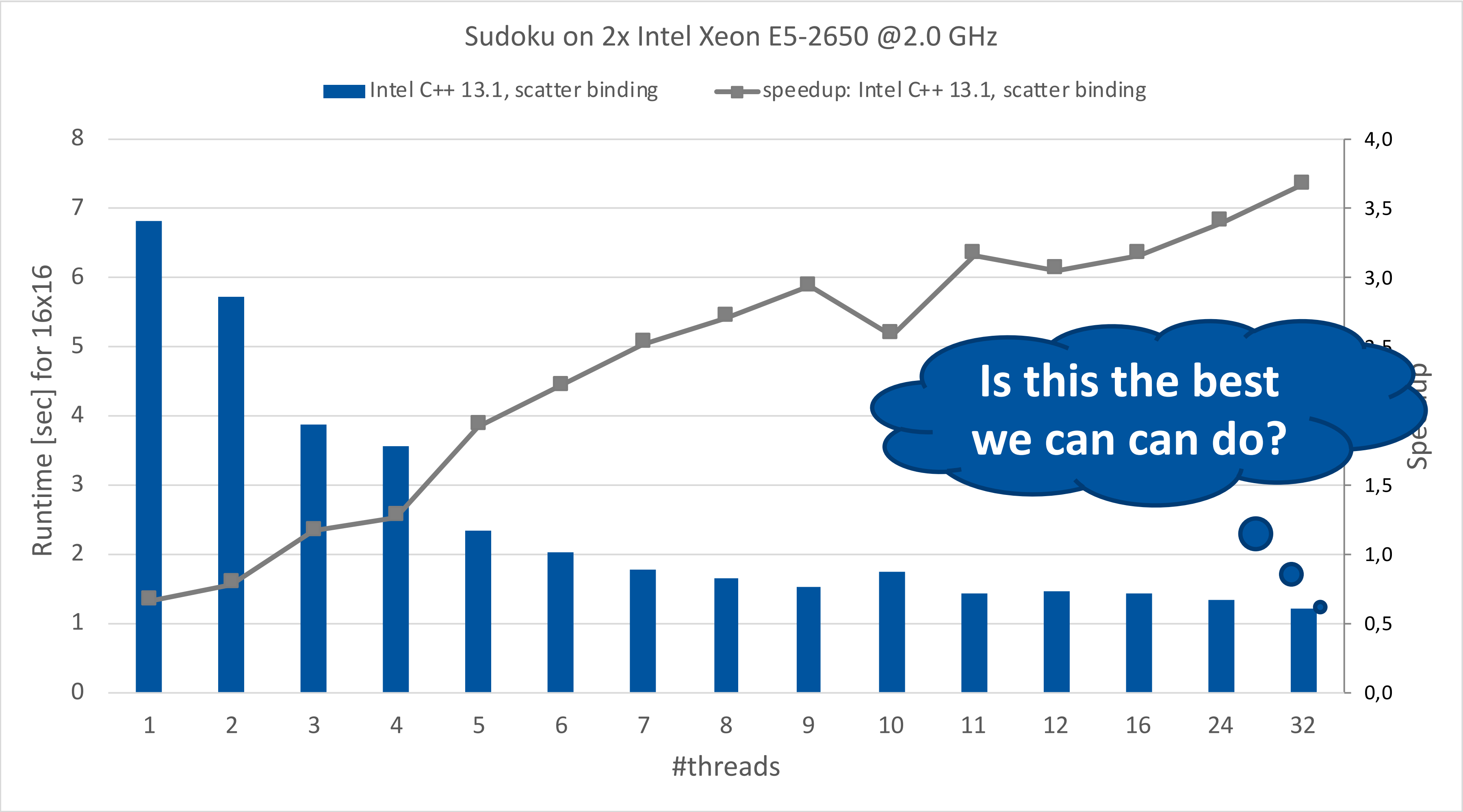
- The actual implementation

```
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

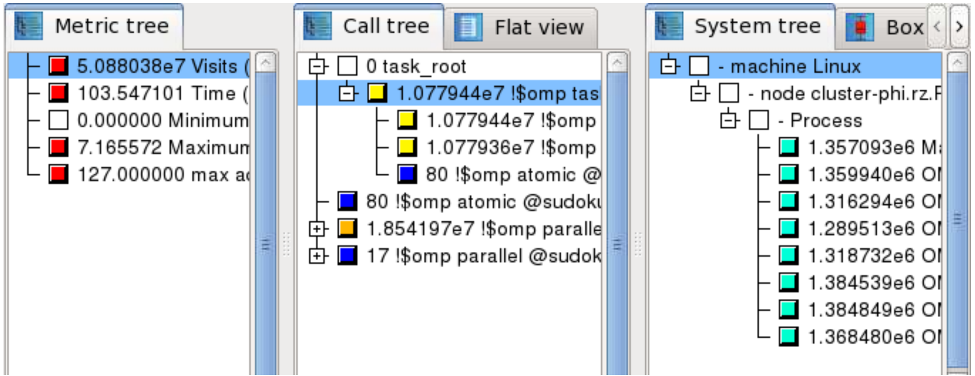
Performance Evaluation



Is this the best we can can do?

Performance Analysis

Event-based profiling gives a good overview :



Metric tree

- 5.088038e7 Visits (
- 103.547101 Time (
- 0.000000 Minimum
- 7.165572 Maximum
- 127.000000 max a

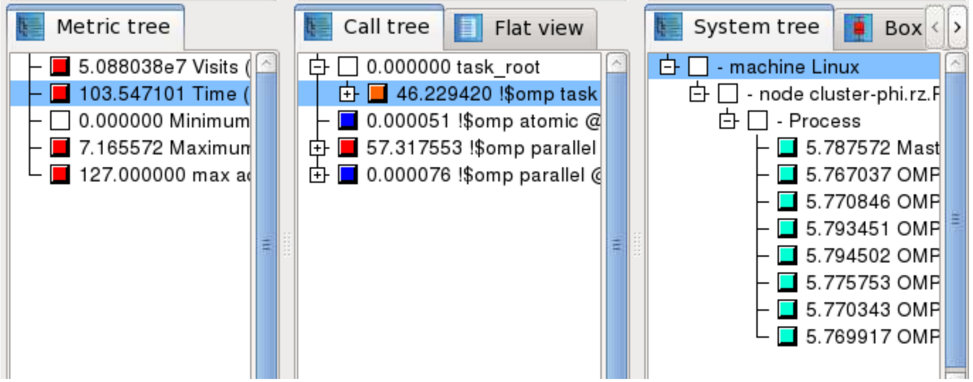
Call tree | Flat view

- 0 task_root
  - 1.077944e7 !$omp tas
    - 1.077944e7 !$omp
    - 1.077936e7 !$omp
    - 80 !$omp atomic @
  - 80 !$omp atomic @sudoku
  - 1.854197e7 !$omp paralle
  - 17 !$omp parallel @sudok

System tree | Box

- machine Linux
  - node cluster-phi.rz.F
    - Process
      - 1.357093e6 Ma
      - 1.359940e6 OI
      - 1.316294e6 OI
      - 1.289513e6 OI
      - 1.318732e6 OI
      - 1.384539e6 OI
      - 1.384849e6 OI
      - 1.368480e6 OI

Every thread is executing ~1.3m tasks…



Metric tree

- 5.088038e7 Visits (
- 103.547101 Time (
- 0.000000 Minimum
- 7.165572 Maximum
- 127.000000 max a

Call tree | Flat view

- 0.000000 task_root
  - 46.229420 !$omp task
  - 0.000051 !$omp atomic @
  - 57.317553 !$omp parallel
  - 0.000076 !$omp parallel @

System tree | Box

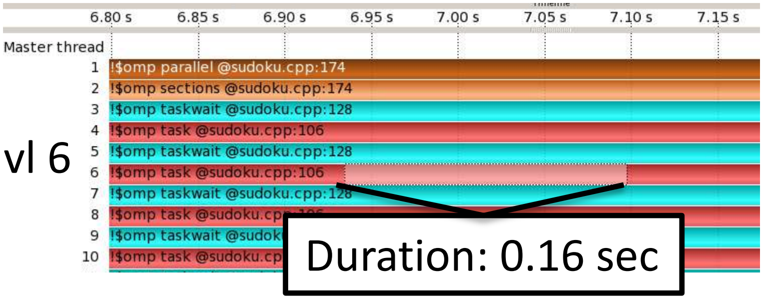
- machine Linux
  - node cluster-phi.rz.F
    - Process
      - 5.787572 Mast
      - 5.767037 OMP
      - 5.770846 OMP
      - 5.793451 OMP
      - 5.794502 OMP
      - 5.775753 OMP
      - 5.770343 OMP
      - 5.769917 OMP

… in ~5.7 seconds.

=> average duration of a task is ~4.4 µs

Tracing gives more details:

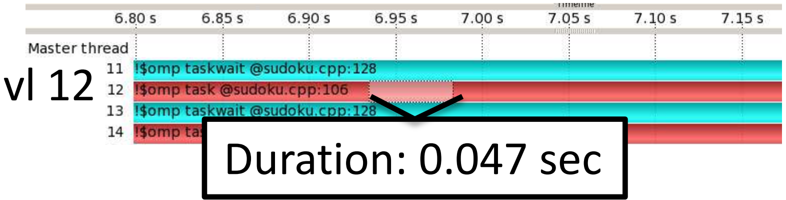
lvl 6



1 !$omp parallel @sudoku.cpp:174
2 !$omp sections @sudoku.cpp:174
3 !$omp taskwait @sudoku.cpp:128
4 !$omp task @sudoku.cpp:106
5 !$omp taskwait @sudoku.cpp:128
6 !$omp task @sudoku.cpp:106
7 !$omp taskwait @sudoku.cpp:128
8 !$omp task @sudoku.cp
9 !$omp taskwait @sudok
10 !$omp task @sudoku.cp

Duration: 0.16 sec

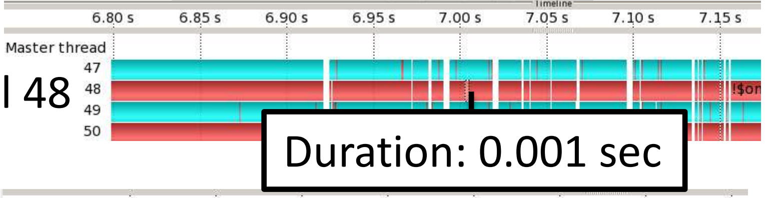
lvl 12



11 !$omp taskwait @sudoku.cpp:128
12 !$omp task @sudoku.cpp:106
13 !$omp taskwait @sudoku.cpp:128
14 !$omp ta

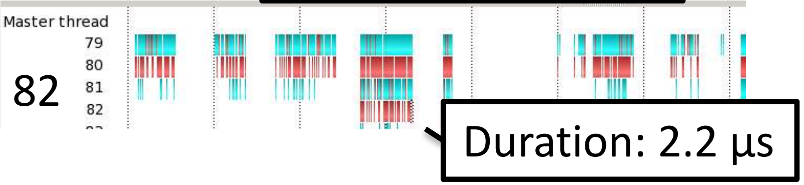
Duration: 0.047 sec

lvl 48



Duration: 0.001 sec

lvl 82

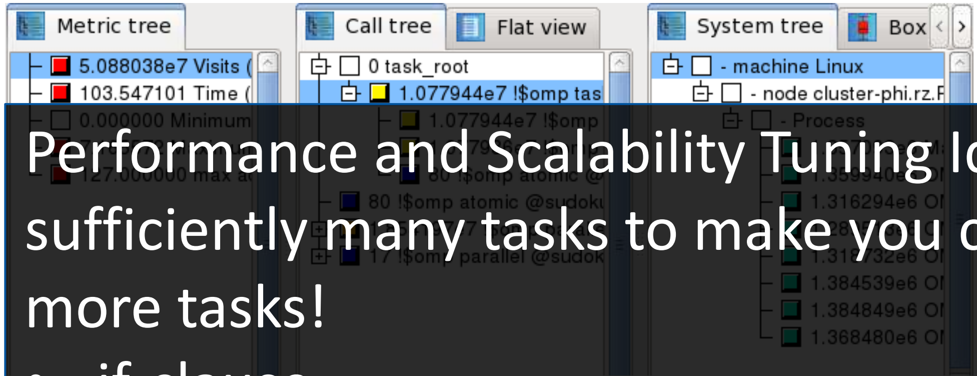


Duration: 2.2 µs

Tasks get much smaller down the call-stack.

Performance Analysis

Event-based profiling gives a good overview :

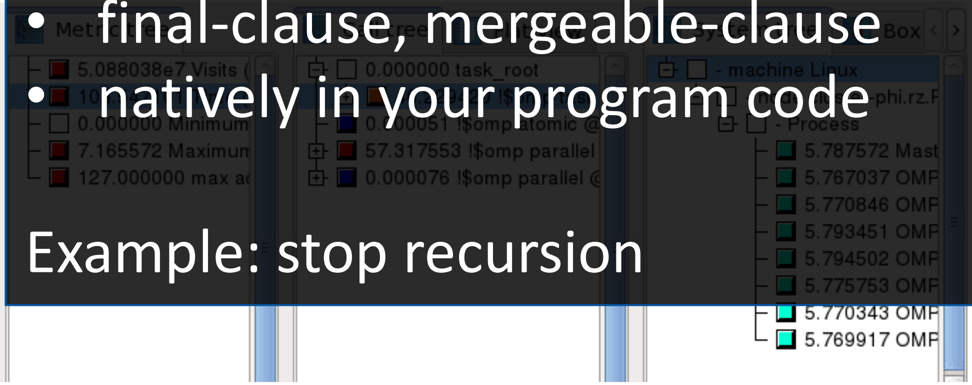


5.088038e7 Visits (
103.547101 Time (
0.000000 Minimum
127.000000 max a

0 task_root
1.077944e7 !$omp tas
1.077944e7 !$omp
80 !$omp atomic @
80 !$omp atomic @sudoku
17 !$omp parallel @sudok

- machine Linux
- node cluster-phi.rz.R
- Process
1.359940e6 OI
1.316294e6 OI
1.318732e6 OI
1.384539e6 OI
1.384849e6 OI
1.368480e6 OI

Every thread is executing ~1.3m tasks…



5.088038e7 Visits (
10
0.000000 Minimum
7.165572 Maximum
127.000000 max ac

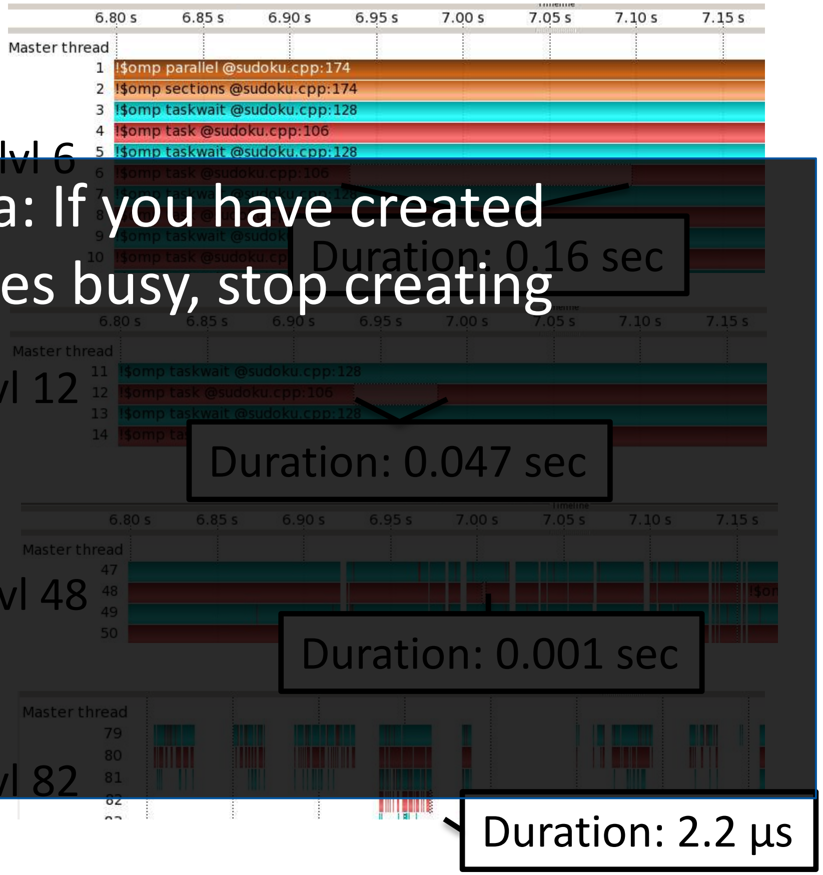
0.000000 task_root
0.00051 !$omp atomic @
57.317553 !$omp parallel
0.000076 !$omp parallel @

- machine Linux
- node cluster-phi.rz.R
- Process
5.787572 Mast
5.767037 OMP
5.770846 OMP
5.793451 OMP
5.794502 OMP
5.775753 OMP
5.770343 OMP
5.769917 OMP

… in ~5.7 seconds.

=> average duration of a task is ~4.4 µs

Tracing gives more details:



6.80 s
6.85 s
6.90 s
6.95 s
7.00 s
7.05 s
7.10 s
7.15 s
Master thread
1 !$omp parallel @sudoku.cpp:174
2 !$omp sections @sudoku.cpp:174
3 !$omp taskwait @sudoku.cpp:128
4 !$omp task @sudoku.cpp:106
5 !$omp taskwait @sudoku.cpp:128
6 !$omp task @sudoku.cpp:106
11 !$omp taskwait @sudoku.cpp:128
12 !$omp task @sudoku.cpp:106
13 !$omp taskwait @sudoku.cpp:128
14 !$omp ta
47
48
49
50
79
80
81
82

lvl 6

Duration: 0.16 sec

lvl 12

Duration: 0.047 sec

lvl 48

Duration: 0.001 sec

lvl 82

Duration: 2.2 µs

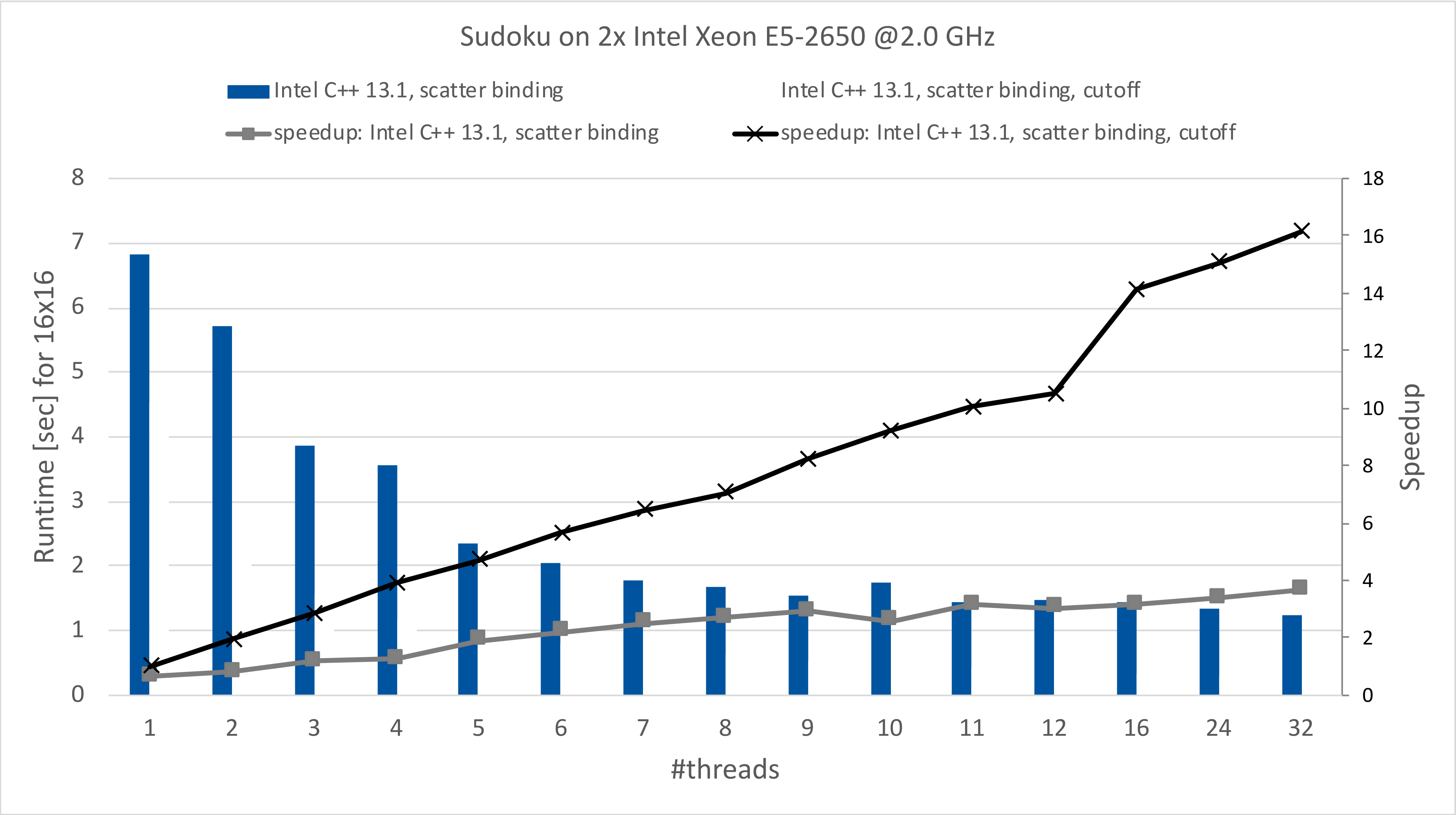
Tasks get much smaller down the call-stack.

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++

```
#pragma omp taskyield
```

Fortran

```
!$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)

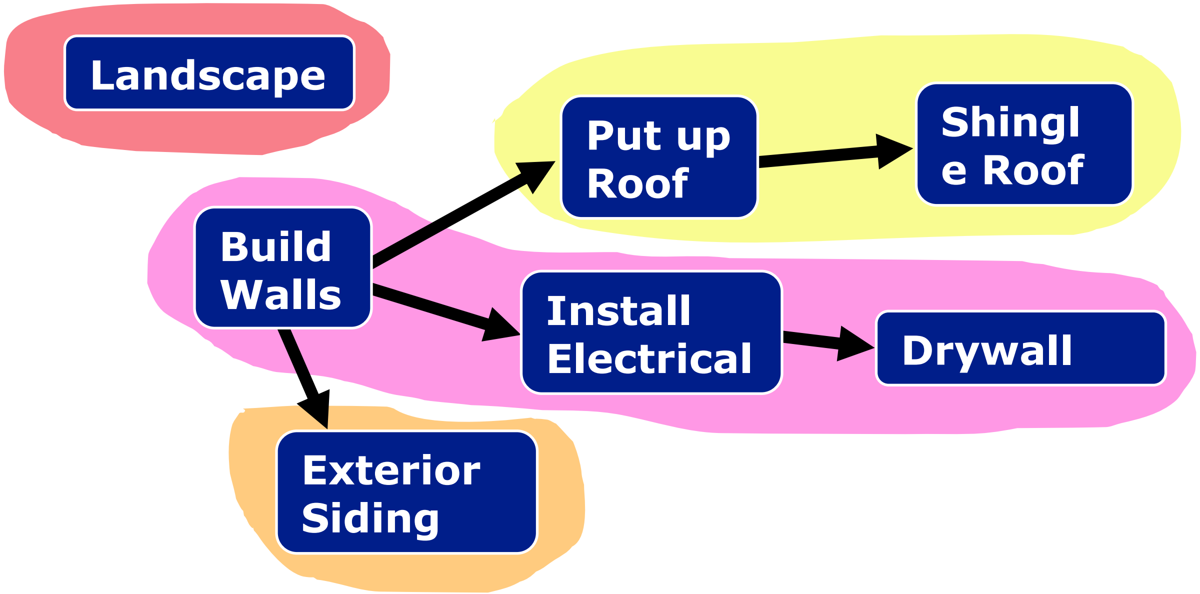
```
#include <omp.h>

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

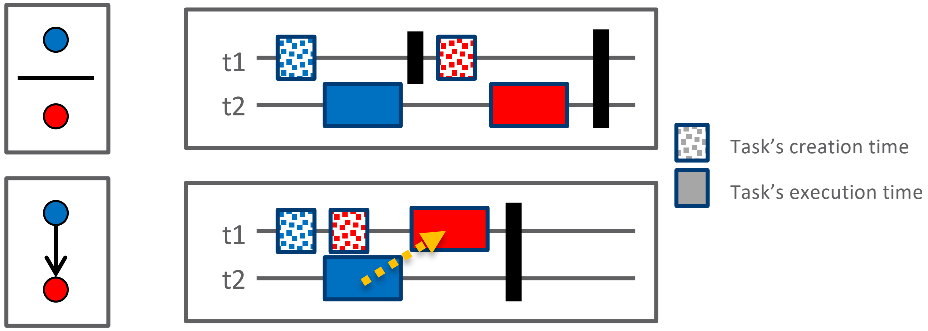
```
int x = 0;
#pragma omp parallel
#pragma omp single
{
🔵 #pragma omp task
   std::cout << x << std::endl;

   #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++;
}
```

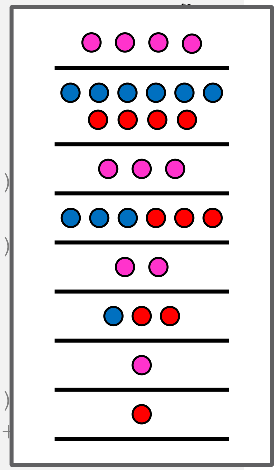
Task wait

Dependencies



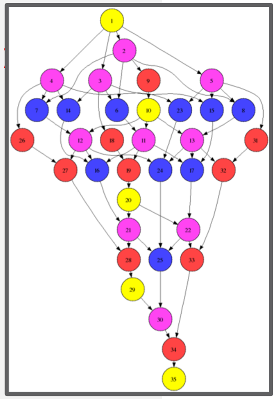
More Complex Example: Cholesky Factorization

```
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?