

Programming OpenMP

Cut-off strategies

Christian Terboven



Example: Sudoku revisited

Parallel Brute-force Sudoku

- This parallel algorithm finds all valid solutions

	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

```
#pragma omp parallel
#pragma omp single
such that one task starts the
execution of the algorithm
```

- (2) Try all numbers:

- (2 a) Check Sudoku

- If invalid: skip

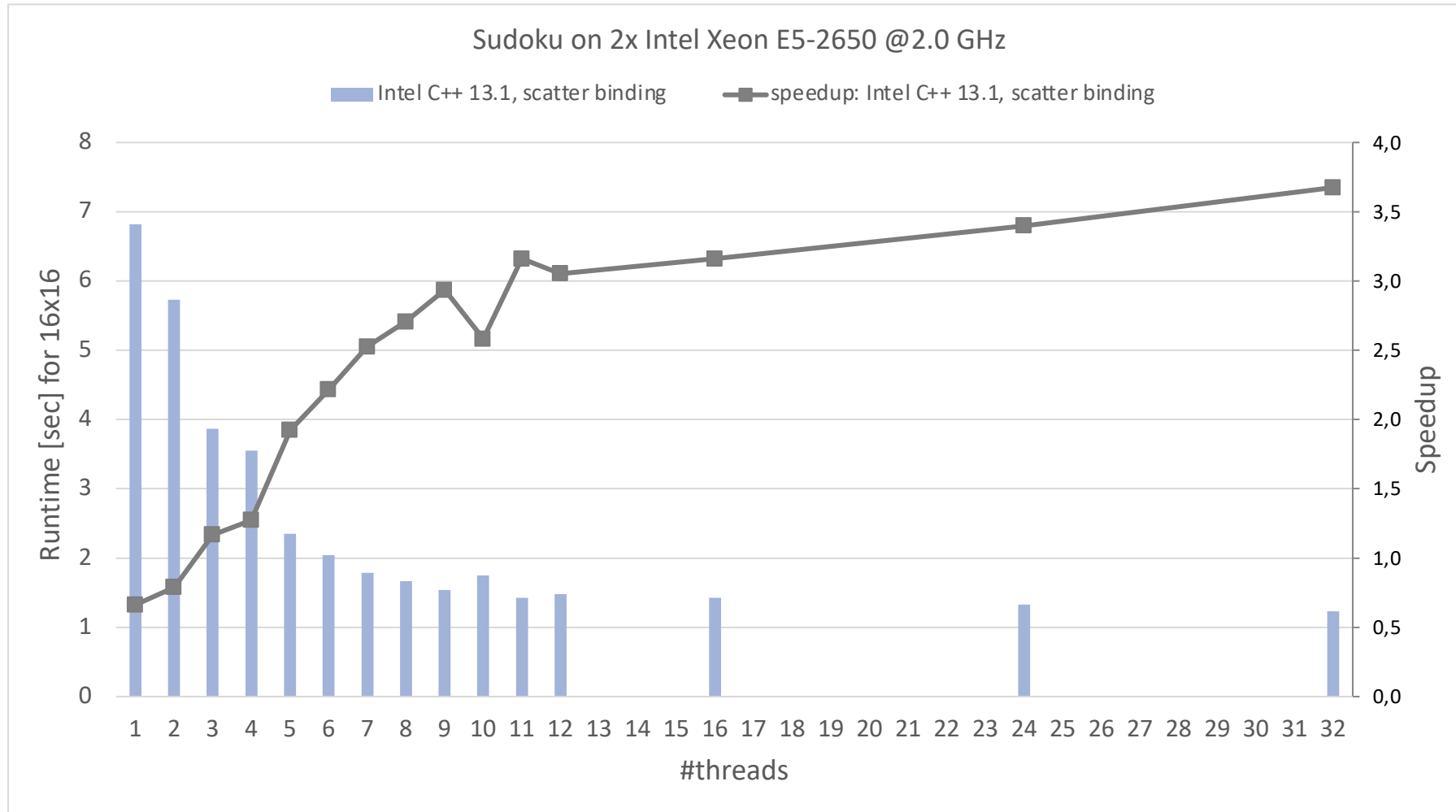
- If valid: Go to next field

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

- Wait for completion

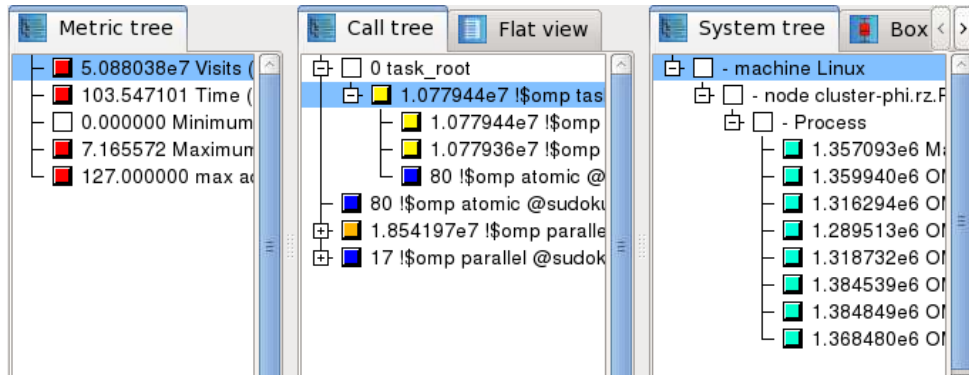
```
#pragma omp taskwait
wait for all child tasks
```

Performance Evaluation

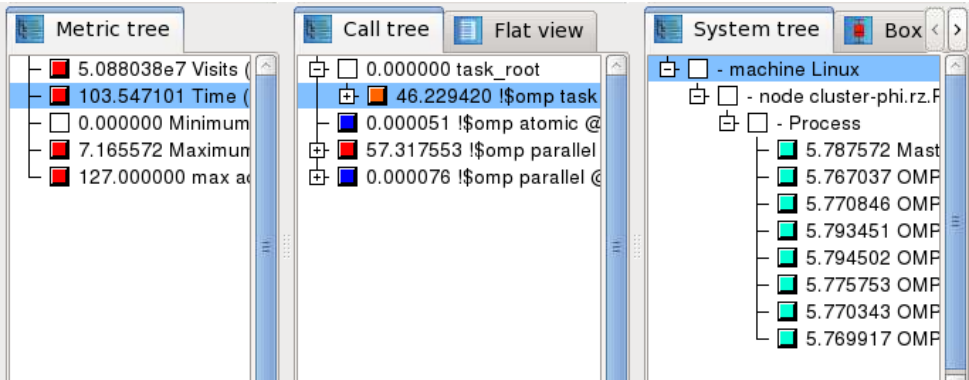


Performance Analysis

Event-based profiling provides a good overview :



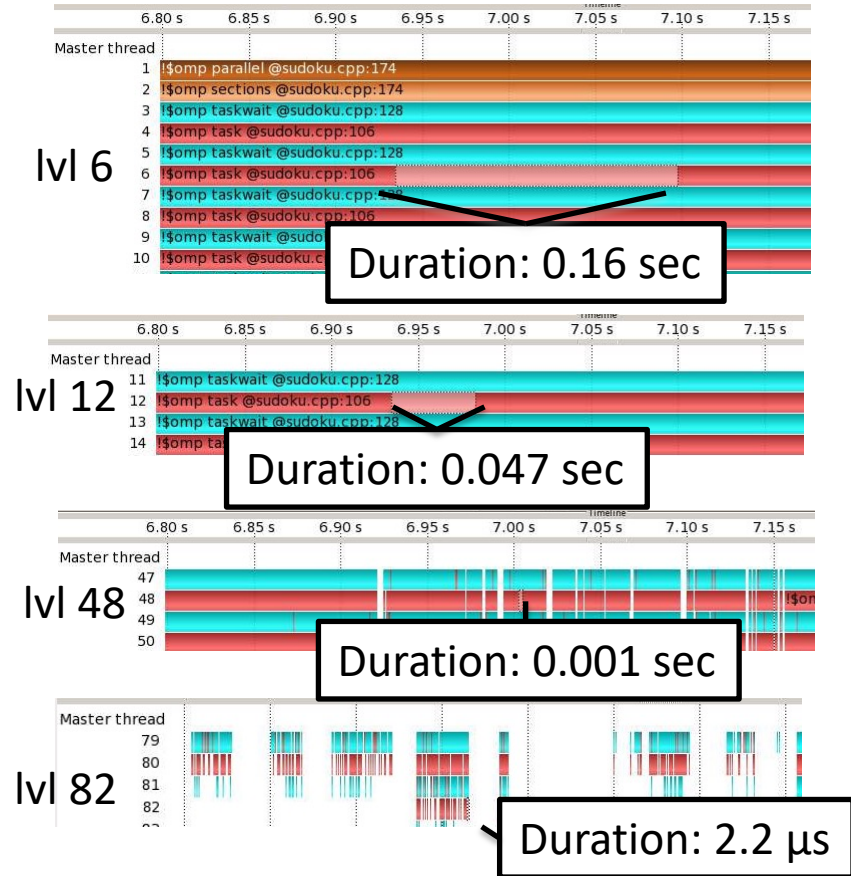
Every thread is executing ~ 1.3 m tasks...



... in ~ 5.7 seconds.

=> average duration of a task is $\sim 4.4 \mu\text{s}$

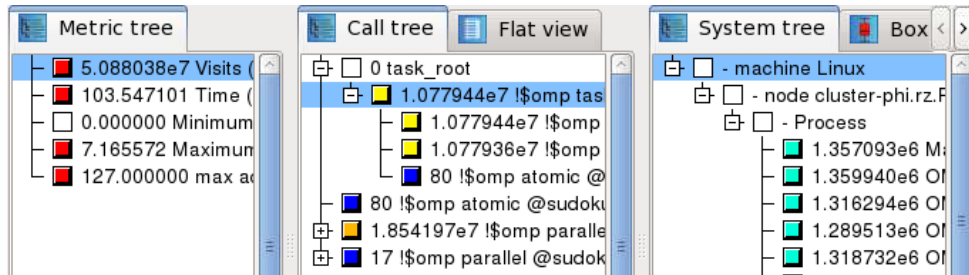
Tracing provides more details:



Tasks get much smaller down the call-stack.

Performance Analysis

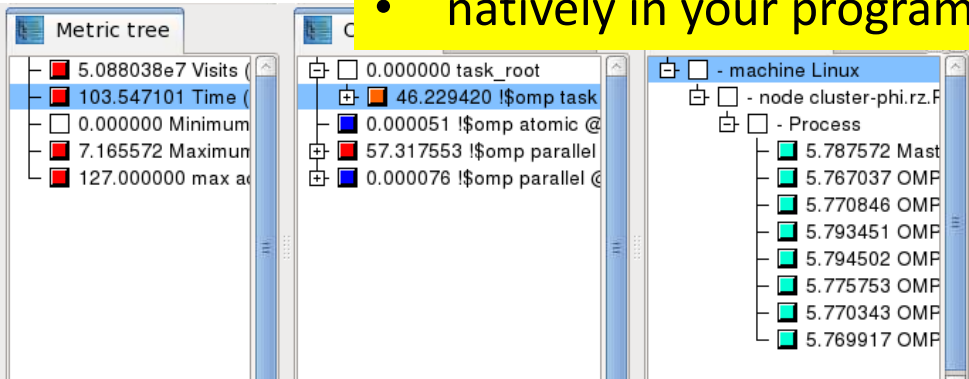
Event-based profiling provides a good overview :



If you have enough parallelism, stop creating more tasks!!

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

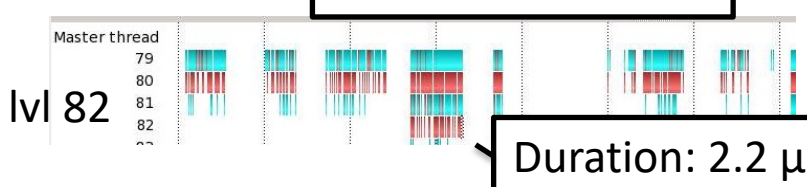
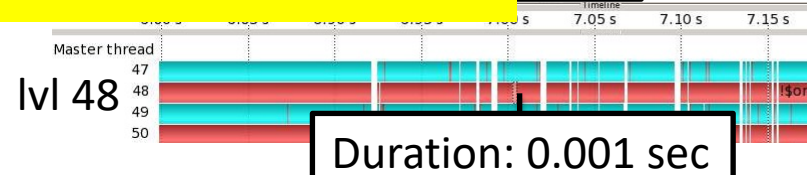
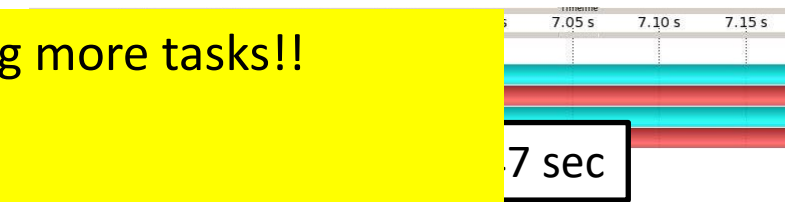
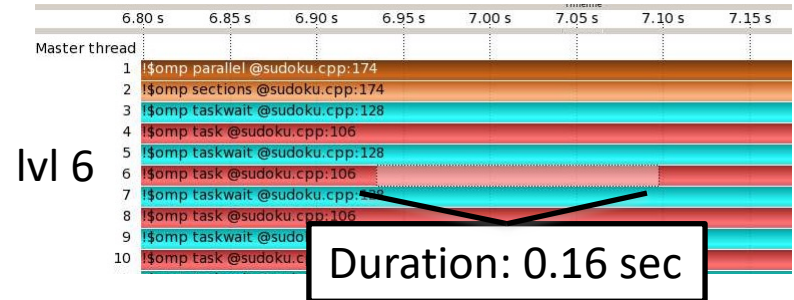
Every thread i



... in ~5.7 seconds.

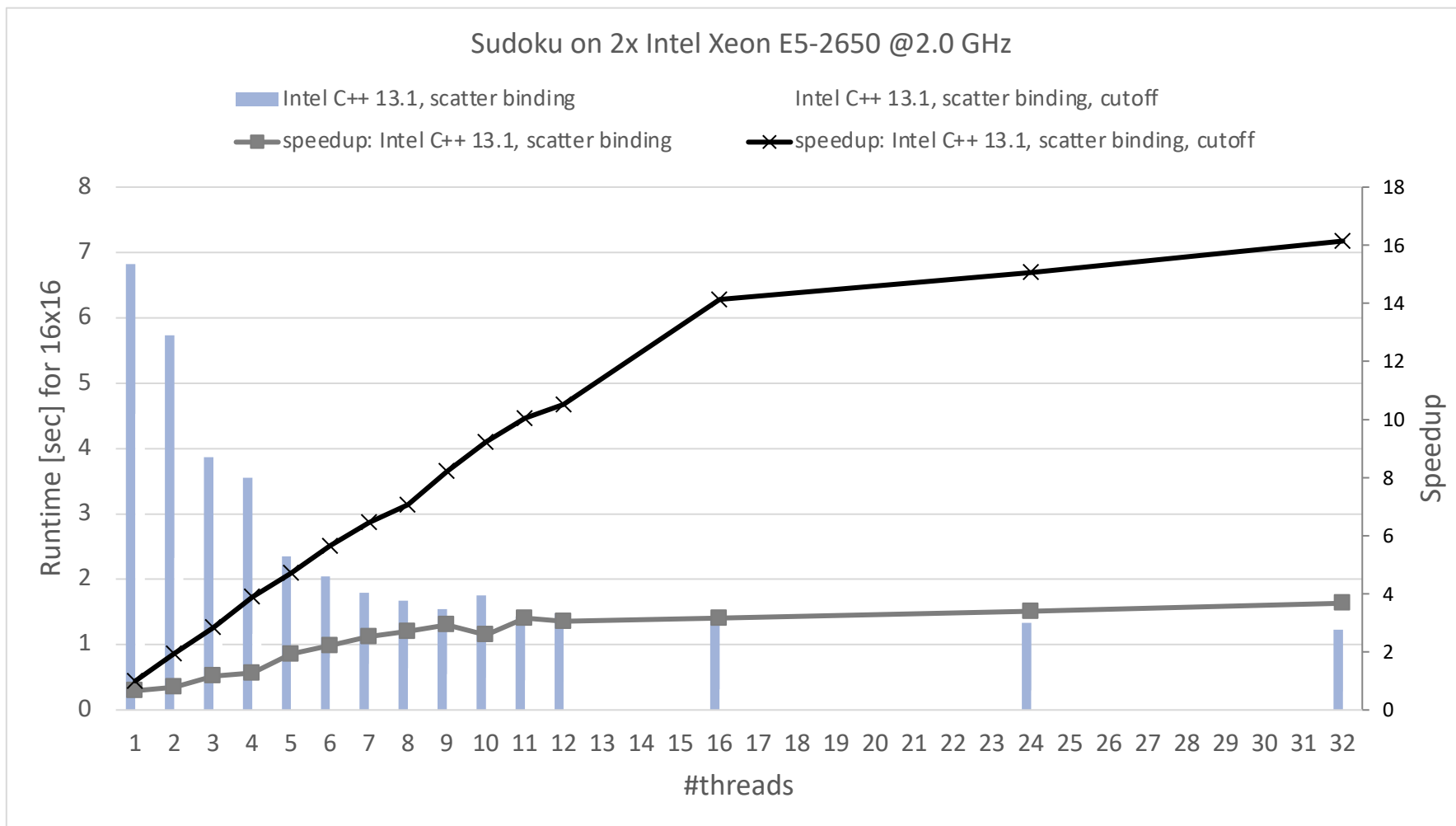
=> average duration of a task is ~4.4 μ s

Tracing provides more details:



Tasks get much smaller down the call-stack.

Performance Evaluation (with cutoff)



Improving Tasking Performance: Cutoff clauses and strategies

The `if` clause

- Rule of thumb: the `if (expression)` clause as a “switch off” mechanism
 - Allows lightweight implementations of task creation and execution but it reduces the parallelism

- If the expression of the `if` clause evaluates to `false`

- the encountering task is suspended
- the new task is executed immediately (task dependences are respected!!)
- the encountering task resumes its execution once the new task is completed
- This is known as *undeferred task*

```
int foo(int x) {  
    printf("entering foo function\n");  
    int res = 0;  
    #pragma omp task shared(res) if(false)  
    {  
        res += x;  
    }  
    printf("leaving foo function\n");  
}
```

Really useful to debug tasking applications!

- Even if the expression is `false`, data-sharing clauses are honored

The final clause

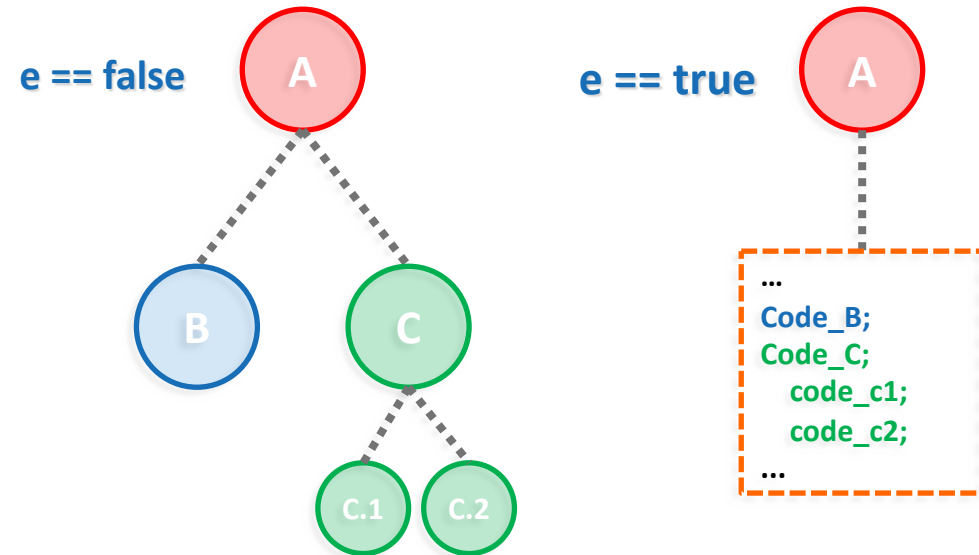
- The `final(expression)` clause

- Nested tasks / recursive applications
- allows to avoid future task creation → reduces overhead but also reduces parallelism

- If the expression of the final clause evaluates to `true`

- The new task is created and executed normally but in its context all tasks will be executed immediately by the same thread (*included tasks*)

```
#pragma omp task final(e)
{
  #pragma omp task
  { ... }
  #pragma omp task
  { ... #C.1; #C.2 ... }
  #pragma omp taskwait
}
```



- Data-sharing clauses are honored too!

The mergeable clause

■ The `mergeable` clause

→ Optimization: get rid of “data-sharing clauses are honored”

→ This optimization can only be applied in *undeferred* or *included tasks*

■ A Task that is annotated with the `mergeable` clause is called a *mergeable task*

→ A task that may be a *merged task* if it is an *undeferred task* or an *included task*

■ A *merged task* is:

→ A task for which the data environment (inclusive of ICVs) may be the same as that of its generating task region

■ A good implementation could execute a merged task without adding any OpenMP-related overhead

Unfortunately, there are no OpenMP commercial implementations taking advantage of `final` `neither` `mergeable` =(