

# OpenMP Tasking, Part 2: Performant Tasking

Xavier Teruel



**Barcelona  
Supercomputing  
Center**

*Centro Nacional de Supercomputación*

# PART 1 Outline: Fundamentals (July 30th)

## The task construct

- Deferring (or not) a unit of work (executable for any member of the team)

```
#pragma omp task [clause[,, clause]...]
{structured-block}
```

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

- Where clause is one of:

→ private(list)	Data Environment
→ firstprivate(list)	
→ shared(list)	
→ default(shared   none)	
→ in_reduction(r-id: list)	
→ allocate([allocator:] list)	Miscellaneous
→ detach(event-handler)	

### Tasking Use Cases

OpenMP

**Tasking use case: Cholesky (synchronization)**

```
void cholesky(int ta, int tb, double** A){int i, j, k;
    for (i = 0; i < tb; i++) {
        for (j = 0; j < tb; j++) {
            if (i == j) {
                for (k = 0; k < tb; k++) {
                    if (k != i) {
                        #pragma omp task
                        trsm(k, k, TA(i), TB(j));
                    }
                }
            }
        }
    }
    #pragma omp taskwait
    // Update trailing matrix
    for (i = 0; i < tb; i++) {
        for (j = 0; j < tb; j++) {
            if (i == j) {
                for (k = 0; k < tb; k++) {
                    if (k != i) {
                        #pragma omp task
                        systm(k, k, TA(i), TB(j));
                    }
                }
            }
        }
    }
    #pragma omp taskwait
}
```

Complex synchronization patterns  
→ Splitting computational phases  
→ taskwait or taskgroup  
→ Needs complex code analysis  
→ Improving programmability  
→ OpenMP dependences  
→ It also improves compositability

OpenMP

OpenMP

**Tasking use case: Fibonacci (recursion)**

```
int comp_fib_numbers ( int n ) {
    int f0, f1, fn;
    if ( n == 0 || n == 1 ) return(n);
    #pragma omp task shared(f0)
    f0 = comp_fib_numbers(n-1);
    #pragma omp task shared(f1)
    f1 = comp_fib_numbers(n-2);
    #pragma omp taskwait
    return(f0 + f1);
}
```

Functionally correct  
Poor performance  
→ Tasks are very fine-grained  
→ Too much parallelism?  
→ Improving programmability  
→ Cut-off strategies

OpenMP

**Tasking use case: saxpy (blocking)**

```
for ( i = 0; i < SIZE; i+=2 ) {
    A[i]=A[i]*B[i];
}
for ( i = 0; i < SIZE; i+=2 ) {
    B[i] = B[i] - A[i];
}
for ( i = 1; i < SIZE; i+=2 ) {
    A[i]=A[i]*B[i];
}
for ( i = 1; i < SIZE; i+=2 ) {
    B[i] = B[i] - A[i];
}

#pragma omp parallel
#pragma omp single
for ( i = 0; i < SIZE; i+=2 ) {
    #pragma omp task private(i)
    #pragma omp task private(A,B)
    #pragma omp task private(i1)
    A[i]=A[i]*B[i];
    A[i1]=A[i1]*B[i1];
}
```

Difficult to determine grain  
→ 1 single iteration → to fine  
→ whole loop → no parallelism  
→ Manually transform the code  
→ blocking techniques  
→ Improving programmability  
→ OpenMP taskloop

OpenMP

## Pre-determined data-sharing attributes

- threadprivate variables are threadprivate (1)
- dynamic storage duration objects are shared (malloc, new,... ) (2)
- static data members are shared (3)
- variables declared inside the construct

```
#pragma omp task
{
    int x = MN;
    // Scope of x: private
}
```

- static storage duration variables are shared (4)
- automatic storage duration variables are private (5)
- the loop iteration variable(s)...

```
int A[SIZE];
#pragma omp threadprivate(A)
// ...
#pragma omp task
{
    // A: threadprivate
}
```

```
int *p;
p = malloc(sizeof(float)*SIZE);
#pragma omp task
{
    // *p: shared
}
```

```
class foo {
    static int s = MN;
};

#pragma omp task
{
    foo::s; // s@foo: shared
}
```

## 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}
```

```
!$omp taskloop [clause[,, clause]...]
...structured-do-loops...
!$omp end taskloop
```

- Where clause is one of:

→ shared(list)	Data Environment
→ private(list)	
→ firstprivate(list)	
→ lastprivate(list)	
→ default(sh   pr   fp   none)	
→ reduction(r-id: list)	Scheduler (R/H)
→ in_reduction(r-id: list)	
→ grainsize(grain-size)	
→ num_tasks(num-tasks)	Chunks/Grain
→ nogroup	



# Slides and Exercises

■ <https://tinyurl.com/openmp-umt-tasking>

2021 Tasking UMT

Nombre

- OpenMP-UMT-Exercises.tar.gz
- OpenMP-UMT-Tasking-1.pdf
- OpenMP-UMT-Tasking-2.pdf



# PART 2 Outline: Performant Tasking

## ■ Cutting-off strategies

- Motivation
- If-, final- and mergeable- clauses

## ■ Task dependences

- Motivation
- What's in the spec: depend clause, deps on taskwait and dependable objects
- The data-flow philosophy

## ■ More on performant tasking

\* These slides are part of the tutorial “**Mastering Tasking with OpenMP**”; presented at SC and ISC conferences. Authors: Christian Terboven, Michael Klemm, Xavier Teruel, and Bronis R. de Supinski. All of them members of the OpenMP Language Committee.

# Cutoff clauses and strategies

# 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	
	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		

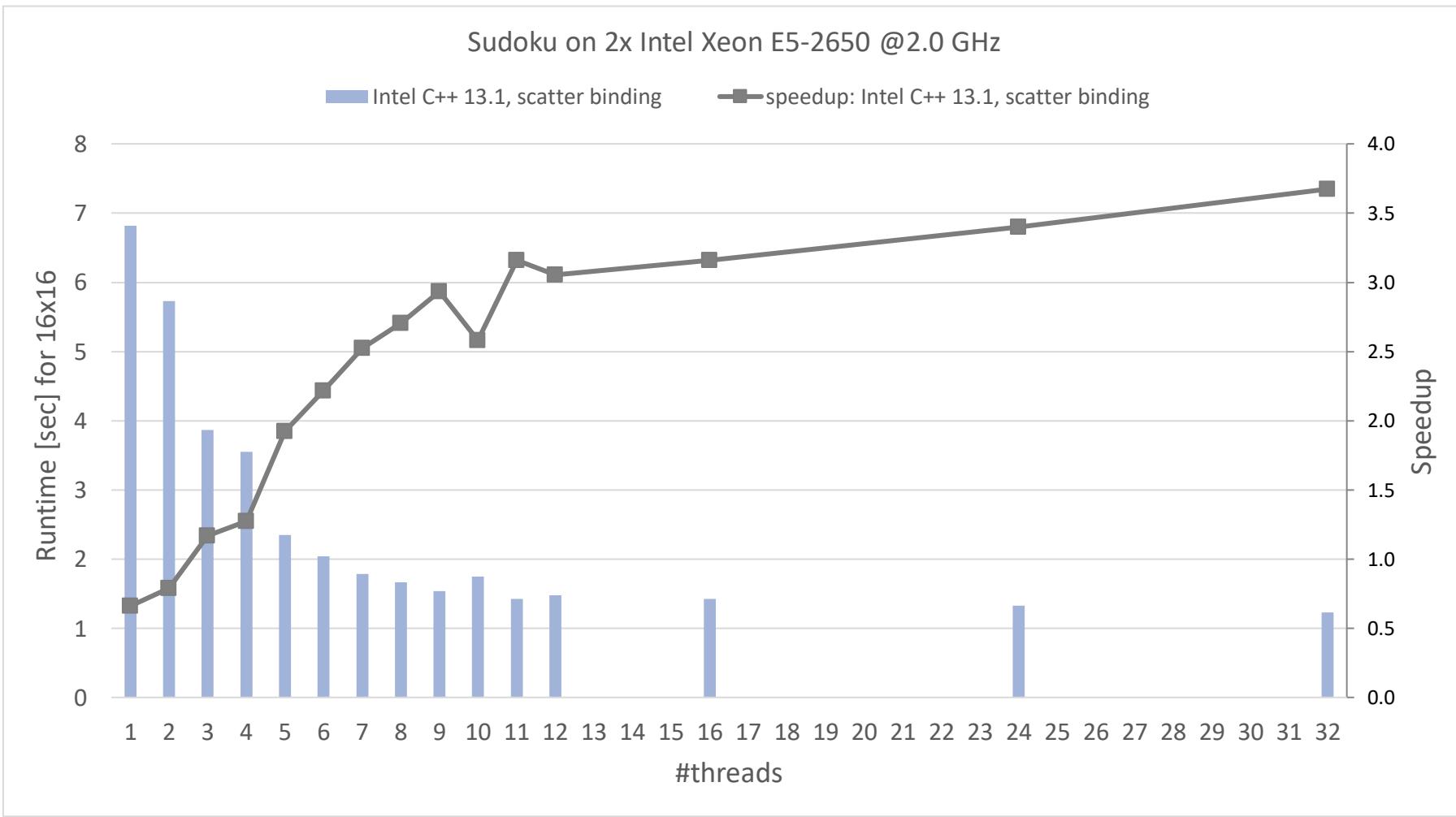
- (1) Search an empty field
- (2) Try all numbers:
  - Check Sudoku
  - If invalid: skip
  - If valid: Go to next field
- (3) Wait for completion

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

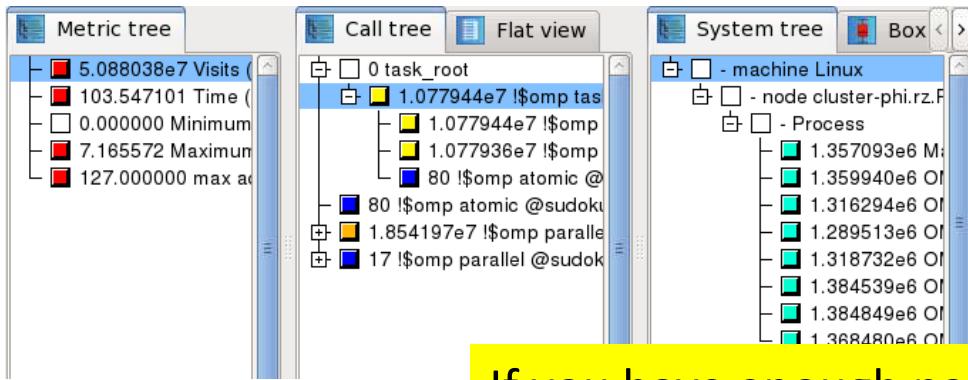
`#pragma omp taskwait`  
wait for all child tasks

# Performance Evaluation



# Performance Analysis

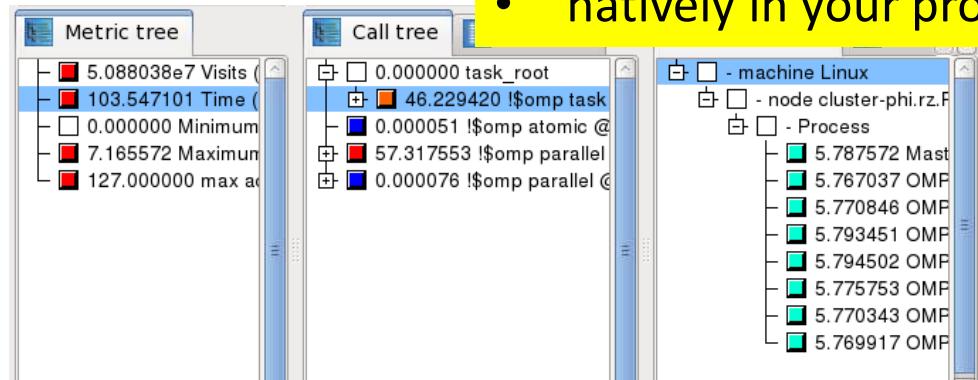
Event-based profiling :



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

Every thread is ex

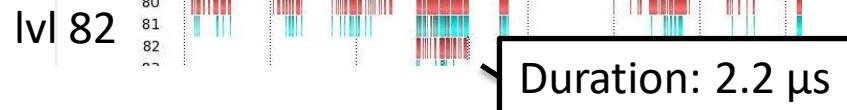
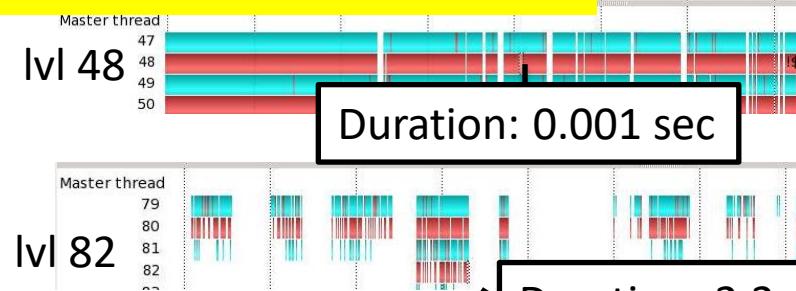
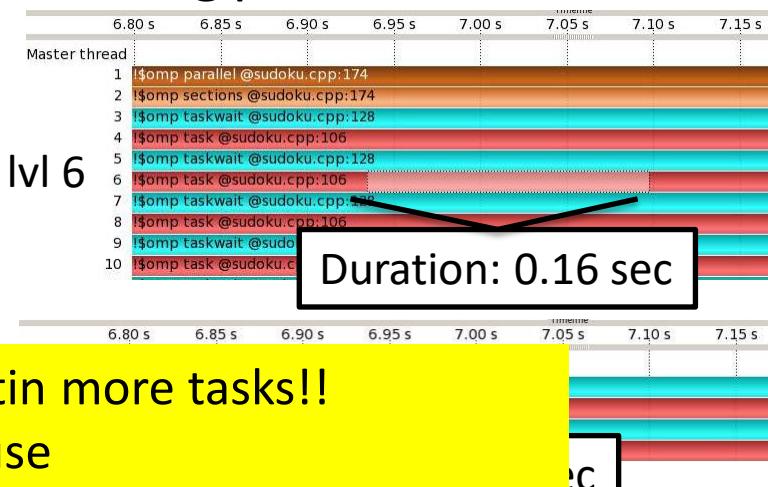
- if-clause, final-clause, mergeable-clause
- natively in your program code



... in ~5.7 seconds.

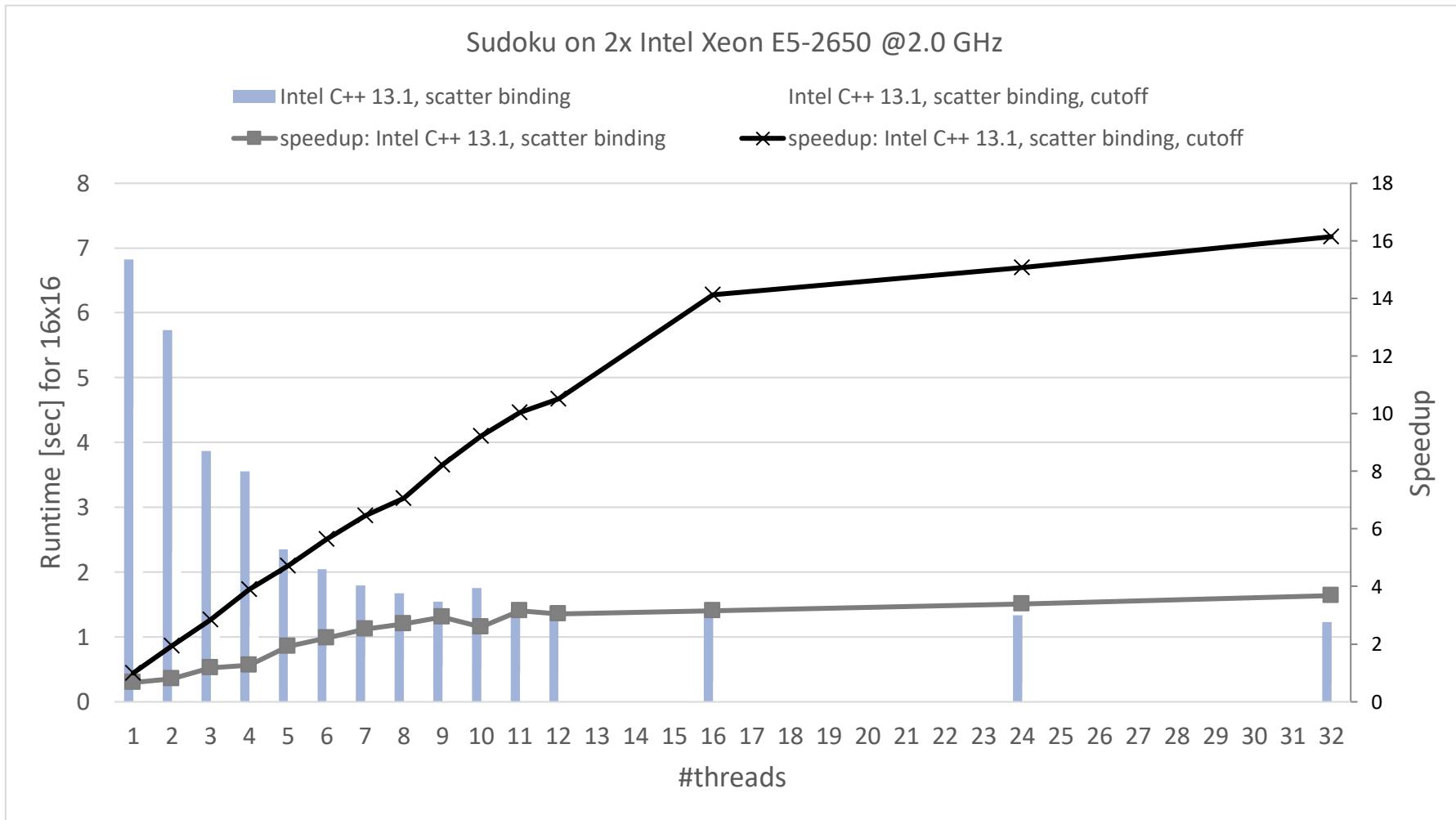
average duration of a task is ~4.4  $\mu$ s

Tracing provides more details:



Tasks get much smaller down the call-stack.

# Performance Evaluation (with cutoff)



# 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 *undelayed task*
- Even if the expression is `false`, data-sharing clauses are honored

```
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!

# The final clause

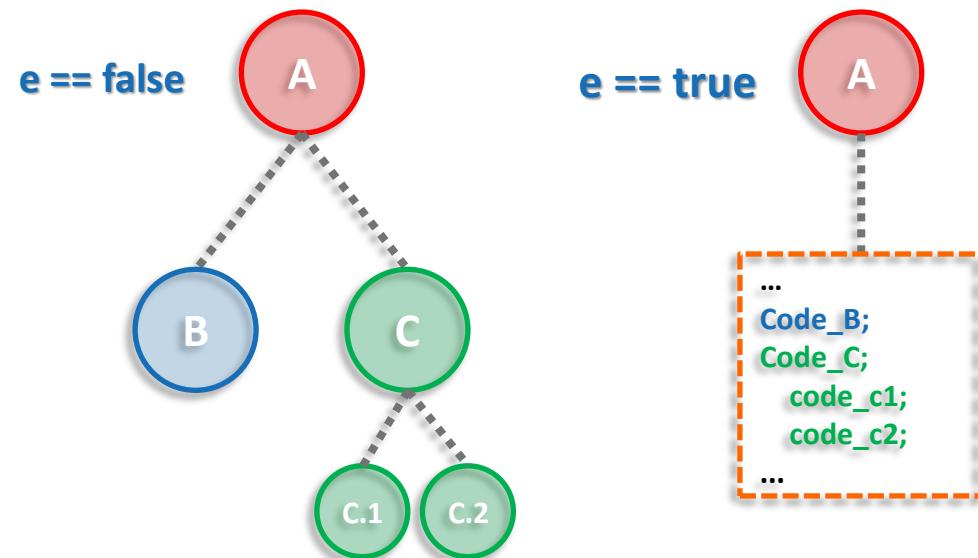
## ■ The final (expression) clause

- 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!

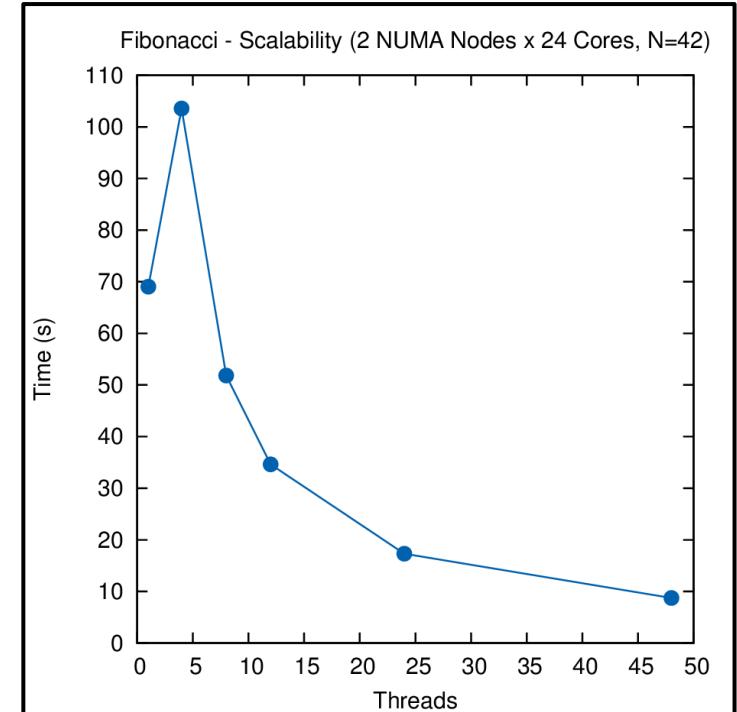
# The mergeable clause

- 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

# *Example: Fibonacci*

# Fibonacci: without cutoff

```
int fib(int n) {  
    if (n < 2)  
        return n;  
  
    int res1, res2;  
    #pragma omp task shared(res1)  
    res1 = fib(n-1);  
  
    #pragma omp task shared(res2)  
    res2 = fib(n-2);  
  
    #pragma omp taskwait  
  
    return res1 + res2;  
}
```

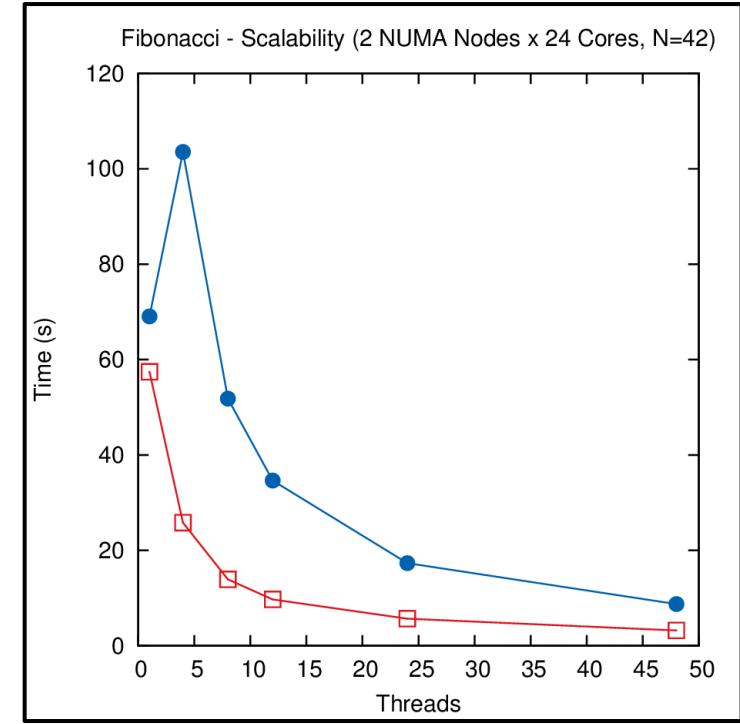


icc 2018.0

no\_cutoff —●—

# Fibonacci: if clause

```
int fib(int n) {  
    if (n < 2)  
        return n;  
  
    int res1, res2;  
#pragma omp task shared(res1) if(n > 30)  
res1 = fib(n-1);  
  
#pragma omp task shared(res2) if(n > 30)  
res2 = fib(n-2);  
  
#pragma omp taskwait  
  
    return res1 + res2;  
}
```

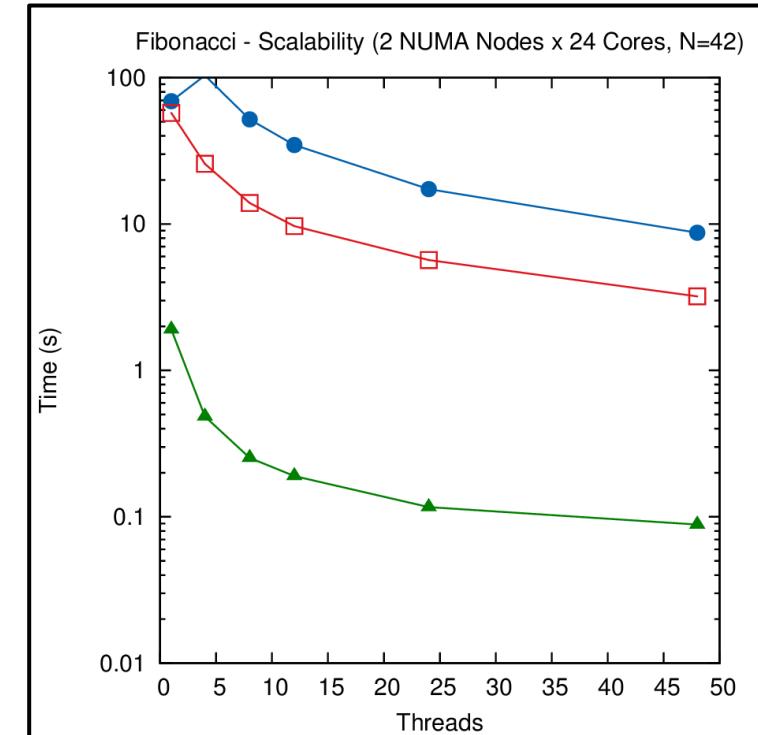


icc 2018.0

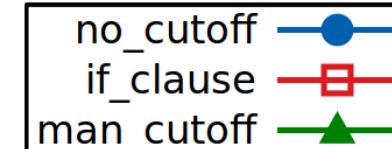
no\_cutoff —●—  
if\_clause —□—

# Fibonacci: manual optimization

```
int fib(int n) {  
    if (n < 30)  
        return fib_serial(n);  
  
    int res1, res2;  
    #pragma omp task shared(res1)  
    res1 = fib(n-1);  
  
    #pragma omp task shared(res2)  
    res2 = fib(n-2);  
  
    #pragma omp taskwait  
  
    return res1 + res2;  
}
```



icc 2018.0



# Task dependences

# Motivation

## ■ Task dependences as a way to define task-execution constraints

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

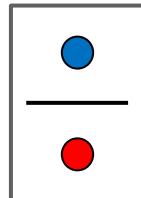
OpenMP 3.1

```
int x = 0;  
#pragma omp parallel  
#pragma omp single  
{  
    #pragma omp task depend(in: x)  
    std::cout << x << std::endl;  
  
    x++;  
}  
  
end(inout: x)
```

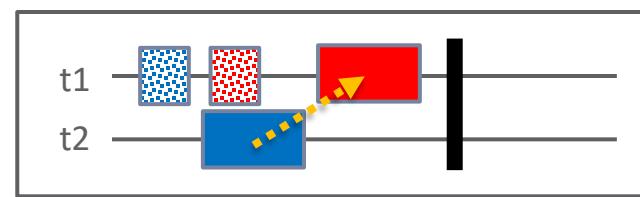
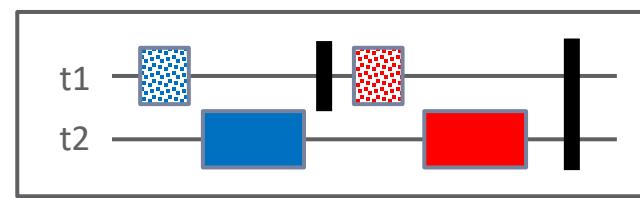
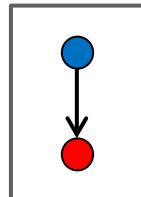
OpenMP 4.0

Task dependences can help us to remove  
“strong” synchronizations, increasing the look  
ahead and, frequently, the parallelism!!!!

OpenMP 3.1



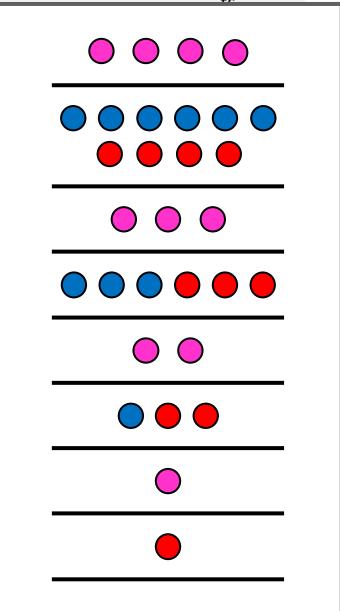
OpenMP 4.0



Task’s creation time  
Task’s execution time

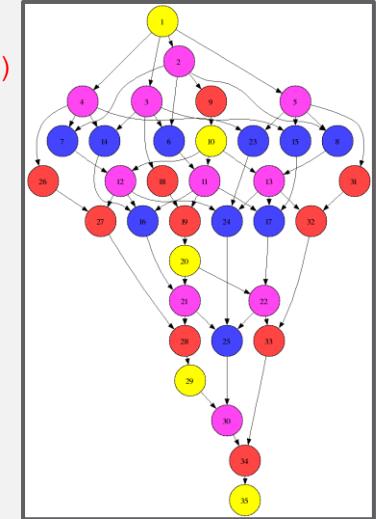
# Motivation: Cholesky factorization

```
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, ts);  
            }  
            #pragma omp task  
            syrk(a[k][i], a[i][i], ts, ts);  
        }  
        #pragma omp taskwait  
    }  
}
```



OpenMP 3.1

```
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])  
            #pragma omp task 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])  
                #pragma omp task depend(in: a[k][i], a[k][j])  
                dgemm(a[k][i], a[k][j], a[j][i], ts, ts, ts);  
            }  
            #pragma omp task depend(inout: a[i][i])  
            #pragma omp task depend(in: a[k][i])  
            syrk(a[k][i], a[i][i], ts, ts);  
        }  
    }  
}
```



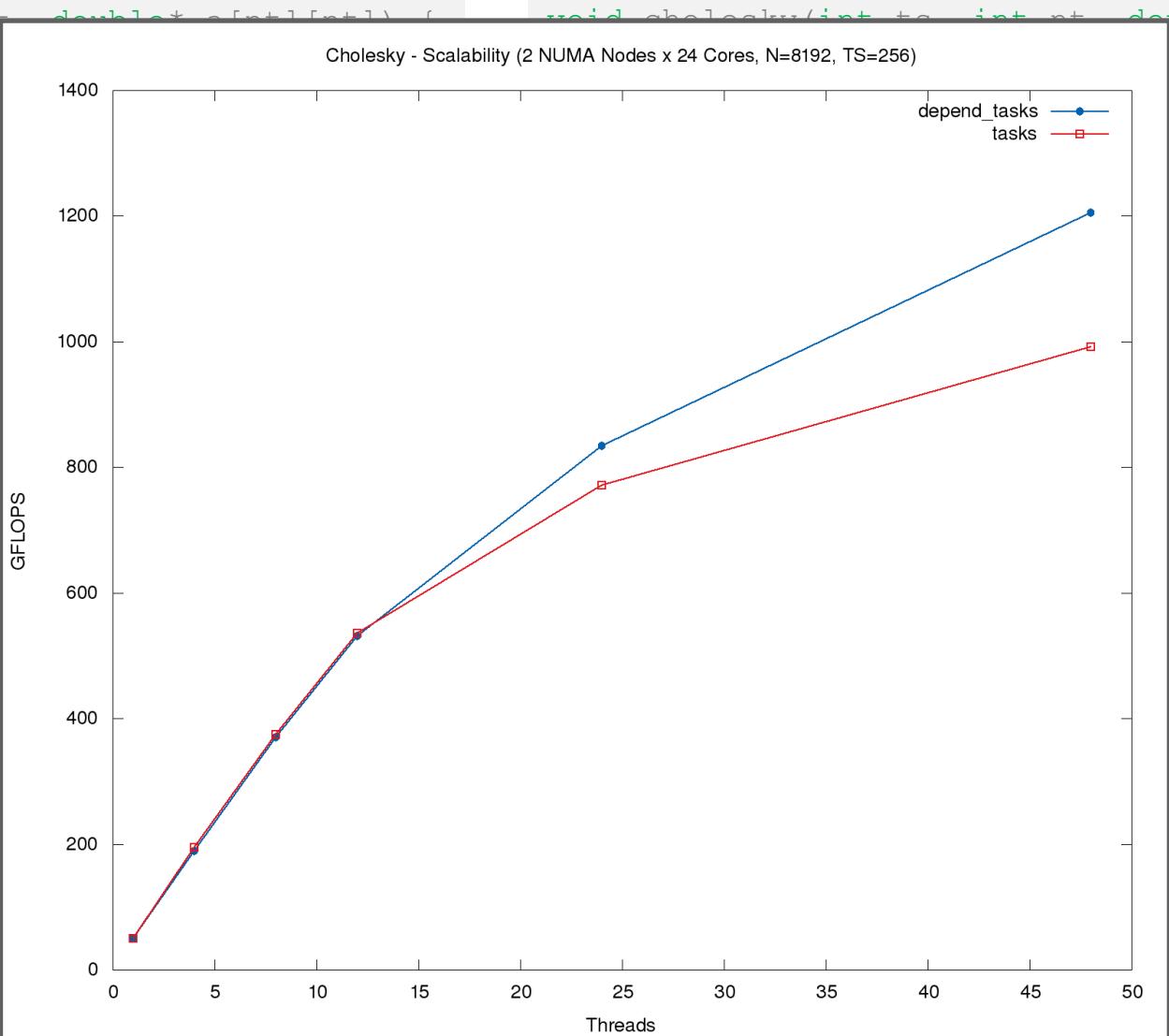
OpenMP 4.0

# Motivation: 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]);
        }
        #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], 1);
            }
            #pragma omp task
            syrk(a[k][i], a[i][i], ts, ts);
        }
        #pragma omp taskwait
    }
}
```



Using 2017 Intel compiler

```
void cholesky(int ts, int nt, double* a[nt][nt]) {
    for (int k = 0; k < nt; k++) {
        #pragma omp task
        a[k][k] = ts;
        for (int i = k + 1; i < nt; i++) {
            #pragma omp task
            a[k][i] = 0;
            for (int j = k + 1; j < i; j++) {
                #pragma omp task
                a[j][i] = 0;
            }
            #pragma omp task
            a[i][i] = ts;
        }
        #pragma omp taskwait
    }
}
```

The dependency graph illustrates the data dependencies in the Cholesky factorization algorithm. Nodes represent matrix elements, and edges represent assignments. The root node (0) is assigned a value of ts. This value is then used in the assignment of element (k, i) for all i > k. Element (k, i) is also used in the assignment of element (j, i) for all j < i < k. Finally, element (i, i) is assigned a value of ts.

OpenMP 4.0

OpenMP®

# What's in the spec: a bit of history

## OpenMP 4.0

- The `depend` clause was added to the task construct

## OpenMP 4.5

- The `depend` clause was added to the target constructs
- Support to doacross loops

## OpenMP 5.0

- lvalue expressions in the depend clause
- New dependency type: `mutexinoutset`
- Iterators were added to the depend clause
- The `depend` clause was added to the taskwait
- Dependable objects

## OpenMP 5.1

- New dependency type: `inoutset`

# What's in the spec: syntax depend clause

```
depend( [depend-modifier,] dependency-type: list-items)
```

where:

- depend-modifier is used to define iterators
- dependency-type may be: in, out, inout, inoutset, mutexinoutset and depobj
- A list-item may be:
  - C/C++: A lvalue expr or an array section      `depend(in: x, v[i], *p, w[10:10])`
  - Fortran: A variable or an array section      `depend(in: x, v(i), w(10:20))`

# What's in the spec: sema depend clause (1)

- A task cannot be executed until all its predecessor tasks are completed
- If a task defines an `in` dependence over a list-item
  - the task will depend on all previously generated sibling tasks that reference that list-item in an `out` or `inout` dependence
- If a task defines an `out/inout` dependence over list-item
  - the task will depend on all previously generated sibling tasks that reference that list-item in an `in`, `out` or `inout` dependence

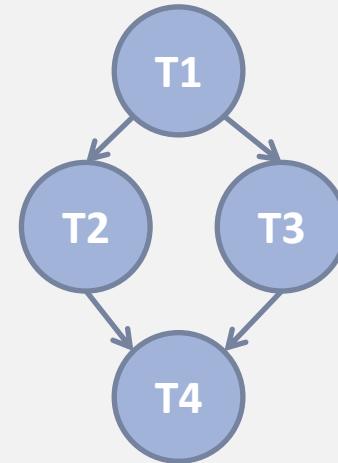
# What's in the spec: sema depend clause (1)

- A task cannot be executed until all its predecessor tasks are completed

- If a task defines

→ the task will complete  
inout dependency

```
int x = 0;  
#pragma omp parallel  
#pragma omp single  
{  
    #pragma omp task depend(inout: x) //T1  
    { ... }  
  
    #pragma omp task depend(in: x)      //T2  
    { ... }  
  
    #pragma omp task depend(in: x)      //T3  
    { ... }  
  
    #pragma omp task depend(inout: x) //T4  
    { ... }  
}
```



- If a task defines

→ the task will complete  
inout dependency

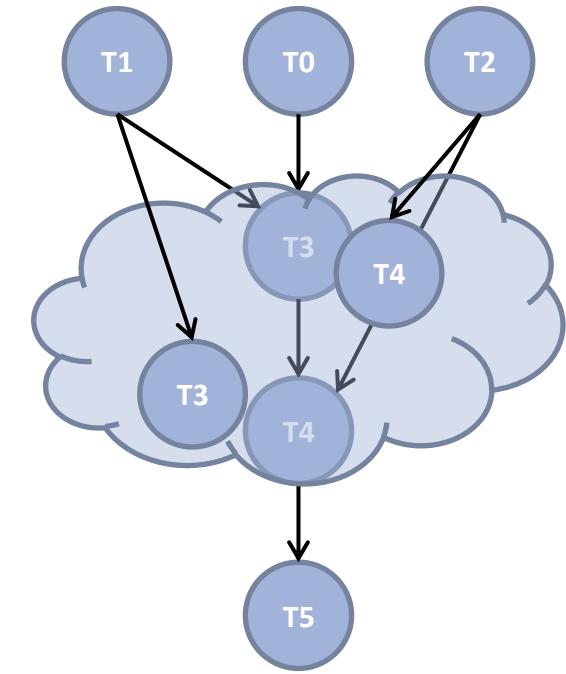
:em in an out or

:em in an in, out or

# What's in the spec: sema depend clause (2)

## ■ Set types: inoutset & mutexinoutset

```
int x = 0, y = 0, res = 0;  
#pragma omp parallel  
#pragma omp single  
{  
    #pragma omp task depend(out: res) //T0  
    res = 0;  
  
    #pragma omp task depend(out: x) //T1  
    long_computation(x);  
  
    #pragma omp task depend(out: y) //T2  
    short_computation(y);  
  
    #pragma omp task depend(in: x) depend(mutexinoutset:T3res) //T3  
    res += x;  
  
    #pragma omp task depend(in: y) depend(mutexinoutset:T4res) //T4  
    res += y;  
  
    #pragma omp task depend(in: res) //T5  
    std::cout << res << std::endl;  
}
```



1. *inoutset property*: tasks with a `mutexinoutset` dependence create a cloud of tasks (an inout set) that synchronizes with previous & posterior tasks that dependent on the same list item
2. *mutex property*: Tasks inside the inout set can be executed in any order but with mutual exclusion

# What's in the spec: sema depend clause (3)

- Task dependences are defined among **sibling tasks**

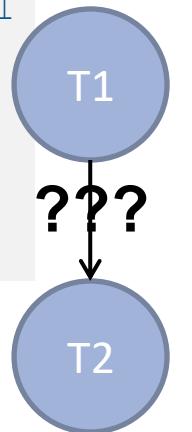
```
//test1.cc
int x = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(inout: x)    //T1
    {
        #pragma omp task depend(inout: x) //T1.1
        x++;

        #pragma omp taskwait
    }
    #pragma omp task depend(in: x) //T2
    std::cout << x << std::endl;
}
```

- List items used in the depend clauses [...] must indicate **identical** or **disjoint** storage

```
//test2.cc
int a[100] = {0};
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(inout: a[50:99]) //T1
    compute(/* from */ &a[50], /*elems*/ 50);

    #pragma omp task depend(in: a)    //T2
    print(/* from */ a, /* elem */ 100);
}
```



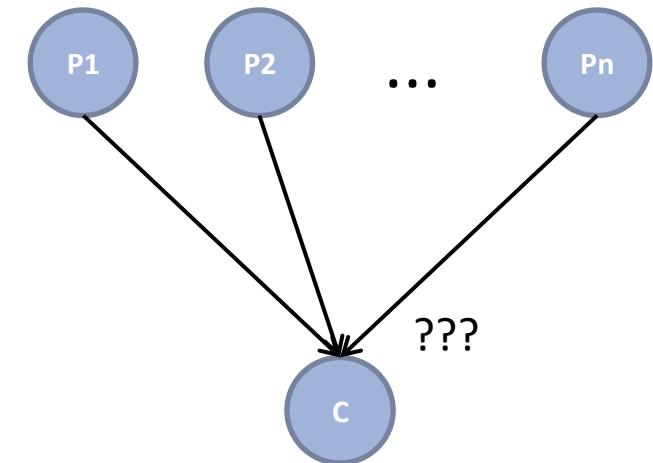
# What's in the spec: sema depend clause (4)

- Iterators + deps: a way to define a dynamic number of dependences

```
std::list<int> list = ....;
int n = list.size();

#pragma omp parallel
#pragma omp single
{
    for (int i = 0; i < n; ++i)
        #pragma omp task depend(out: list[i])      //Px
        compute_elem(list[i]);

        #pragma omp task depend(iterator(j=0:n), in : list[j]) //C
        print_elems(list);
}
```



Equivalent to:  
depend(in: list[0], list[1], ..., list[n-1])

# What's in the spec: deps on taskwait

- Adding dependences to the taskwait construct
  - Using a taskwait construct to explicitly wait for some predecessor tasks
  - Syntactic sugar!

```
int x = 0, y = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(inout: x) //T1
    x++;

    #pragma omp task depend(in: y)      //T2
    std::cout << y << std::endl;

    #pragma omp taskwait depend(in: x)
    std::cout << x << std::endl;
}
```

# What's in the spec: dependable objects (1)

- Offer a way to manually handle dependences

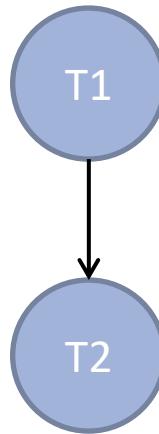
- Useful for complex task dependences
  - It allows a more efficient allocation of task dependences
  - New `omp_depend_t` opaque type
  - 3 new constructs to manage dependable objects
    - `#pragma omp depobj(obj) depend(dep-type: list)`
    - `#pragma omp depobj(obj) update(dep-type)`
    - `#pragma omp depobj(obj) destroy`

# What's in the spec: dependable objects (2)

- Offer a way to manually handle dependences

```
int x = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(inout: x) //T1
    x++;

    #pragma omp task depend(in: x)      //T2
    std::cout << x << std::endl;
}
```



```
int x = 0;
#pragma omp parallel
#pragma omp single
{
    omp_depend_t obj;
    #pragma omp depobj(obj) depend(inout: x)

    #pragma omp task depend(depobj: obj)      //T1
    x++;

    #pragma omp depobj(obj) update(in)
    #pragma omp task depend(depobj: obj)      //T2
    std::cout << x << std::endl;

    #pragma omp depobj(obj) destroy
}
```

# Philosophy: data-flow model

- Task dependences are orthogonal to data-sharings
  - Dependences as a way to define a **task-execution constraints**
  - Data-sharings as **how the data is captured** to be used inside the task

```
// test1.cc
int x = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(inout: x) \
                    firstprivate(x) //T1
    x++;

    #pragma omp task depend(in: x) //T2
    std::cout << x << std::endl;
}
```

```
// test2.cc
int x = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(inout: x) //T1
    x++;

    #pragma omp task depend(in: x) \
                    firstprivate(x) //T2
    std::cout << x << std::endl;
}
```

OK, but it always prints '0' :(

We have a data-race!!

# Philosophy: data-flow model (2)

- Properly combining dependences and data-sharings allow us to define a **task data-flow model**
  - Data that is read in the task → input dependence
  - Data that is written in the task → output dependence
- A task data-flow model
  - Enhances the **composability**
  - **Eases the parallelization** of new regions of your code

# Philosophy: data-flow model (3)

```
//test1_v1.cc
int x = 0, y = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(inout: x) //T1
    {
        x++;
        y++; // !!!
    }
    #pragma omp task depend(in: x) //T2
    std::cout << x << std::endl;

    #pragma omp taskwait
    std::cout << y << std::endl;
}
```

```
//test1_v2.cc
i //test1_v3.cc
#i //test1_v4.cc
{ int x = 0, y = 0;
# pragma omp parallel
# pragma omp single
{
    #pragma omp task depend(inout: x, y) //T1
    {
        x++;
        y++;
    }
    #pragma omp task depend(in: x) //T2
    std::cout << x << std::endl;

    #pragma omp task depend(in: y) //T3
    std::cout << y << std::endl;
}
```

If all tasks are **properly annotated**,  
we only have to worry about the  
dependences & data-sharings of the new task!!!

# Summary

- Cutt-offs, may allow to control task granularity
  - Trade-off among overhead and amount of parallelism
- Dependences, may allow to avoid strong synchronization
  - Increase the look-ahead and the amount of parallelism
- More on performant tasks
  - Affinity clause, reduce runtime by improving data locality (hint)
  - Cancellation, avoid unneeded execution of tasks (when possible, e.g., pruning)

# Thanks!

# Slides and Exercises

■ <https://tinyurl.com/openmp-umt-tasking>

2021 Tasking UMT

Nombre

- OpenMP-UMT-Exercises.tar.gz
- OpenMP-UMT-Tasking-1.pdf
- OpenMP-UMT-Tasking-2.pdf

