

# Programming OpenMP

**Christian Terboven**

Michael Klemm



# Agenda (in total 7 Sessions)

- Session 1: OpenMP Introduction
- **Session 2: Tasking**
  - Review of Session 1 / homework assignments
  - Tasking Motivation
  - Task Model in OpenMP
  - Scoping
  - Taskloop
  - Dependencies
  - Cut-off strategies
  - Homework assignments ☺
- Session 3: Optimization for NUMA and SIMD
- Session 4: What Could Possibly Go Wrong Using OpenMP
- Session 5: Introduction to Offloading with OpenMP
- Session 6: Advanced OpenMP Offloading Topics
- Session 7: Selected / Remaining Topics

# Programming OpenMP

## *Review*

Christian Terboven

**Michael Klemm**



# *Questions?*

# *Solution of Homework Assignments*

# Example: Hello

```
21 #include <stdio.h>
22 #include <omp.h>
23
24
25 int main()
26 {
27
28 #pragma omp parallel
29 {
30     int thread_num = omp_get_thread_num();
31     int num_threads = omp_get_num_threads();
32
33     printf("Hello from thread %d of %d.\n", thread_num, num_threads);
34 }
35
36     return 0;
37 }
```

# Example: Pi

```
66 double CalcPi(int n)
67 {
68     const double fH    = 1.0 / (double) n;
69     double fSum = 0.0;
70     double fX;
71     int i;

73 #pragma omp parallel for private(fX) reduction(+:fSum)
74 for (i = 0; i < n; i += 1)
75 {
76     fX = fH * ((double)i + 0.5);
77     fSum += f(fX);
78 }
79 return fH * fSum;
80 }
```

# Example: Jacobi / 1

```
167  ****
168 * Initializes data
169 * Assumes exact solution is u(x,y) = (1-x^2)*(1-y^2)
170 *
171 ****
172 void initialize(
173     int n,
174     int m,
175     double alpha,
176     double *dx,
177     double *dy,
178     double *u,
179     double *f)
180 {
181     int i,j,xx,yy;
182
183     *dx = 2.0 / (n-1);
184     *dy = 2.0 / (m-1);
185
186     /* Initialize initial condition and RHS */
187 #pragma omp parallel for private(i, xx, yy) // or collapse(2) instead of private(i)
188     for (j=0; j<m; j++){
189         for (i=0; i<n; i++){
190             xx = -1.0 + *dx * (i-1);
191             yy = -1.0 + *dy * (j-1);
192             U(j,i) = 0.0;
193             F(j,i) = -alpha * (1.0 - xx*xx) * (1.0 - yy*yy)
194                         - 2.0 * (1.0 - xx*xx) - 2.0 * (1.0 - yy*yy);
195         }
196     }
197 }
```

# Example: Jacobi / 2

```
85 #pragma omp parallel
86 {
87
88     /* copy new solution into old */
89     #pragma omp for private(i) // or collapse(2)
90     for (j=0; j<m; j++){
91         for (i=0; i<n; i++){
92             UOLD(j,i) = U(j,i);
93         }
94
95         /* compute stencil, residual and update */
96         #pragma omp for private(i, resid) reduction(+:error) // or collapse(2) instead of private(i)
97         for (j=1; j<m-1; j++){
98             for (i=1; i<n-1; i++){
99                 resid =(
100                     ax * (UOLD(j,i-1) + UOLD(j,i+1))
101                     + ay * (UOLD(j-1,i) + UOLD(j+1,i))
102                     + b * UOLD(j,i) - F(j,i)
103                 ) / b;
104
105                 /* update solution */
106                 U(j,i) = UOLD(j,i) - omega * resid;
107
108                 /* accumulate residual error */
109                 error =error + resid*resid;
110
111             }
112         }
```

# Example: for

```
28 double do_some_computation(int i)
29 {
30     double t = 0.0;
31     int j;
32     for (j = 0; j < i*i; j++)
33     {
34         t += sin((double)j) * cos((double)j);
35     }
36     return t;
37 }
38
39
40 int main(int argc, char* argv[])
41 {
42     const int dimension = 500;
43     int i;
44     double result = 0.0;
45
46     double t1 = omp_get_wtime();
47
48     #pragma omp parallel for schedule(dynamic) reduction(+:result)
49     for (i = 0; i < dimension; i++)
50     {
51         result += do_some_computation(i);
52     }
53
54     double t2 = omp_get_wtime();
```

# Example: minmaxreduction

```
42 #pragma omp parallel
43 {
44
45     #pragma omp for reduction(min: dMin) reduction(max: dMax)
46     for (i = 0; i < dimension; i++)
47     {
48         dArray[i] = func(i);
49         dMin = fmin(dMin, dArray[i]);
50         dMax = fmax(dMax, dArray[i]);
51     }
52
53 } // end omp parallel
```

# Programming OpenMP

## *Tasking Introduction*

**Christian Terboven**

Michael Klemm

RWTHAACHEN  
UNIVERSITYOpenMP®

# *Tasking Motivation*

# Sudoku 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) 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

- 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				

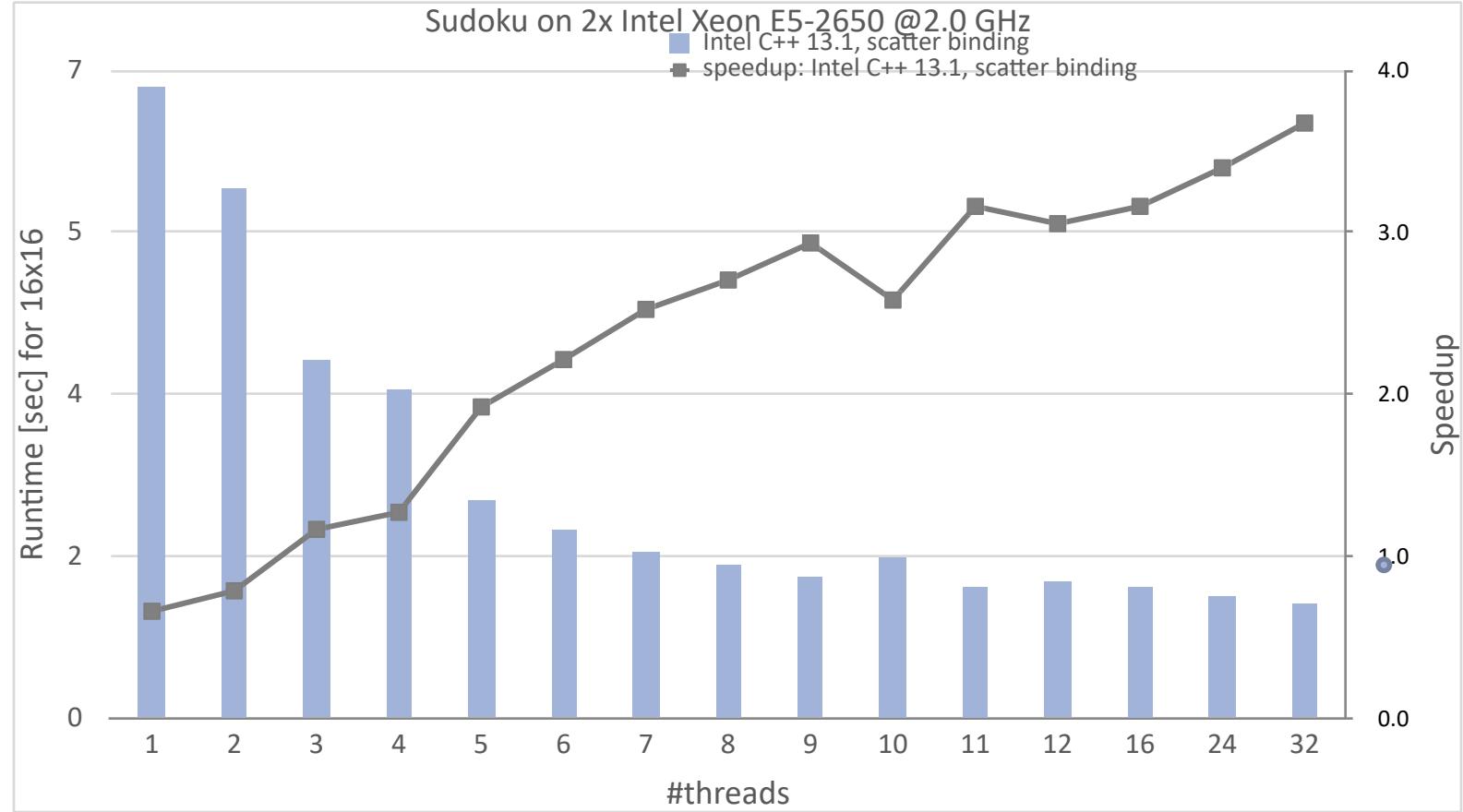
- (1) Search an empty field
- (2) Try all numbers:
  - (2 a) Check Sudoku
  - If invalid: skip
  - If valid: Go to next field
- Wait for completion

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



Is this the best  
we can do?

# Tasking Overview

# What is a task in OpenMP?

- Tasks are work units whose execution
  - may be deferred or...
  - ... can be executed immediately
- Tasks are composed of
  - **code** to execute, a **data** environment (initialized at creation time), internal **control** variables (ICVs)
- Tasks are created...
  - ... when reaching a parallel region → implicit tasks are created (per thread)
  - ... when encountering a task construct → explicit task is created
  - ... when encountering a taskloop construct → explicit tasks per chunk are created
  - ... when encountering a target construct → target task is created

# Tasking execution model

- Supports unstructured parallelism  
→ unbounded loops

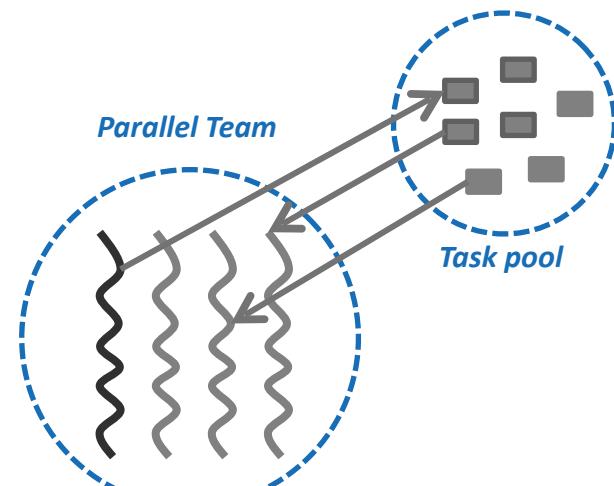
```
while ( <expr> ) {  
    ...  
}  
// RECURSIVE FUNCTIONS
```

```
void myfunc( <args> )  
{  
    ...; myfunc( <newargs> ); ...;  
}
```

- Several scenarios are possible:  
→ single creator, multiple creators, nested tasks (tasks & WS)
- All threads in the team are candidates to execute tasks

- Example (unstructured parallelism)

```
#pragma omp parallel  
#pragma omp masked ! Also: single  
while (elem != NULL) {  
    #pragma omp task  
    compute(elem);  
    elem = elem->next;  
}
```



# 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)
- firstprivate(list)
- shared(list)
- default(shared | none)
- in\_reduction(r-id: list)
- allocate([allocator:] list)
- detach(event-handler)

## Data Environment

## Miscellaneous

- if(scalar-expression)
- mergeable
- final(scalar-expression)
- depend(dep-type: list)
- untied
- priority(priority-value)
- affinity(list)

## Cutoff Strategies

## Synchronization

## Task Scheduling

# Task scheduling: tied vs untied tasks

- Tasks are tied by default (when no untied clause present)
  - tied tasks are executed always by the same thread (not necessarily creator)
  - tied tasks may run into performance problems
- Programmers may specify tasks to be untied (relax scheduling)

```
#pragma omp task untied  
{structured-block}
```

- can potentially switch to any thread (of the team),
- bad mix with thread based features: thread-id, threadprivate, critical regions...
  - gives the runtime more flexibility to schedule tasks
  - but most of OpenMP implementations doesn't "honor" untied ☹

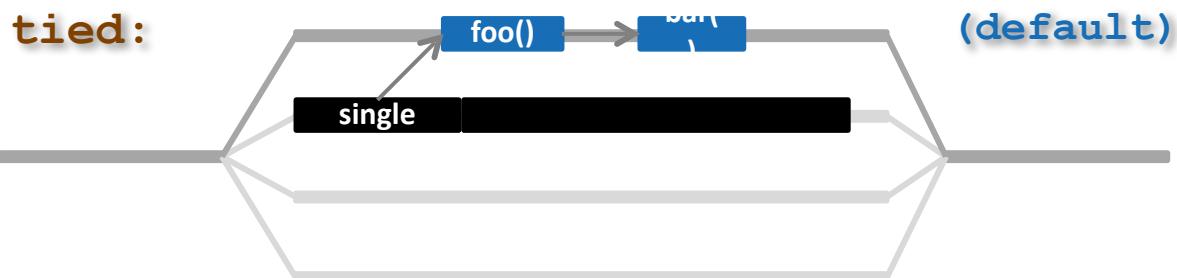
# Task scheduling: taskyield directive

- Task scheduling points (and the taskyield directive)
  - tasks can be suspended/resumed at TSPs → some additional constraints to avoid deadlock problems
  - implicit scheduling points (creation, synchronization, ... )
  - explicit scheduling point: the taskyield directive

```
#pragma omp taskyield
```

- Scheduling [tied/untied] tasks: example

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp task
    {
        foo();
        #pragma omp taskyield
        bar()
    }
}
```



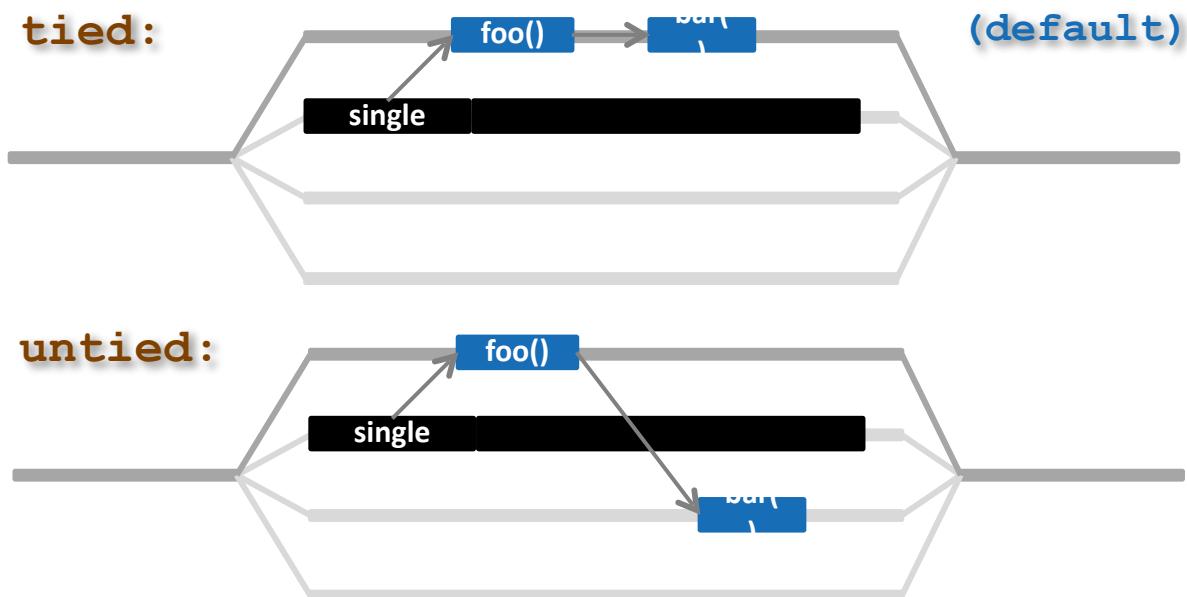
# Task scheduling: taskyield directive

- Task scheduling points (and the taskyield directive)
  - tasks can be suspended/resumed at TSPs → some additional constraints to avoid deadlock problems
  - implicit scheduling points (creation, synchronization, ... )
  - explicit scheduling point: the taskyield directive

```
#pragma omp taskyield
```

- Scheduling [tied/untied] tasks: example

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp task untied
    {
        foo();
        #pragma omp taskyield
        bar()
    }
}
```



# Task synchronization: taskwait directive

- The taskwait directive (shallow task synchronization)
  - It is a stand-alone directive

```
#pragma omp taskwait
```

→ wait on the completion of child tasks of the current task, just direct children, not all descendant tasks,  
includes an implicit task scheduling point (TSP)

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp task
    {
        #pragma omp task
        { ... }
        #pragma omp task
        { ...         ...}
        #pragma omp taskwait
    }
} // implicit barrier will wait for C.x
```

# Task synchronization: taskwait directive

- The taskwait directive (shallow task synchronization)
  - It is a stand-alone directive

```
#pragma omp taskwait
```

→ wait on the completion of child tasks of the current task, just direct children, not all descendant tasks,  
includes an implicit task scheduling point (TSP)

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp task
    {
        #pragma omp task
        { ... }
        #pragma omp task
        { ...         ...}
        #pragma omp taskwait
    }
    // implicit barrier will wait for C.x
```



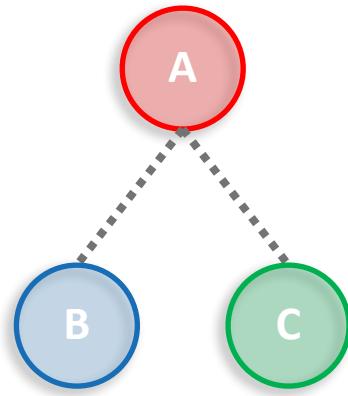
# Task synchronization: taskwait directive

- The taskwait directive (shallow task synchronization)
  - It is a stand-alone directive

```
#pragma omp taskwait
```

→ wait on the completion of child tasks of the current task, just direct children, not all descendant tasks,  
includes an implicit task scheduling point (TSP)

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp task
    {
        #pragma omp task : A
        {
            #pragma omp task : B
            {
                ...
            }
            #pragma omp task : C
            {
                ...
            }
            #pragma omp taskwait
        }
    }
    // implicit barrier will wait for C.x
}
```



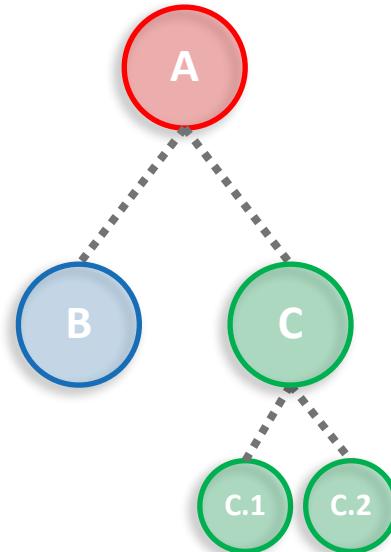
# Task synchronization: taskwait directive

- The taskwait directive (shallow task synchronization)
  - It is a stand-alone directive

```
#pragma omp taskwait
```

→ wait on the completion of child tasks of the current task, just direct children, not all descendant tasks,  
includes an implicit task scheduling point (TSP)

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp task
    {
        #pragma omp task : A
        {
            #pragma omp task : B
            {
                ...
            }
            #pragma omp task : C
            {
                ...
                #pragma omp taskwait
            }
        }
        // implicit barrier will wait for C.x
    }
}
```



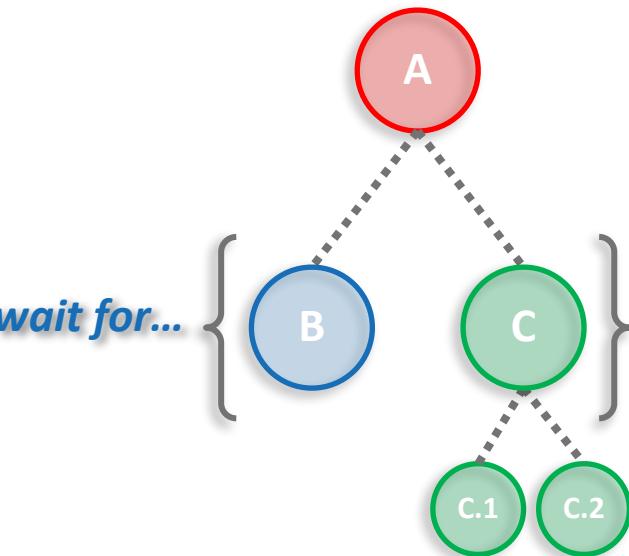
# Task synchronization: taskwait directive

- The taskwait directive (shallow task synchronization)
  - It is a stand-alone directive

```
#pragma omp taskwait
```

→ wait on the completion of child tasks of the current task, just direct children, not all descendant tasks,  
includes an implicit task scheduling point (TSP)

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp task
    {
        #pragma omp task : A
        {
            #pragma omp task : B
            {
                ...
            }
            #pragma omp task : C
            {
                ...
                #pragma omp task : C.1; #pragma omp task : C.2;
                ...
            }
        }
        #pragma omp taskwait
    }
    // implicit barrier will wait for C.x
}
```



# Task synchronization: barrier semantics

- OpenMP barrier (implicit or explicit)
  - All tasks created by any thread of the current team are guaranteed to be completed at barrier exit

```
#pragma omp barrier
```

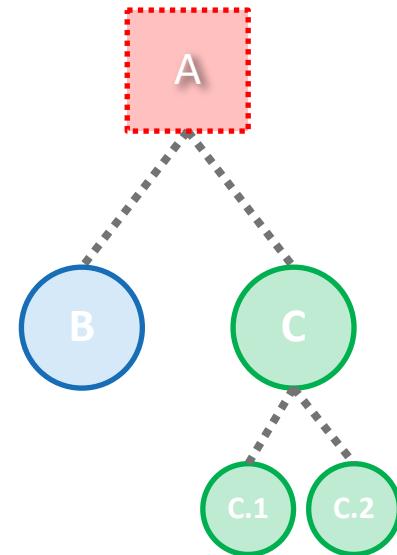
→ And all other implicit barriers at parallel, sections, for, single, etc...

# Task synchronization: taskgroup construct

- The taskgroup construct (deep task synchronization)
  - attached to a structured block; completion of all descendants of the current task; TSP at the end

```
#pragma omp taskgroup [clause[, clause]...]
{structured-block}
    → where clause (could only be). reduction(reduction-identifier, list-items)
```

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp taskgroup
    {
        #pragma omp task :A
        {
            #pragma omp task :B
            {
                ...
            }
            #pragma omp task :C
            {
                ...
                #C.1; #C.2; ...
            }
        } // end of taskgroup
    }
}
```



# Task synchronization: taskgroup construct

- The taskgroup construct (deep task synchronization)
  - attached to a structured block; completion of all descendants of the current task; TSP at the end

```
#pragma omp taskgroup [clause[, clause]...]
{structured-block}
    → where clause (could only be). reduction(reduction-identifier, list-items)
```

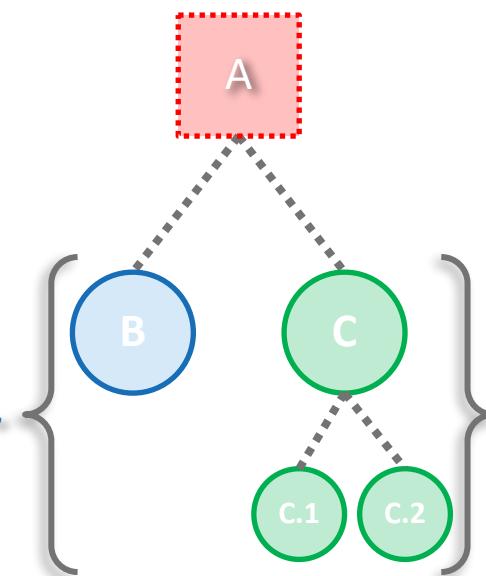
```
#pragma omp parallel
#pragma omp single
{
    #pragma omp taskgroup
    {
        #pragma omp task
        { ... }
        #pragma omp task
        { ... #C.1; #C.2; ... }
    } // end of taskgroup
}
```

:A

:B

:C

wait for...



# Data Environment

# Explicit data-sharing clauses

- Explicit data-sharing clauses (shared, private and firstprivate)

```
#pragma omp task shared(a)
{
    // Scope of a: shared
}
```

```
#pragma omp task private(b)
{
    // Scope of b: private
}
```

```
#pragma omp task
firstprivate(c)
{
    // Scope of c: firstprivate
}
```

# Explicit data-sharing clauses

- Explicit data-sharing clauses (shared, private and firstprivate)

```
#pragma omp task shared(a)
{
    // Scope of a: shared
}
```

```
#pragma omp task private(b)
{
    // Scope of b: private
}
```

```
#pragma omp task
firstprivate(c)
{
    // Scope of c: firstprivate
}
```

- If **default** clause present, what the clause says

- shared: data which is not explicitly included in any other data sharing clause will be **shared**
- none: compiler will issue an error if the attribute is not explicitly set by the programmer (very useful!!!!)

```
#pragma omp task default(shared)
{
    // Scope of all the references, not explicitly
    // included in any other data sharing clause,
    // and with no pre-determined attribute: shared
}
```

```
#pragma omp task default(None)
{
    // Compiler will force to specify the scope for
    // every single variable referenced in the
    // context
}
Hint: Use default(None) to be forced to think about every
variable if you do not see clearly.
```

# 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
  - 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
}
```

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

```
#pragma omp task
{
  static int y;
  // Scope of y: shared
}
```

```
void foo(void){
  static int s = MN;
}

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

# Implicit data-sharing attributes (in-practice)

- Implicit data-sharing rules for the task region
  - the **shared** attribute is lexically inherited
  - in any other case the variable is **firstprivate**
- Pre-determined rules (could not change)
- Explicit data-sharing clauses (+ default)
- Implicit data-sharing rules

# Implicit data-sharing attributes (in-practice)

- Implicit data-sharing rules for the task region
  - the **shared** attribute is lexically inherited
  - in any other case the variable is **firstprivate**

- Pre-determined rules (could not change)
- Explicit data-sharing clauses (+ default)
- Implicit data-sharing rules

- (in-practice) variable values within the task:
  - value of a: 1
  - value of b: x // undefined (undefined in parallel)
  - value of c: 3
  - value of d: 4
  - value of e: 5

```
int a = 1;
void foo() {
    int b = 2, c = 3;
    #pragma omp parallel private(b)
    {
        int d = 4;
        #pragma omp task
        {
            int e = 5;
            // Scope of a: -----
            // Scope of b: firstprivate
            // Scope of c: -----
            // Scope of d: firstprivate
            // Scope of e: -----
        }
    }
}
```

# Implicit data-sharing attributes (in-practice)

- Implicit data-sharing rules for the task region
  - the **shared** attribute is lexically inherited
  - in any other case the variable is **firstprivate**

- Pre-determined rules (could not change)
- Explicit data-sharing clauses (+ default)
- Implicit data-sharing rules

- (in-practice) variable values within the task:
  - value of a: 1
  - value of b: x // undefined (undefined in parallel)
  - value of c: 3
  - value of d: 4
  - value of e: 5

```
int a = 1;
void foo() {
    int b = 2, c = 3;
    #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: firstprivate
            // Scope of e: -----
        }
    }
}
```

# Implicit data-sharing attributes (in-practice)

- Implicit data-sharing rules for the task region
  - the **shared** attribute is lexically inherited
  - in any other case the variable is **firstprivate**

- Pre-determined rules (could not change)
- Explicit data-sharing clauses (+ default)
- Implicit data-sharing rules

- (in-practice) variable values within the task:
  - value of a: 1
  - value of b: x // undefined (undefined in parallel)
  - value of c: 3
  - value of d: 4
  - value of e: 5

```
int a = 1;
void foo() {
    int b = 2, c = 3;
    #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: ...
        }
    }
}
```

# Implicit data-sharing attributes (in-practice)

- Implicit data-sharing rules for the task region
  - the **shared** attribute is lexically inherited
  - in any other case the variable is **firstprivate**

- Pre-determined rules (could not change)
- Explicit data-sharing clauses (+ default)
- Implicit data-sharing rules

- (in-practice) variable values within the task:
  - value of a: 1
  - value of b: x // undefined (undefined in parallel)
  - value of c: 3
  - value of d: 4
  - value of e: 5

```
int a = 1;
void foo() {
    int b = 2, c = 3;
    #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: .....
        }
    }
}
```

# Implicit data-sharing attributes (in-practice)

- Implicit data-sharing rules for the task region
  - the **shared** attribute is lexically inherited
  - in any other case the variable is **firstprivate**

- Pre-determined rules (could not change)
- Explicit data-sharing clauses (+ default)
- Implicit data-sharing rules

- (in-practice) variable values within the task:
  - value of a: 1
  - value of b: x // undefined (undefined in parallel)
  - value of c: 3
  - value of d: 4
  - value of e: 5

```
int a = 1;
void foo() {
    int b = 2, c = 3;
    #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
        }
    }
}
```

# Implicit data-sharing attributes (in-practice)

- Implicit data-sharing rules for the task region
  - the **shared** attribute is lexically inherited
  - in any other case the variable is **firstprivate**

- Pre-determined rules (could not change)
- Explicit data-sharing clauses (+ default)
- Implicit data-sharing rules

- (in-practice) variable values within the task:
  - value of a: 1
  - value of b: x // undefined (undefined in parallel)
  - value of c: 3
  - value of d: 4
  - value of e: 5

```
int a = 1;
void foo() {
    int b = 2, c = 3;
    #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
        }
    }
}
```

# Task reductions (using taskgroup)

- Reduction operation
  - perform some forms of recurrence calculations
  - associative and commutative operators
- The (taskgroup) scoping reduction clause

```
#pragma omp taskgroup task_reduction(op: list)
{structured-block}
```

- The (task) in\_reduction clause [participating]

→ Task participates in a reduction operation [2]

```
#pragma omp task in_reduction(op: list)
{structured-block}
```

```
int res = 0;
node_t* node = NULL;

...
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp taskgroup task_reduction(+: res)
        { // [1]
            while (node) {
                #pragma omp task in_reduction(+: res) \
                    firstprivate(node)
                { // [2]
                    res += node->value;
                }
                node = node->next;
            }
        } // [3]
    }
}
```

# Task reductions (+ modifiers)

- Reduction modifiers
  - Former reductions clauses have been extended
  - task modifier allows to express task reductions
  - Registering a new task reduction [1]
  - Implicit tasks participate in the reduction [2]
  - Compute final result after [4]
- The (task) in\_reduction clause [participating]

```
#pragma omp task in_reduction(op: list)
{structured-block}
```

- Task participates in a reduction operation [3]

```
int res = 0;
node_t* node = NULL;
...
#pragma omp parallel reduction(task,+: res)
{ // [1][2]
#pragma omp single
{
#pragma omp taskgroup
{
while (node) {
#pragma omp task in_reduction(+: res) \
firstprivate(node)
{ // [3]
res += node->value;
}
node = node->next;
}
}
}
} // [4]
```

# Tasking illustrated

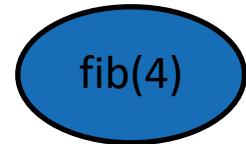
# Fibonacci illustrated

```
1 int main(int argc,
2          char* argv[])
3 {
4     [...]
5     #pragma omp parallel
6     {
7         #pragma omp single
8         {
9             fib(input);
10        }
11    }
12    [...]
13.}
```

```
14 int fib(int n)  {
15     if (n < 2) return n;
16     int x, y;
17     #pragma omp task shared(x)
18     {
19         x = fib(n - 1);
20     }
21     #pragma omp task shared(y)
22     {
23         y = fib(n - 2);
24     }
25     #pragma omp taskwait
26     return x+y;
27.}
```

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

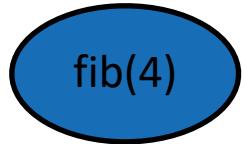
- T1 enters fib(4)



Task Queue



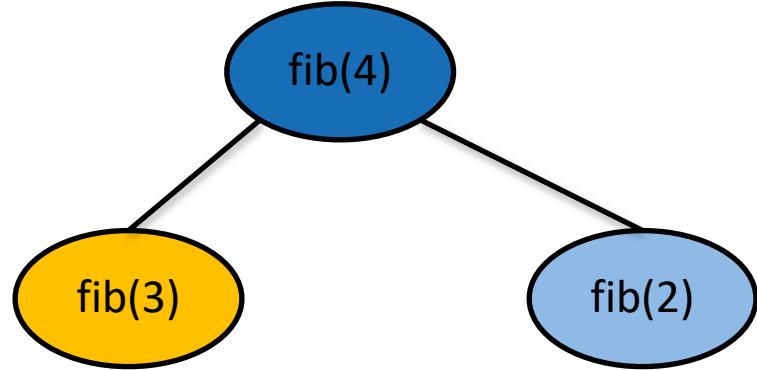
- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)



Task Queue



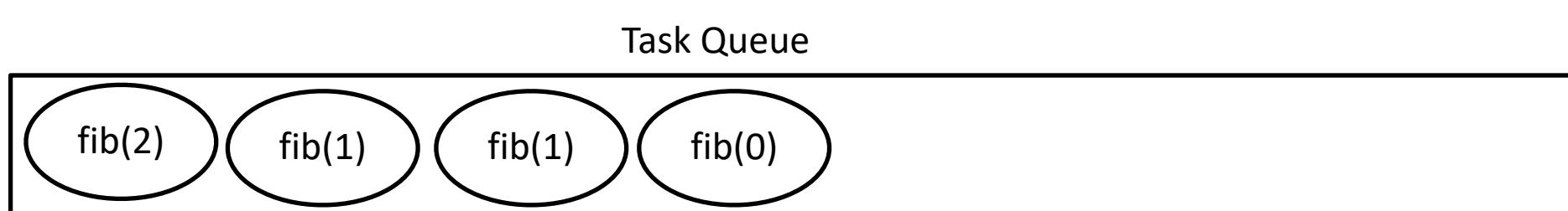
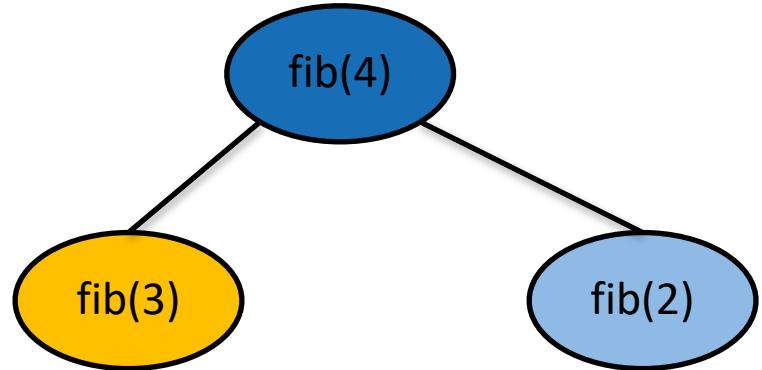
- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue



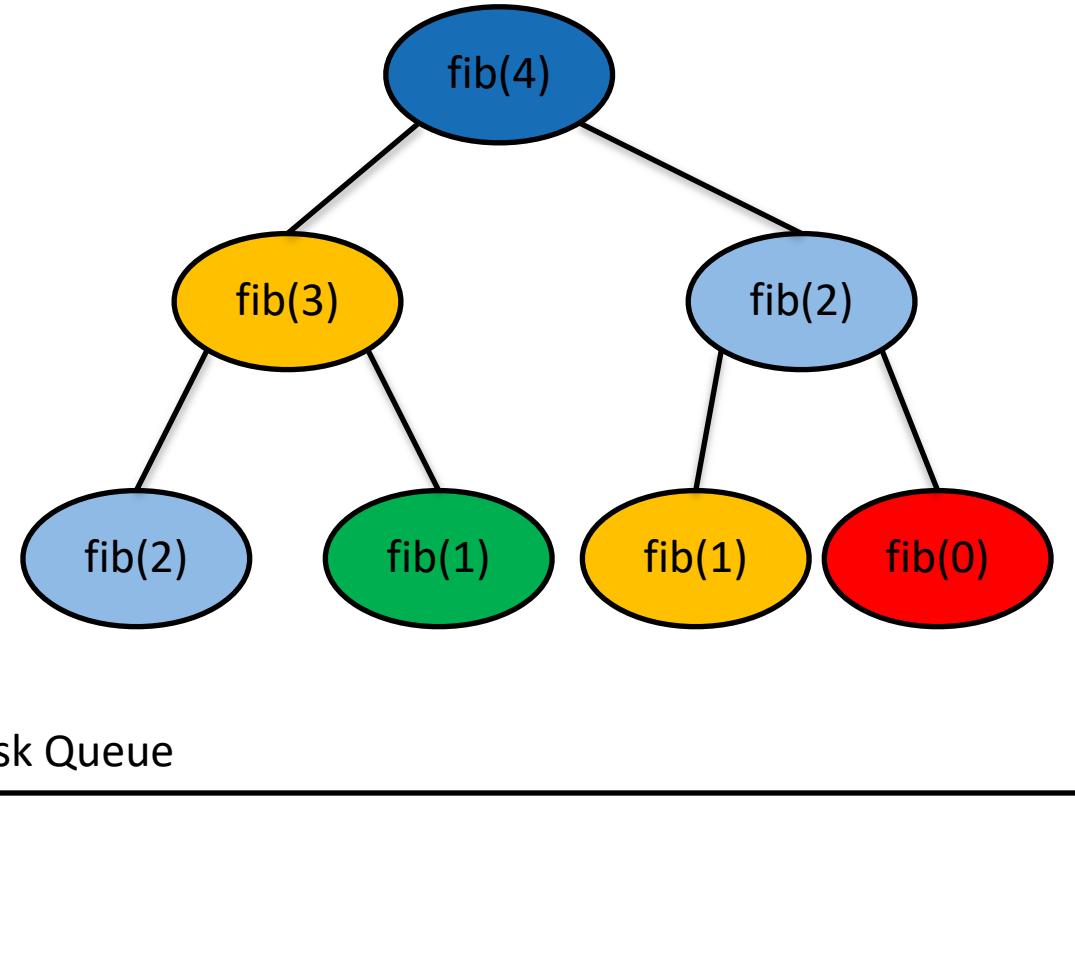
Task Queue



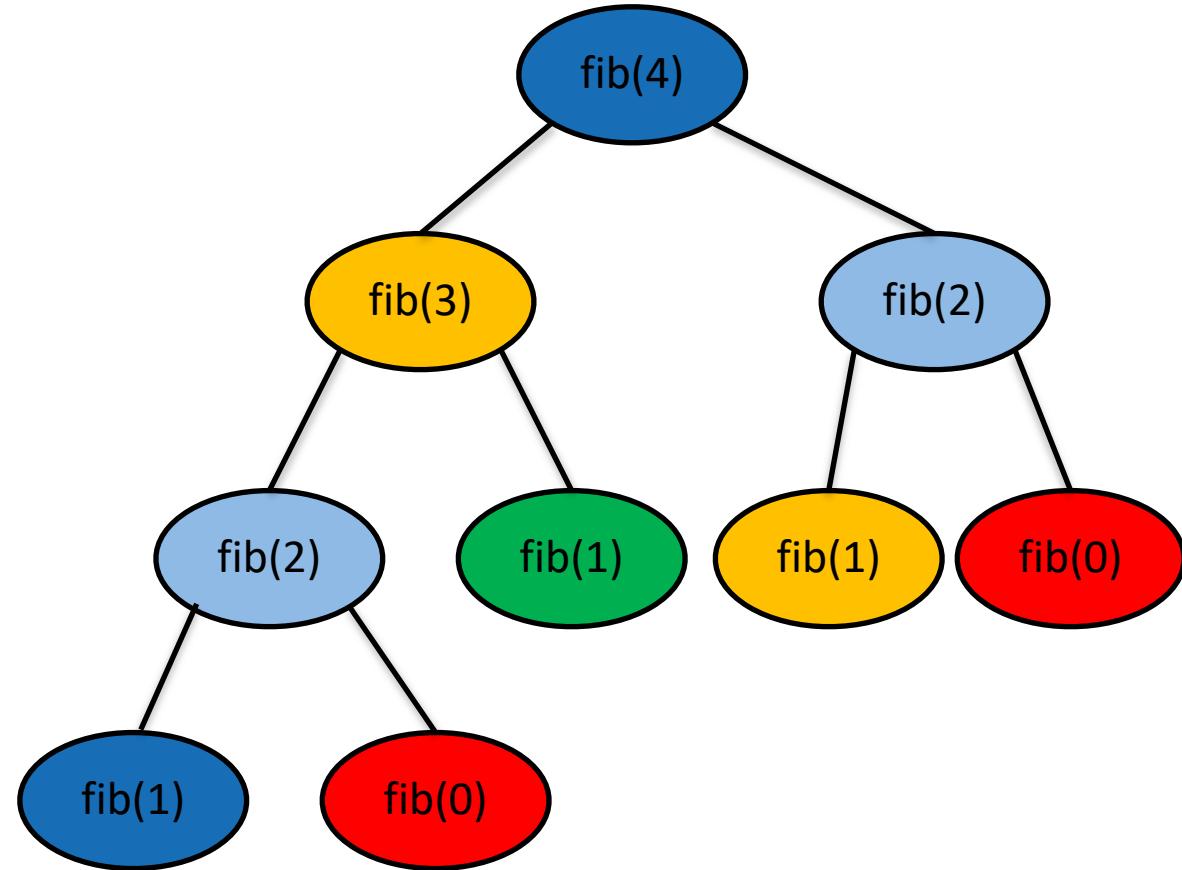
- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks



- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks



- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks
- ...



# The taskloop Construct

# Tasking use case: saxpy (taskloop)

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

```
for ( i = 0; i<SIZE; i+=TS) {
    UB = SIZE < (i+TS)?SIZE:i+TS;
    for ( ii=i; ii<UB; ii++) {
        A[ii]=A[ii]*B[ii]*S;
    }
}
```

```
#pragma omp parallel
#pragma omp single
for ( i = 0; i<SIZE; i+=TS) {
    UB = SIZE < (i+TS)?SIZE:i+TS;
    #pragma omp task private(ii) \
        firstprivate(i,UB) shared(S,A,B)
    for ( ii=i; ii<UB; ii++) {
        A[ii]=A[ii]*B[ii]*S;
    }
}
```

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

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

→ Grain size ~ Tile size (TS) → but implementation decides exact grain size

# 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)
- private(list)
- firstprivate(list)
- lastprivate(list)
- default(sh | *pr* | *fp* | none)
- reduction(r-id: list)
- in\_reduction(r-id: list)
- grainsize(grain-size)
- num\_tasks(num-tasks)

## Data Environment

## Chunks/Grain

- if(scalar-expression)
- final(scalar-expression)
- mergeable

## Cutoff Strategies

- untied
- priority(priority-value)
- collapse(n)

## Scheduler (R/H)

- 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
```

```
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
```

# 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
```

```
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
```

# 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 iterations
- Clause: num\_tasks(num-tasks)
  - Create num-tasks chunks
  - Each chunk must have at least one iteration

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

```
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

# Collapsing iteration spaces with taskloop

- The collapse clause in the taskloop construct

```
#pragma omp taskloop collapse(n)
{structured-for-loops}
```

- Number of loops associated with the taskloop construct (n)
- Loops are collapsed into one larger iteration space
- Then divided according to the **grainsize** and **num\_tasks**

- Intervening code between any two associated loops
  - at least once per iteration of the enclosing loop
  - at most once per iteration of the innermost loop

```
#pragma omp taskloop collapse(2)
for ( i = 0; i<SX; i+=1) {
    for ( j= 0; i<SY; j+=1) {
        for ( k = 0; i<SZ; k+=1) {
            A[f(i,j,k)]=<expression>;
        }
    }
}
```



```
#pragma omp taskloop
for ( ij = 0; i<SX*SY; ij+=1) {
    for ( k = 0; i<SZ; k+=1) {
        i = index_for_i(ij);
        j = index_for_j(ij);
        A[f(i,j,k)]=<expression>;
    }
}
```

# Task reductions (using taskloop)

- Clause: reduction(r-id: list)
  - It defines the scope of a new reduction
  - All created tasks participate in the reduction
  - It cannot be used with the `nogroup` clause

```
double dotprod(int n, double *x, double *y)
{
    double r = 0.0;
#pragma omp taskloop reduction(+: r)
    for (i = 0; i < n; i++)
        r += x[i] * y[i];

    return r;
}
```

# Task reductions (using taskloop)

- Clause: reduction(r-id: list)
  - It defines the scope of a new reduction
  - All created tasks participate in the reduction
  - It cannot be used with the **nogroup** clause
- Clause: in\_reduction(r-id: list)
  - Reuse an already defined reduction scope
  - All created tasks participate in the reduction
  - It can be used with the **nogroup\*** clause, but it is user responsibility to guarantee result

```
double dotprod(int n, double *x, double *y)
{
    double r = 0.0;
#pragma omp taskloop reduction(+: r)
    for (i = 0; i < n; i++)
        r += x[i] * y[i];

    return r;
}
```

```
double dotprod(int n, double *x, double *y)
{
    double r = 0.0;
#pragma omp taskgroup task_reduction(+: r)
{
    #pragma omp taskloop in_reduction(+: r)*
    for (i = 0; i < n; i++)
        r += x[i] * y[i];
}
return r;
}
```

# Composite construct: taskloop simd

- Task generating construct: decompose a loop into chunks, create a task for each loop chunk
- Each generated task will apply (internally) SIMD to each loop chunk  
→ C/C++ syntax:

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

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

- Where clause is any of the clauses accepted by **taskloop** or **simd** directives

# Improving Tasking Performance: Task dependences

# Motivation

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

# 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 task  
    x++;  
}
```

OpenMP 3.1

# 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

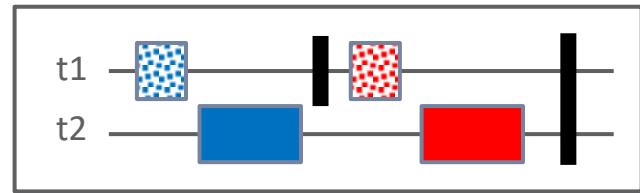
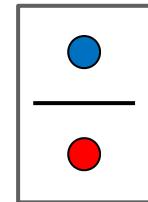
# Motivation

- 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

OpenMP 3.1



Task's creation time



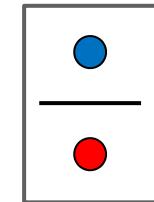
Task's execution time

# 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++;  
}
```

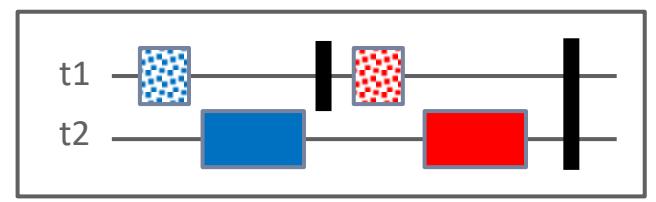
OpenMP 3.1



OpenMP 3.1

```
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++;  
}
```

OpenMP 4.0



Task's creation time  
 Task's execution time

# Motivation

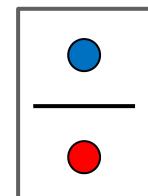
- Task dependences as a way to define task-execution constraints

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

    #pragma omp taskwait

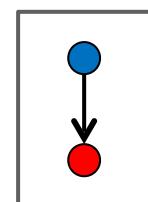
    #pragma omp task
    x++;
}
```

**OpenMP 3.1**



**OpenMP 3.1**

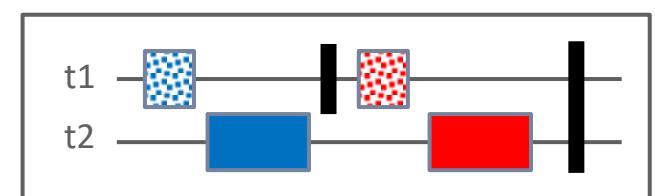
**OpenMP 4.0**



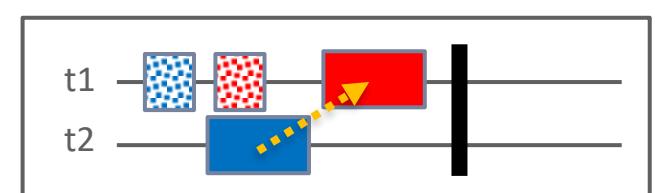
```
int x = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(in: x)
    std::cout << x << std::endl;
```

**OpenMP 4.0**

```
#pragma omp task depend(inout: x)
x++;
}
```



[■] Task's creation time  
[■] Task's execution time



# 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;

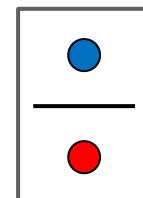
    x++;

    #pragma omp end(inout: x)
```

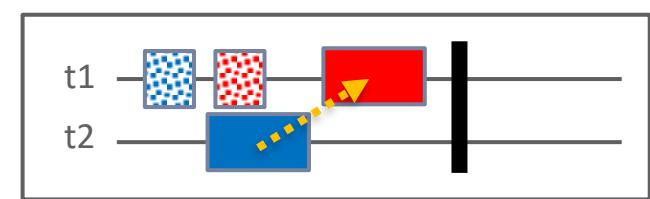
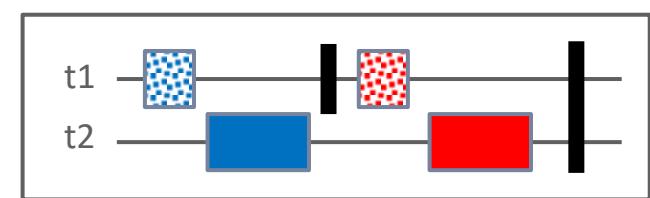
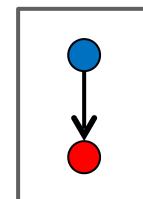
**OpenMP 4.0**

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

**OpenMP 3.1**



**OpenMP 4.0**



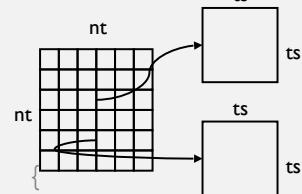
Task's creation time  
 Task's execution time

# 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

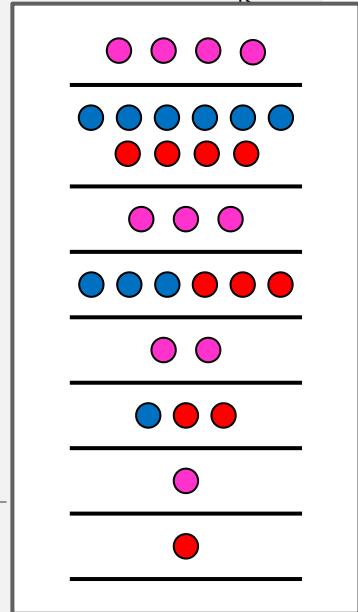
        // Update trailing matrix
        for (int i = k + 1; i < nt; i++) {
            for (int j = k + 1; j < i; j++) {
                #pragma omp task
                dgemm(a[k][i], a[k][j], a[j][i], ts, ts);
            }
            #pragma omp task
            syrk(a[k][i], a[i][i], ts, ts);
        }
        #pragma omp taskwait
    }
}
```



OpenMP 3.1

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



OpenMP 3.1

# 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);

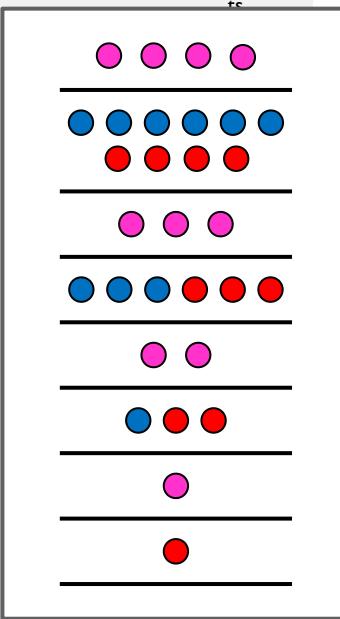
        // Triangular systems
        for (int i = k + 1; i < nt; i++) {
            #pragma omp task
            trsm(a[k][k], a[k][i], ts, ts);
        }

        #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]);
            }

            #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);
            }

            #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);
        }
    }
}
```

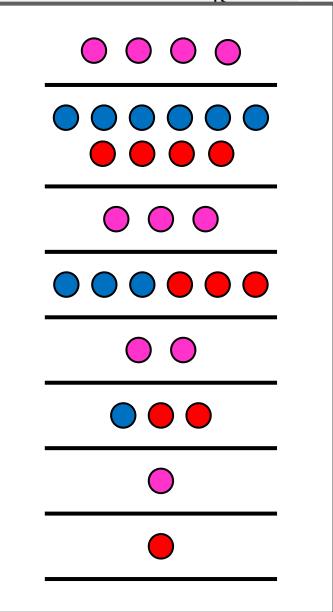
OpenMP 4.0

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



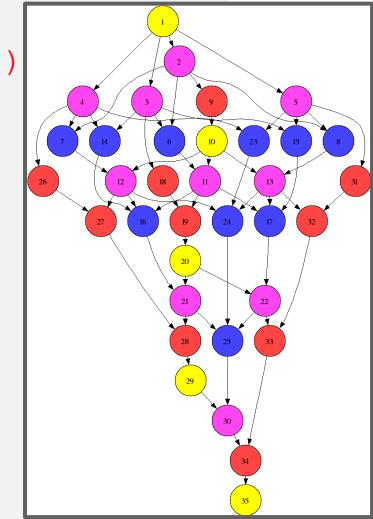
OpenMP 3.1

```
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);

                #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);
            }
        }
    }
}
```



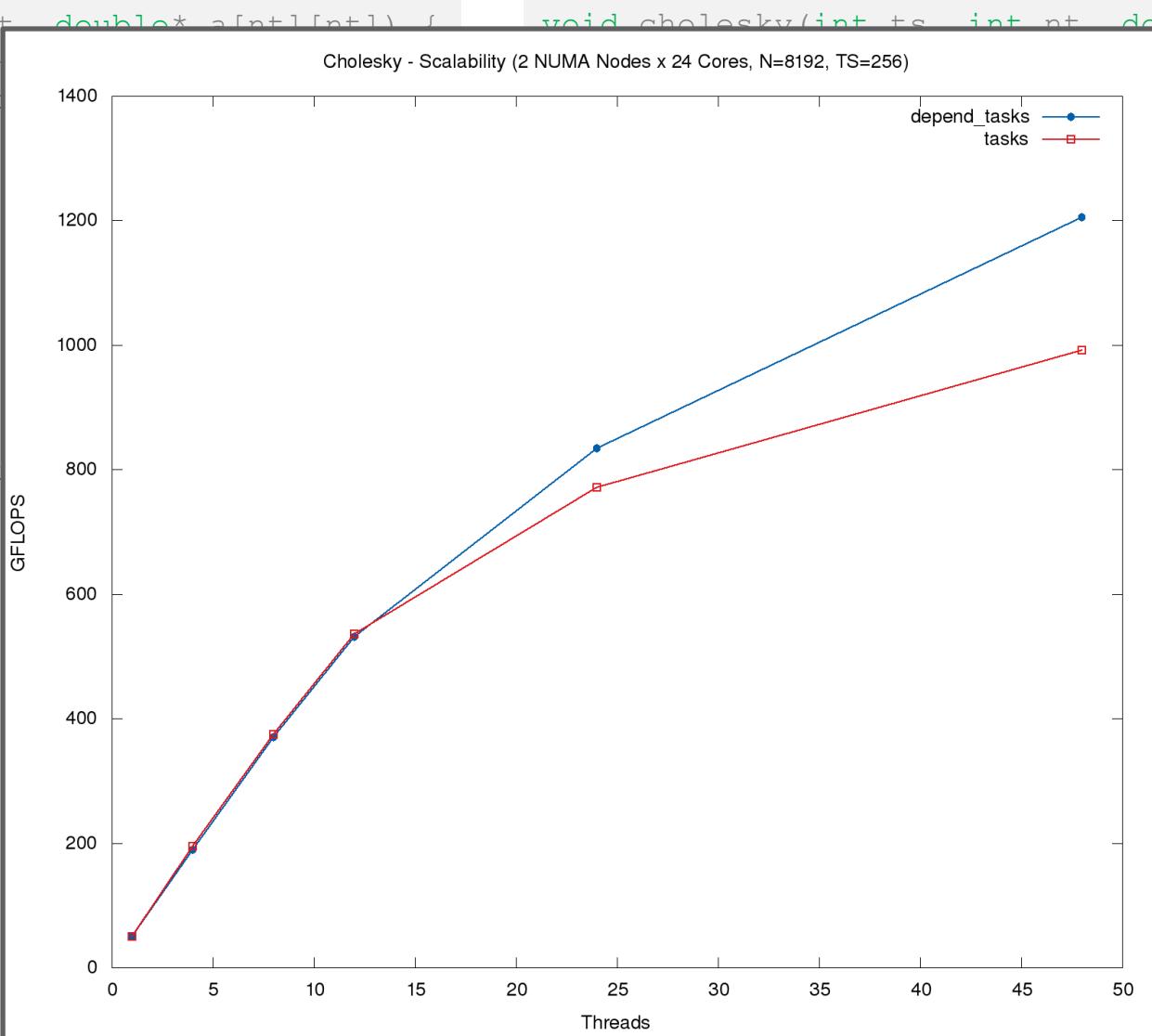
OpenMP 4.0

# 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], 1.0);
            }
            #pragma omp task
            syrk(a[k][i], a[i][i], ts);
        }
        #pragma omp taskwait
    }
}
```



```
ion
t: a[k][k])
i++ {
: a[k][k])
a[k][i])
ts);

i++ {
j++ {
inout: a[j][i])
[k][i], a[k][j])
a[j][i], ts, ts);

out: a[i][i])
k][i])
ts);
```

OpenMP 4.0

# *What's in the spec*

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

## OpenMP 4.0

- The depend clause was added to the task construct

# 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

# 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` construct
- Dependable objects

# 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, 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

# 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: depend clause (1)

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

- If a task defines

→ the task will complete

an out or in

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

    #pragma omp task depend(in: x)      //T2
    { ... }

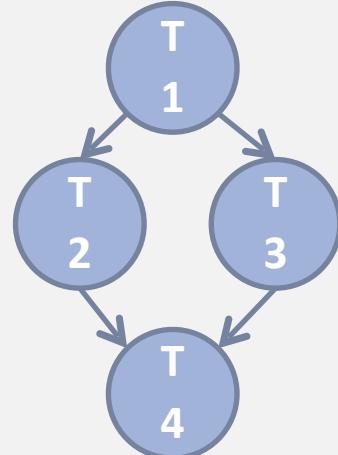
    #pragma omp task depend(in: x)      //T3
    { ... }

    #pragma omp task depend(inout: x) //T4
    { ... }
}
```

- If a task defines

→ the task will complete

an in, out or inout



one of the list items in

one of the list items in

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

- New dependency type: mutexinoutset

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

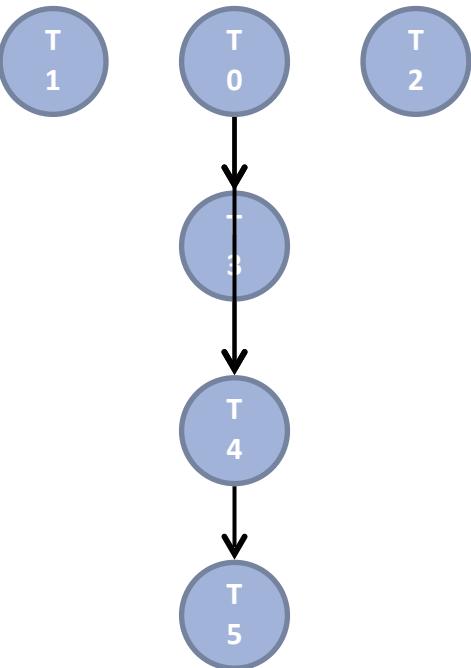
- New dependency type: mutexinoutset



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

- New dependency type: 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(inout: res) //T3  
    res += x;  
  
    #pragma omp task depend(in: y)      depend(inout: res) //T4  
    res += y;  
  
    #pragma omp task depend(in: res) //T5  
    std::cout << res << std::endl;  
}
```



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

- New dependency type: 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);

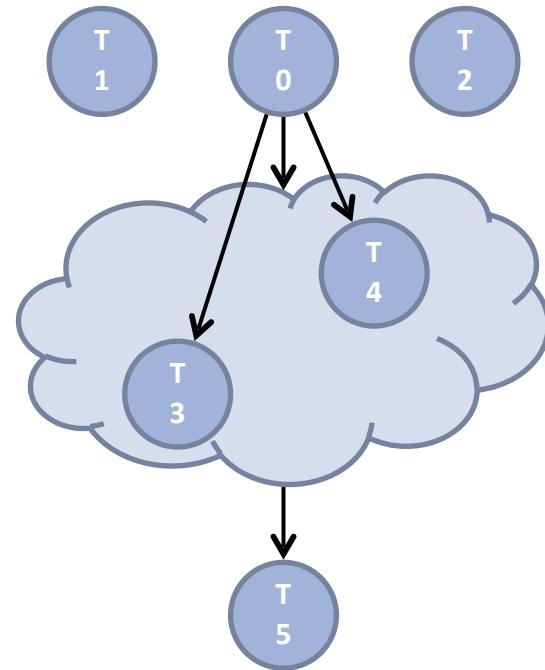
    #pragma omp task depend(out: y) //T2
    short_computation(y);

    #pragma omp task depend(in: x)      depend(mutexinoutset: res) //T3
    res += x;

    #pragma omp task depend(in: y)      depend(mutexinoutset: res) //T4
    res += y;

    #pragma omp task depend(in: res)   //T5
    std::cout << res << std::endl;
}

```



- 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
- mutex property:* Tasks inside the inout set can be executed in any order but with mutual exclusion

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

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

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

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

```
depend(inout: x)
```

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

- 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;
}
```

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

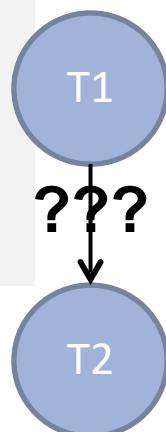
- Task dependences are defined among **sibling tasks**
- List items used in the depend clauses [...] must indicate **identical** or **disjoint storage**

```
//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;
}
```

```
//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: depend clause (4)

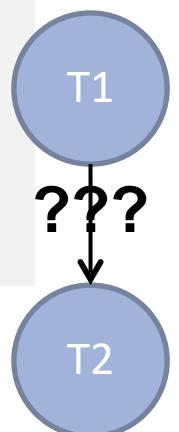
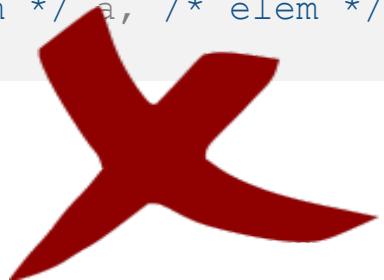
- Task dependences are defined among **sibling tasks**
- List items used in the depend clauses [...] must indicate **identical** or **disjoint storage**

```
//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;
}
```

```
//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);
}
```



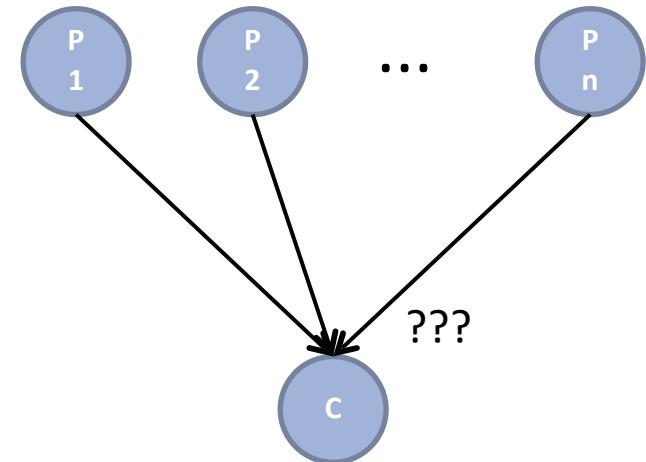
# What's in the spec: depend clause (5)

- 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
    print_elems(list), depend(in: ???)               //C
}
```



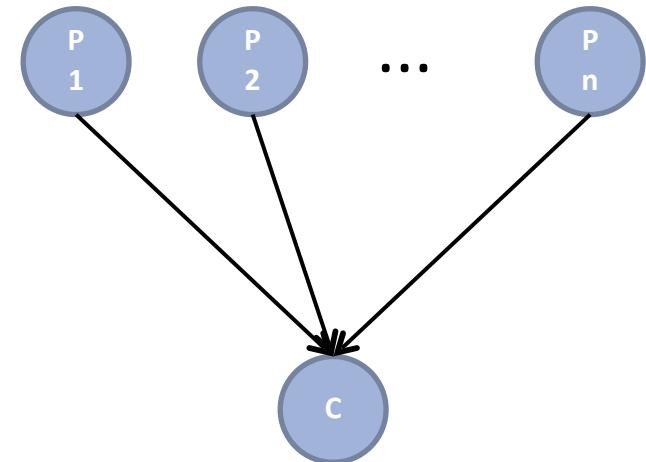
# What's in the spec: depend clause (5)

- 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
    print_elems(list), depend(iterator(j=0:n), in : list[j]) //C
}
```



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

# What's in the spec: depend clause (5)

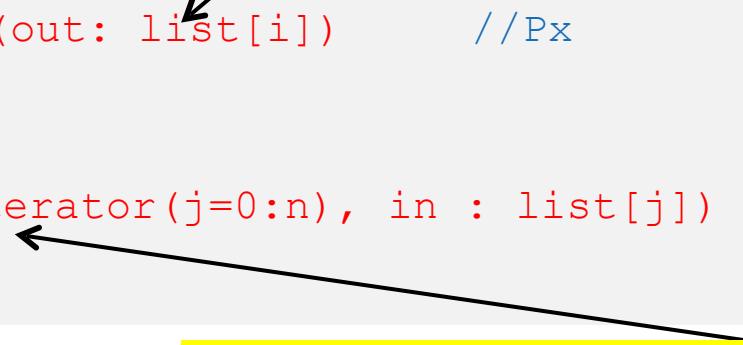
- Iterators + deps: a way to define a dynamic number of dependences

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

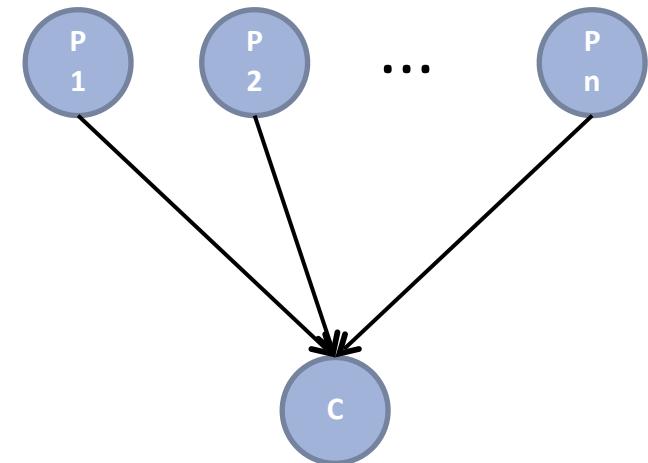
#pragma omp parallel
#pragma omp single
{
    for (int i = 0; i < n; ++i)
        #pragma omp task depend(out: list[i])           // Px
        compute_elem(list[i]);

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

It seems innocent but it's not:  
depend(out: list.operator[](i))

 // Px

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



# *Use case*

# Use case: intro to Gauss-seidel

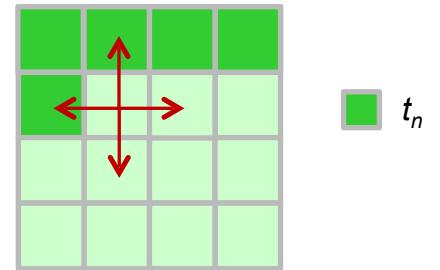
```
void serial_gauss_seidel(int tsteps, int size, int (*p) [size]) {
    for (int t = 0; t < tsteps; ++t) {
        for (int i = 1; i < size-1; ++i) {
            for (int j = 1; j < size-1; ++j) {
                p[i][j] = 0.25 * (p[i][j-1] + // left
                                    p[i][j+1] + // right
                                    p[i-1][j] + // top
                                    p[i+1][j]); // bottom
            }
        }
    }
}
```

# Use case: intro to Gauss-seidel

```
void serial_gauss_seidel(int tsteps, int size, int (*p) [size]) {  
    for (int t = 0; t < tsteps; ++t) {  
        for (int i = 1; i < size-1; ++i) {  
            for (int j = 1; j < size-1; ++j) {  
                p[i][j] = 0.25 * (p[i][j-1] // left  
                                    + p[i][j+1] // right  
                                    + p[i-1][j] // top  
                                    + p[i+1][j]); // bottom  
            }  
        }  
    }  
}
```

## Access pattern analysis

*For a specific  $t, i$  and  $j$*



Each cell depends on:

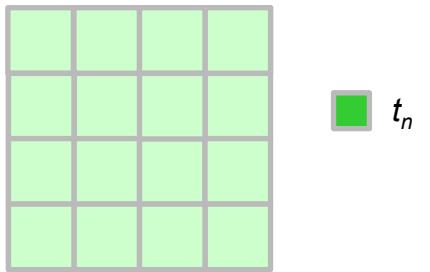
- two cells (north & west) that are computed in the current time step, and
- two cells (south & east) that were computed in the previous time step

# Use case: Gauss-seidel (2)

```
void serial_gauss_seidel(int tsteps, int size, int (*p) [size]) {  
    for (int t = 0; t < tsteps; ++t) {  
        for (int i = 1; i < size-1; ++i) {  
            for (int j = 1; j < size-1; ++j) {  
                p[i][j] = 0.25 * (p[i][j-1] + // left  
                                    p[i][j+1] + // right  
                                    p[i-1][j] + // top  
                                    p[i+1][j]); // bottom  
            }  
        }  
    }  
}
```

## 1<sup>st</sup> parallelization strategy

For an specific  $t$

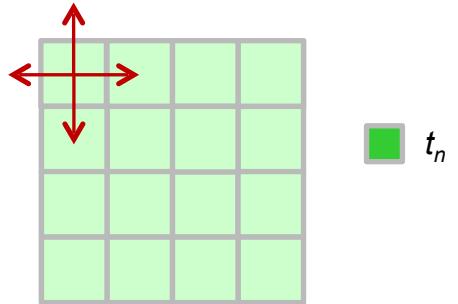


# Use case: Gauss-seidel (2)

```
void serial_gauss_seidel(int tsteps, int size, int (*p) [size]) {  
    for (int t = 0; t < tsteps; ++t) {  
        for (int i = 1; i < size-1; ++i) {  
            for (int j = 1; j < size-1; ++j) {  
                p[i][j] = 0.25 * (p[i][j-1] + // left  
                                    p[i][j+1] + // right  
                                    p[i-1][j] + // top  
                                    p[i+1][j]); // bottom  
            }  
        }  
    }  
}
```

## 1<sup>st</sup> parallelization strategy

For an specific  $t$

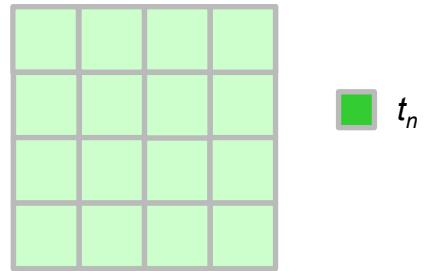


# Use case: Gauss-seidel (2)

```
void serial_gauss_seidel(int tsteps, int size, int (*p) [size]) {  
    for (int t = 0; t < tsteps; ++t) {  
        for (int i = 1; i < size-1; ++i) {  
            for (int j = 1; j < size-1; ++j) {  
                p[i][j] = 0.25 * (p[i][j-1] + // left  
                                    p[i][j+1] + // right  
                                    p[i-1][j] + // top  
                                    p[i+1][j]); // bottom  
            }  
        }  
    }  
}
```

## 1<sup>st</sup> parallelization strategy

For an specific  $t$

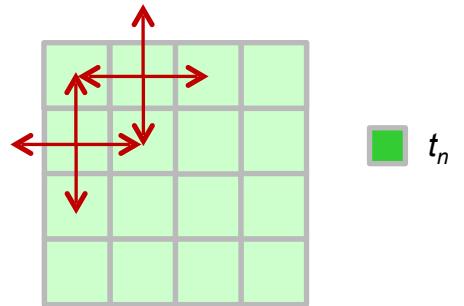


# Use case: Gauss-seidel (2)

```
void serial_gauss_seidel(int tsteps, int size, int (*p) [size]) {  
    for (int t = 0; t < tsteps; ++t) {  
        for (int i = 1; i < size-1; ++i) {  
            for (int j = 1; j < size-1; ++j) {  
                p[i][j] = 0.25 * (p[i][j-1] + // left  
                                    p[i][j+1] + // right  
                                    p[i-1][j] + // top  
                                    p[i+1][j]); // bottom  
            }  
        }  
    }  
}
```

## 1<sup>st</sup> parallelization strategy

For an specific  $t$

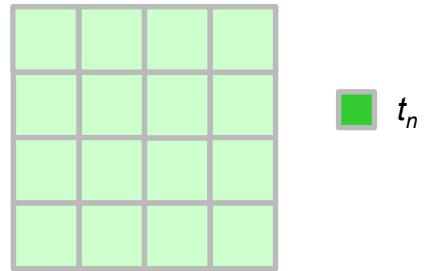


# Use case: Gauss-seidel (2)

```
void serial_gauss_seidel(int tsteps, int size, int (*p) [size]) {  
    for (int t = 0; t < tsteps; ++t) {  
        for (int i = 1; i < size-1; ++i) {  
            for (int j = 1; j < size-1; ++j) {  
                p[i][j] = 0.25 * (p[i][j-1] + // left  
                                    p[i][j+1] + // right  
                                    p[i-1][j] + // top  
                                    p[i+1][j]); // bottom  
            }  
        }  
    }  
}
```

## 1<sup>st</sup> parallelization strategy

For an specific  $t$

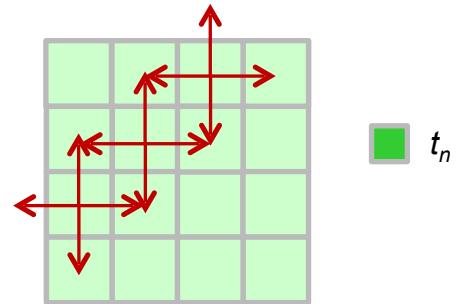


# Use case: Gauss-seidel (2)

```
void serial_gauss_seidel(int tsteps, int size, int (*p) [size]) {  
    for (int t = 0; t < tsteps; ++t) {  
        for (int i = 1; i < size-1; ++i) {  
            for (int j = 1; j < size-1; ++j) {  
                p[i][j] = 0.25 * (p[i][j-1] + // left  
                                    p[i][j+1] + // right  
                                    p[i-1][j] + // top  
                                    p[i+1][j]); // bottom  
            }  
        }  
    }  
}
```

## 1<sup>st</sup> parallelization strategy

For an specific  $t$

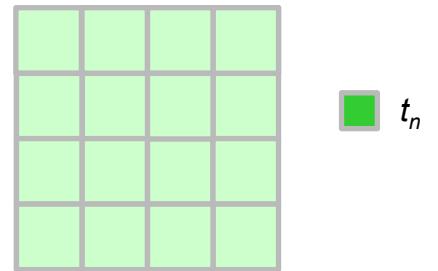


# Use case: Gauss-seidel (2)

```
void serial_gauss_seidel(int tsteps, int size, int (*p) [size]) {
    for (int t = 0; t < tsteps; ++t) {
        for (int i = 1; i < size-1; ++i) {
            for (int j = 1; j < size-1; ++j) {
                p[i][j] = 0.25 * (p[i][j-1] + // left
                                    p[i][j+1] + // right
                                    p[i-1][j] + // top
                                    p[i+1][j]); // bottom
            }
        }
    }
}
```

## 1<sup>st</sup> parallelization strategy

For an specific  $t$

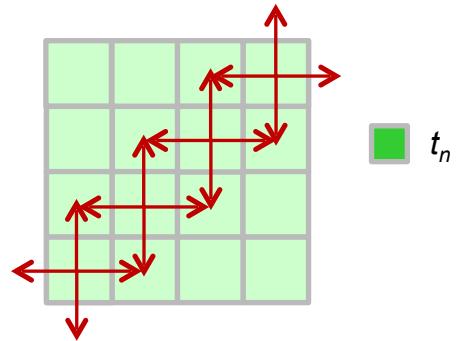


# Use case: Gauss-seidel (2)

```
void serial_gauss_seidel(int tsteps, int size, int (*p) [size]) {  
    for (int t = 0; t < tsteps; ++t) {  
        for (int i = 1; i < size-1; ++i) {  
            for (int j = 1; j < size-1; ++j) {  
                p[i][j] = 0.25 * (p[i][j-1] + // left  
                                    p[i][j+1] + // right  
                                    p[i-1][j] + // top  
                                    p[i+1][j]); // bottom  
            }  
        }  
    }  
}
```

## 1<sup>st</sup> parallelization strategy

For an specific  $t$

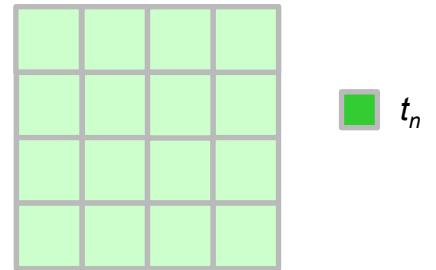


# Use case: Gauss-seidel (2)

```
void serial_gauss_seidel(int tsteps, int size, int (*p) [size]) {  
    for (int t = 0; t < tsteps; ++t) {  
        for (int i = 1; i < size-1; ++i) {  
            for (int j = 1; j < size-1; ++j) {  
                p[i][j] = 0.25 * (p[i][j-1] + // left  
                                    p[i][j+1] + // right  
                                    p[i-1][j] + // top  
                                    p[i+1][j]); // bottom  
            }  
        }  
    }  
}
```

## 1<sup>st</sup> parallelization strategy

For an specific  $t$



We can exploit the **wavefront** to obtain parallelism!!

# Use case : Gauss-seidel (3)

```
void gauss_seidel(int tsteps, int size, int TS, int (*p)[size]) {
    int NB = size / TS;
    #pragma omp parallel
    for (int t = 0; t < tsteps; ++t) {
        // First NB diagonals
        for (int diag = 0; diag < NB; ++diag) {
            #pragma omp for
            for (int d = 0; d <= diag; ++d) {
                int ii = d;
                int jj = diag - d;
                for (int i = 1+ii*TS; i < ((ii+1)*TS); ++i)
                    for (int j = 1+jj*TS; i < ((jj+1)*TS); ++j)
                        p[i][j] = 0.25 * (p[i][j-1] + p[i][j+1] +
                                           p[i-1][j] + p[i+1][j]);
            }
        }
        // Lasts NB diagonals
        for (int diag = NB-1; diag >= 0; --diag) {
            // Similar code to the previous loop
        }
    }
}
```

# Use case : Gauss-seidel (4)

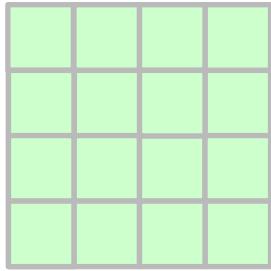
```
void serial_gauss_seidel(int tsteps, int size, int (*p) [size]) {
    for (int t = 0; t < tsteps; ++t) {
        for (int i = 1; i < size-1; ++i) {
            for (int j = 1; j < size-1; ++j) {
                p[i][j] = 0.25 * (p[i][j-1] + // left
                                    p[i][j+1] + // right
                                    p[i-1][j] + // top
                                    p[i+1][j]); // bottom
            }
        }
    }
}
```

# Use case : Gauss-seidel (4)

```
void serial_gauss_seidel(int tsteps, int size, int (*p) [size]) {  
    for (int t = 0; t < tsteps; ++t) {  
        for (int i = 1; i < size-1; ++i) {  
            for (int j = 1; j < size-1; ++j) {  
                p[i][j] = 0.25 * (p[i][j-1] + // left  
                                    p[i][j+1] + // right  
                                    p[i-1][j] + // top  
                                    p[i+1][j]); // bottom  
            }  
        }  
    }  
}
```

## 2<sup>nd</sup> parallelization strategy

*multiple time iterations*

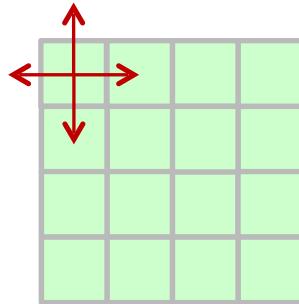


# Use case : Gauss-seidel (4)

```
void serial_gauss_seidel(int tsteps, int size, int (*p) [size]) {  
    for (int t = 0; t < tsteps; ++t) {  
        for (int i = 1; i < size-1; ++i) {  
            for (int j = 1; j < size-1; ++j) {  
                p[i][j] = 0.25 * (p[i][j-1] + // left  
                                    p[i][j+1] + // right  
                                    p[i-1][j] + // top  
                                    p[i+1][j]); // bottom  
            }  
        }  
    }  
}
```

## 2<sup>nd</sup> parallelization strategy

*multiple time iterations*

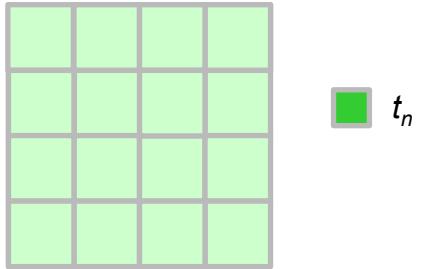


# Use case : Gauss-seidel (4)

```
void serial_gauss_seidel(int tsteps, int size, int (*p) [size]) {  
    for (int t = 0; t < tsteps; ++t) {  
        for (int i = 1; i < size-1; ++i) {  
            for (int j = 1; j < size-1; ++j) {  
                p[i][j] = 0.25 * (p[i][j-1] + // left  
                                    p[i][j+1] + // right  
                                    p[i-1][j] + // top  
                                    p[i+1][j]); // bottom  
            }  
        }  
    }  
}
```

## 2<sup>nd</sup> parallelization strategy

*multiple time iterations*

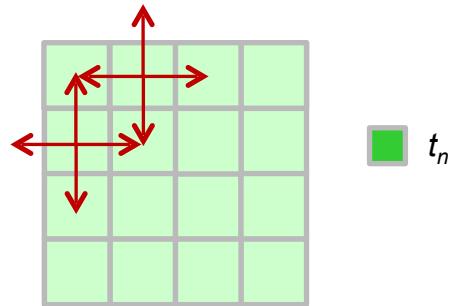


# Use case : Gauss-seidel (4)

```
void serial_gauss_seidel(int tsteps, int size, int (*p) [size]) {  
    for (int t = 0; t < tsteps; ++t) {  
        for (int i = 1; i < size-1; ++i) {  
            for (int j = 1; j < size-1; ++j) {  
                p[i][j] = 0.25 * (p[i][j-1] + // left  
                                    p[i][j+1] + // right  
                                    p[i-1][j] + // top  
                                    p[i+1][j]); // bottom  
            }  
        }  
    }  
}
```

## 2<sup>nd</sup> parallelization strategy

*multiple time iterations*

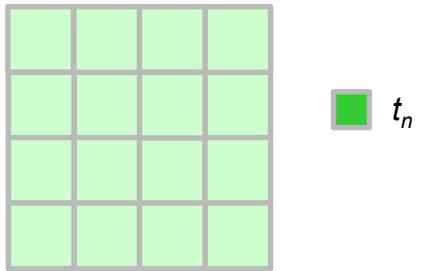


# Use case : Gauss-seidel (4)

```
void serial_gauss_seidel(int tsteps, int size, int (*p) [size]) {  
    for (int t = 0; t < tsteps; ++t) {  
        for (int i = 1; i < size-1; ++i) {  
            for (int j = 1; j < size-1; ++j) {  
                p[i][j] = 0.25 * (p[i][j-1] + // left  
                                    p[i][j+1] + // right  
                                    p[i-1][j] + // top  
                                    p[i+1][j]); // bottom  
            }  
        }  
    }  
}
```

## 2<sup>nd</sup> parallelization strategy

*multiple time iterations*

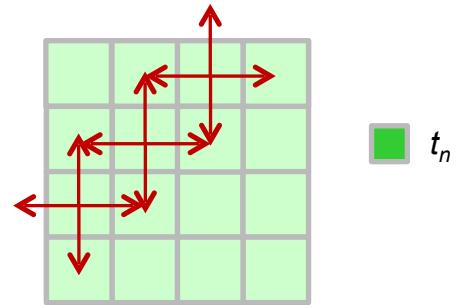


# Use case : Gauss-seidel (4)

```
void serial_gauss_seidel(int tsteps, int size, int (*p) [size]) {  
    for (int t = 0; t < tsteps; ++t) {  
        for (int i = 1; i < size-1; ++i) {  
            for (int j = 1; j < size-1; ++j) {  
                p[i][j] = 0.25 * (p[i][j-1] + // left  
                                    p[i][j+1] + // right  
                                    p[i-1][j] + // top  
                                    p[i+1][j]); // bottom  
            }  
        }  
    }  
}
```

## 2<sup>nd</sup> parallelization strategy

*multiple time iterations*

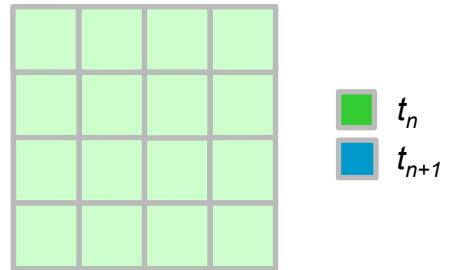


# Use case : Gauss-seidel (4)

```
void serial_gauss_seidel(int tsteps, int size, int (*p) [size]) {  
    for (int t = 0; t < tsteps; ++t) {  
        for (int i = 1; i < size-1; ++i) {  
            for (int j = 1; j < size-1; ++j) {  
                p[i][j] = 0.25 * (p[i][j-1] + // left  
                                    p[i][j+1] + // right  
                                    p[i-1][j] + // top  
                                    p[i+1][j]); // bottom  
            }  
        }  
    }  
}
```

## 2<sup>nd</sup> parallelization strategy

*multiple time iterations*

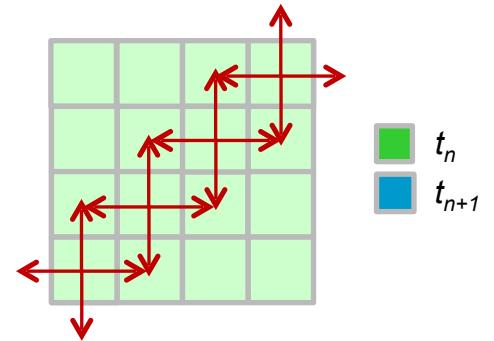


# Use case : Gauss-seidel (4)

```
void serial_gauss_seidel(int tsteps, int size, int (*p) [size]) {  
    for (int t = 0; t < tsteps; ++t) {  
        for (int i = 1; i < size-1; ++i) {  
            for (int j = 1; j < size-1; ++j) {  
                p[i][j] = 0.25 * (p[i][j-1] + // left  
                                    p[i][j+1] + // right  
                                    p[i-1][j] + // top  
                                    p[i+1][j]); // bottom  
            }  
        }  
    }  
}
```

## 2<sup>nd</sup> parallelization strategy

*multiple time iterations*

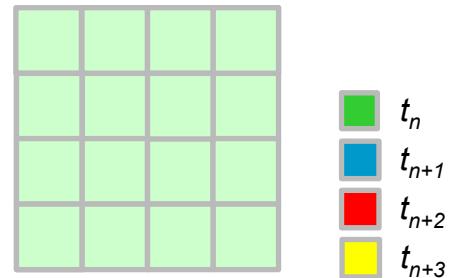


# Use case : Gauss-seidel (4)

```
void serial_gauss_seidel(int tsteps, int size, int (*p) [size]) {  
    for (int t = 0; t < tsteps; ++t) {  
        for (int i = 1; i < size-1; ++i) {  
            for (int j = 1; j < size-1; ++j) {  
                p[i][j] = 0.25 * (p[i][j-1] + // left  
                                    p[i][j+1] + // right  
                                    p[i-1][j] + // top  
                                    p[i+1][j]); // bottom  
            }  
        }  
    }  
}
```

## 2<sup>nd</sup> parallelization strategy

*multiple time iterations*



We can exploit the **wavefront**  
of multiple time steps to obtain **MORE**  
**parallelism!!**

# Use case : Gauss-seidel (5)

```
void gauss_seidel(int tsteps, int size, int TS, int (*p) [size]) {
    int NB = size / TS;

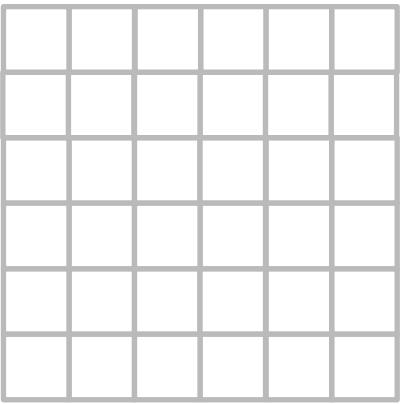
#pragma omp parallel
#pragma omp single
    for (int t = 0; t < tsteps; ++t)
        for (int ii=1; ii < size-1; ii+=TS)
            for (int jj=1; jj < size-1; jj+=TS) {
                #pragma omp task depend(inout: p[ii:TS][jj:TS])
                    depend(in: p[ii-TS:TS][jj:TS], p[ii+TS:TS][jj:TS],
                           p[ii:TS][jj-TS:TS], p[ii:TS][jj:TS])
                {
                    for (int i=ii; i<(1+ii)*TS; ++i)
                        for (int j=jj; j<(1+jj)*TS; ++j)
                            p[i][j] = 0.25 * (p[i][j-1] + p[i][j+1] +
                                              p[i-1][j] + p[i+1][j]);
                }
            }
}
```

# Use case : Gauss-seidel (5)

```
void gauss_seidel(int tsteps, int size, int TS, int (*p) [size]) {
    int NB = size / TS;

#pragma omp parallel
#pragma omp single
    for (int t = 0; t < tsteps; ++t)
        for (int ii=1; ii < size-1; ii+=TS)
            for (int jj=1; jj < size-1; jj+=TS) {
                #pragma omp task depend(inout: p[ii:TS][jj:TS])
                    depend(in: p[ii-TS:TS][jj:TS], p[ii+TS:TS][jj:TS],
                           p[ii:TS][jj-TS:TS], p[ii:TS][jj:TS])
                {
                    for (int i=ii; i<(1+ii)*TS; ++i)
                        for (int j=jj; j<(1+jj)*TS; ++j)
                            p[i][j] = 0.25 * (p[i][j-1] + p[i][j+1] +
                                              p[i-1][j] + p[i+1][j]);
                }
            }
        }
}
```

inner matrix region

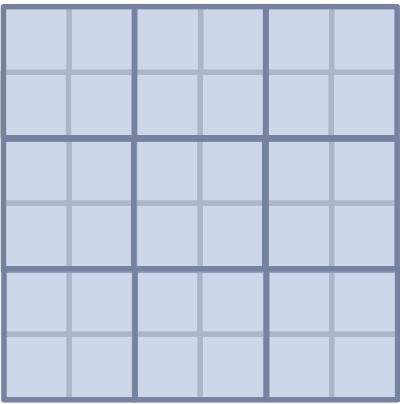


# Use case : Gauss-seidel (5)

```
void gauss_seidel(int tsteps, int size, int TS, int (*p) [size]) {
    int NB = size / TS;

#pragma omp parallel
#pragma omp single
for (int t = 0; t < tsteps; ++t)
    for (int ii=1; ii < size-1; ii+=TS)
        for (int jj=1; jj < size-1; jj+=TS) {
            #pragma omp task depend(inout: p[ii:TS][jj:TS])
            depend(in: p[ii-TS:TS][jj:TS], p[ii+TS:TS][jj:TS],
                   p[ii:TS][jj-TS:TS], p[ii:TS][jj:TS])
            {
                for (int i=ii; i<(1+ii)*TS; ++i)
                    for (int j=jj; j<(1+jj)*TS; ++j)
                        p[i][j] = 0.25 * (p[i][j-1] + p[i][j+1] +
                                           p[i-1][j] + p[i+1][j]);
            }
        }
}
```

inner matrix region

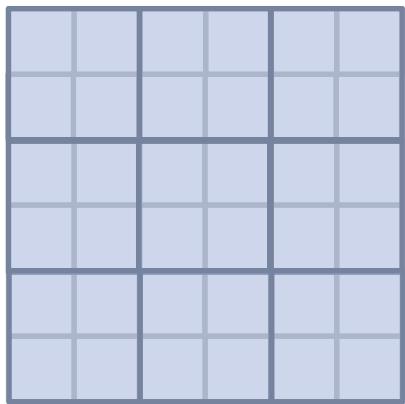


# Use case : Gauss-seidel (5)

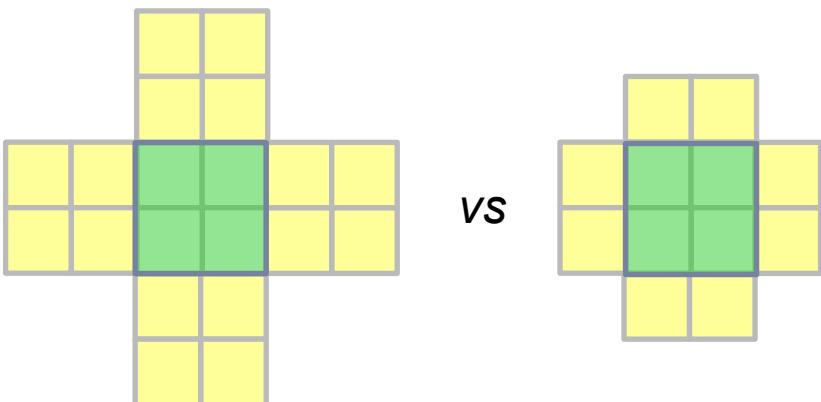
```
void gauss_seidel(int tsteps, int size, int TS, int (*p) [size]) {
    int NB = size / TS;

#pragma omp parallel
#pragma omp single
    for (int t = 0; t < tsteps; ++t)
        for (int ii=1; ii < size-1; ii+=TS)
            for (int jj=1; jj < size-1; jj+=TS) {
                #pragma omp task depend(inout: p[ii:TS][jj:TS])
                    depend(in: p[ii-TS:TS][jj:TS], p[ii+TS:TS][jj:TS],
                           p[ii:TS][jj-TS:TS], p[ii:TS][jj:TS])
                {
                    for (int i=ii; i<(1+ii)*TS; ++i)
                        for (int j=jj; j<(1+jj)*TS; ++j)
                            p[i][j] = 0.25 * (p[i][j-1] + p[i][j+1] +
                                              p[i-1][j] + p[i+1][j]);
                }
            }
}
```

inner matrix region

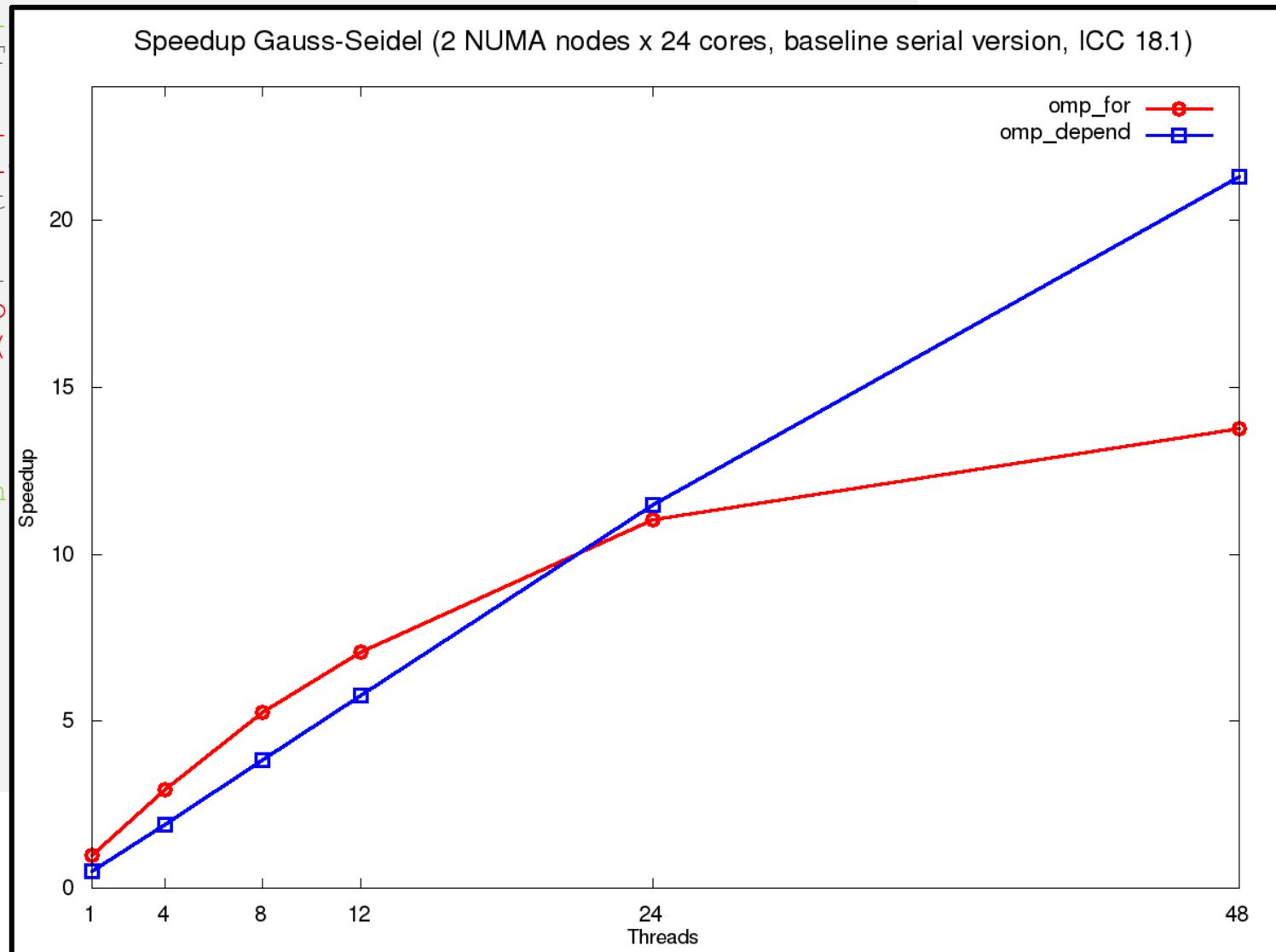


Q: Why do the input dependences depend on the whole block rather than just a column/row?



# Use case : Gauss-seidel (5)

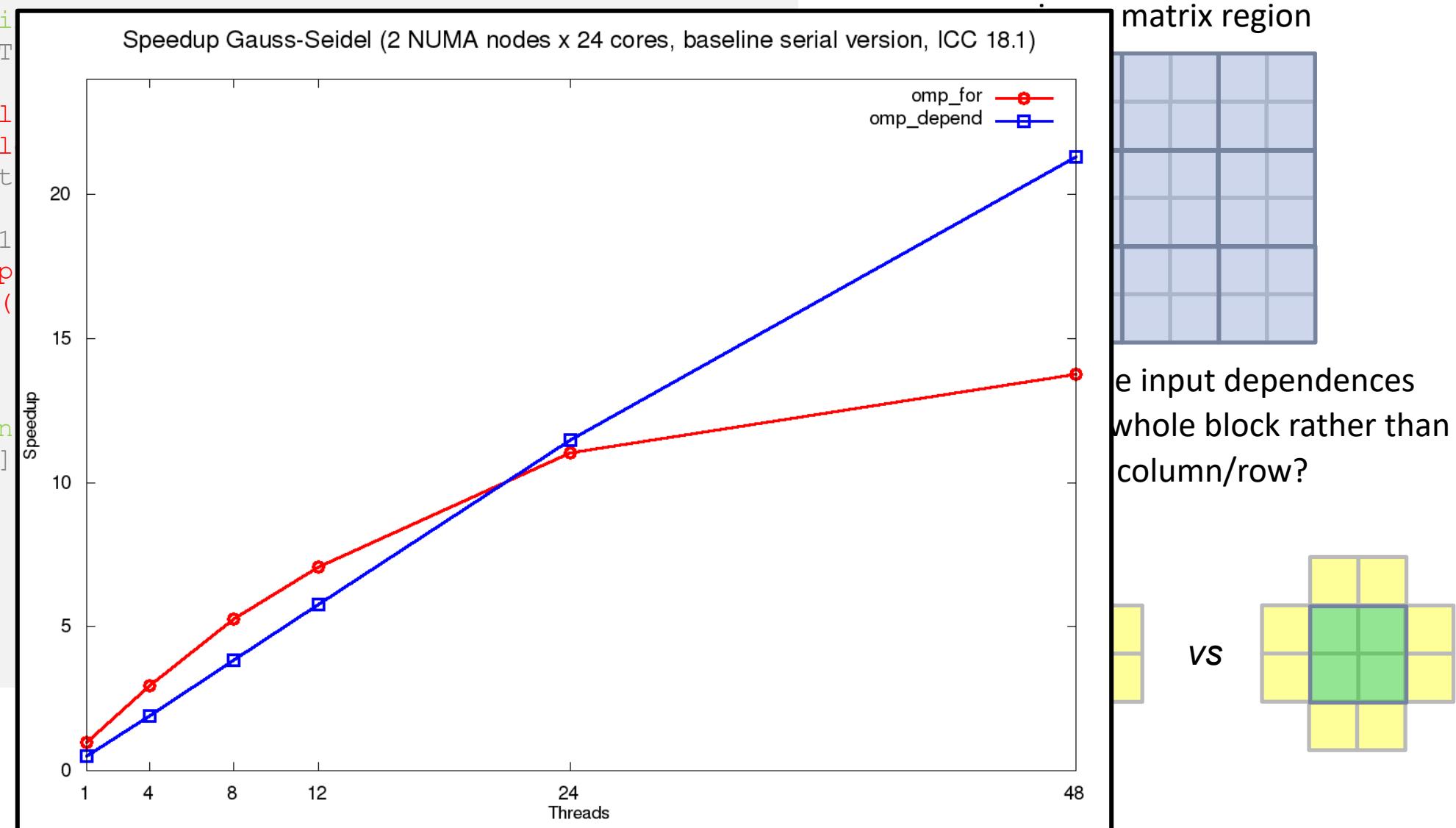
```
void gauss_seidel(i  
    int NB = size / T  
  
#pragma omp paral  
#pragma omp singl  
for (int t = 0; t  
    for (int ii=1;  
        for (int jj=1  
            #pragma omp  
                depend(  
  
    {  
        for (int  
            for (in  
                p[i]  
    }  
}  
}
```



# Use case : Gauss-seidel (5)

```
void gauss_seidel(i
    int NB = size / T

#pragma omp paral
#pragma omp singl
for (int t = 0; t
    for (int ii=1;
        for (int jj=1
            #pragma omp
                depend(
{
    for (int
        for (in
            p[i]
}
}
```



# *OpenMP 5.0: (even) more advanced features*

# Advanced features: 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;
}
```

# Improving Tasking Performance: Cutoff clauses and strategies

# *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
- (2) Try all numbers:
  - (2 a) Check Sudoku
    - If invalid: skip
    - If valid: Go to next field
- Wait for completion

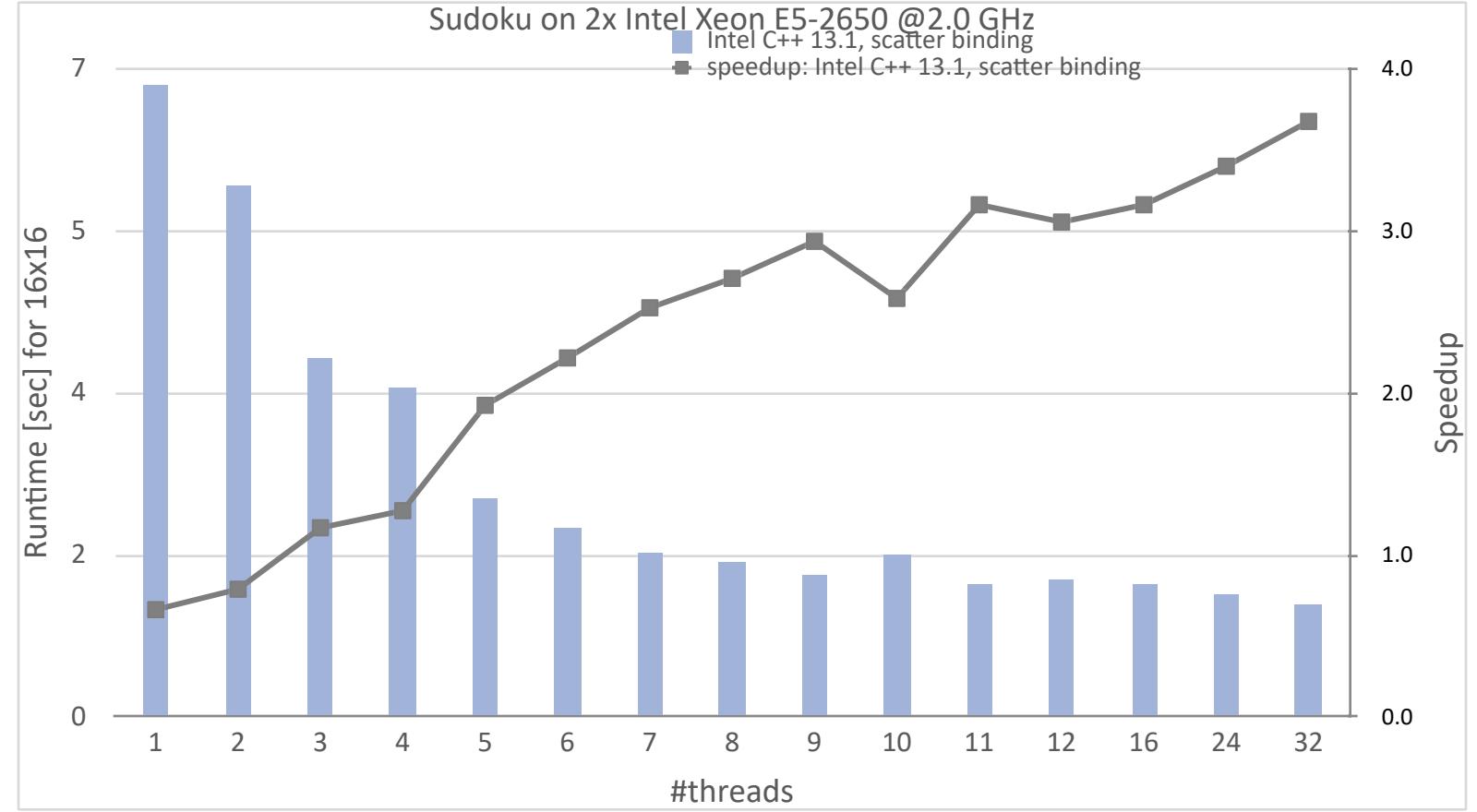
first call contained in a  
`#pragma omp parallel`  
`#pragma omp single`  
 such that one tasks starts the  
 execution of the algorithm

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

`#pragma omp taskwait`  
 wait for all child tasks

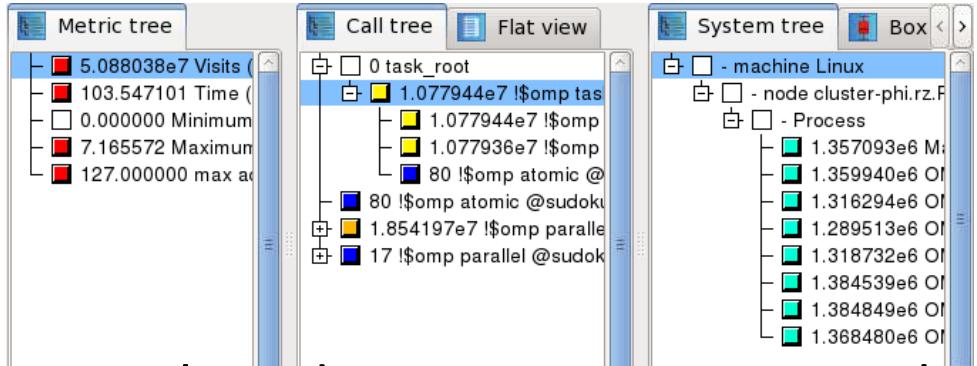
# Performance Evaluation

OpenMP®



# Performance Analysis

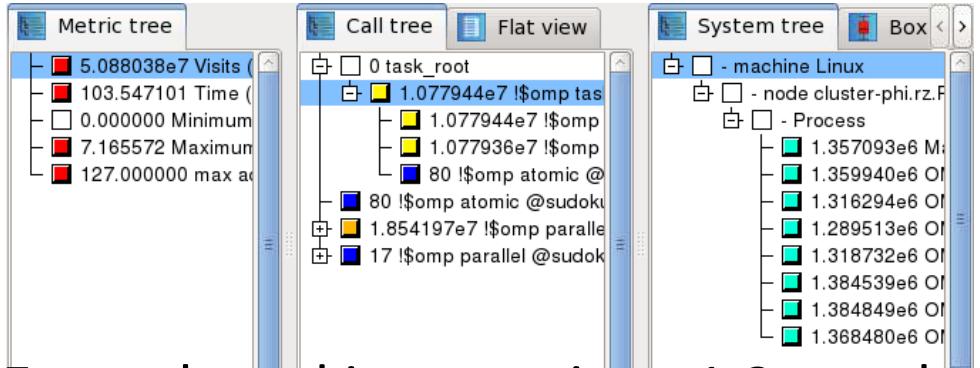
Event-based profiling provides a good overview :



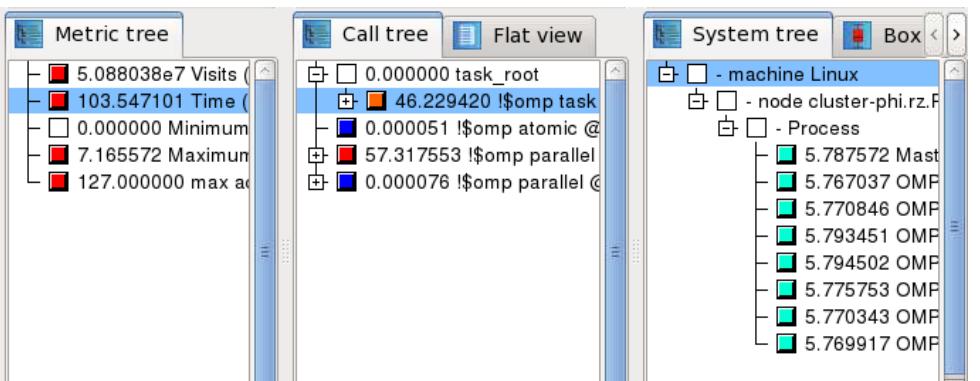
Every thread is executing ~1.3m tasks...

# Performance Analysis

Event-based profiling provides a good overview :



Every thread is executing ~1.3m tasks...

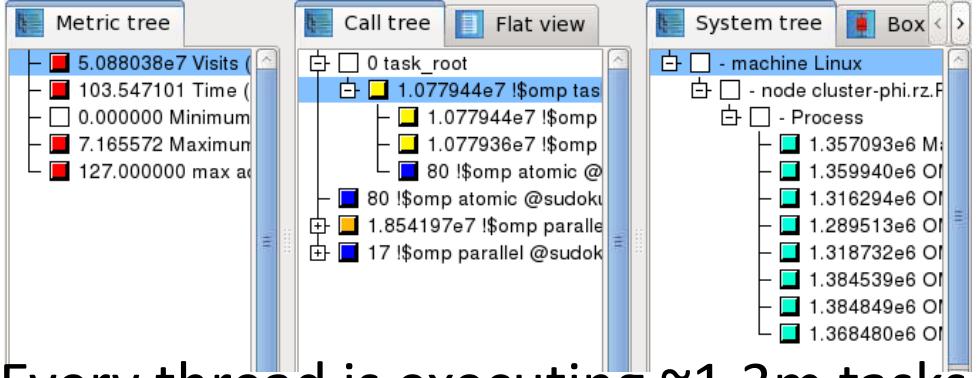


... in ~5.7 seconds.

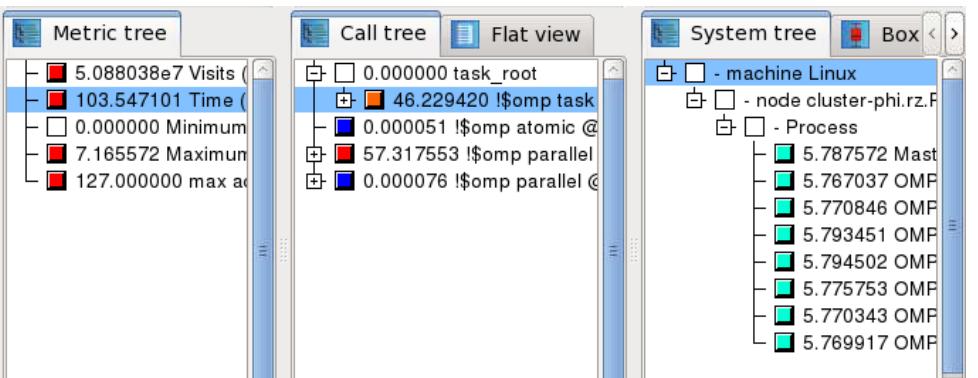
=> average duration of a task is ~4.4 µs

# Performance Analysis

Event-based profiling provides a good overview :



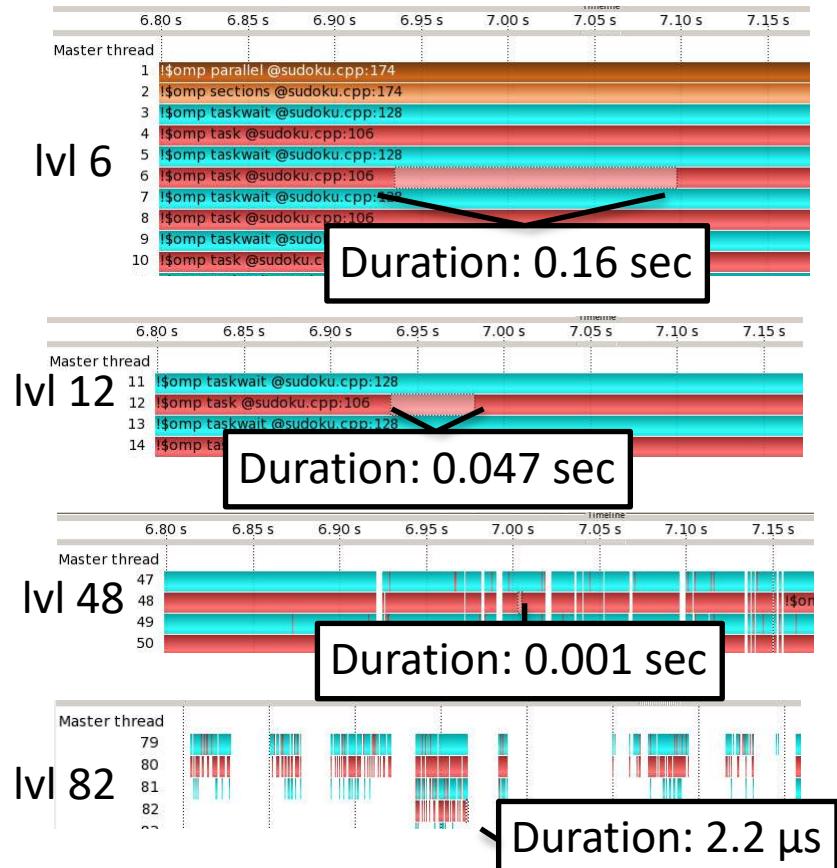
Every thread is executing ~1.3m tasks...



... in ~5.7 seconds.

=> average duration of a task is ~4.4 µs

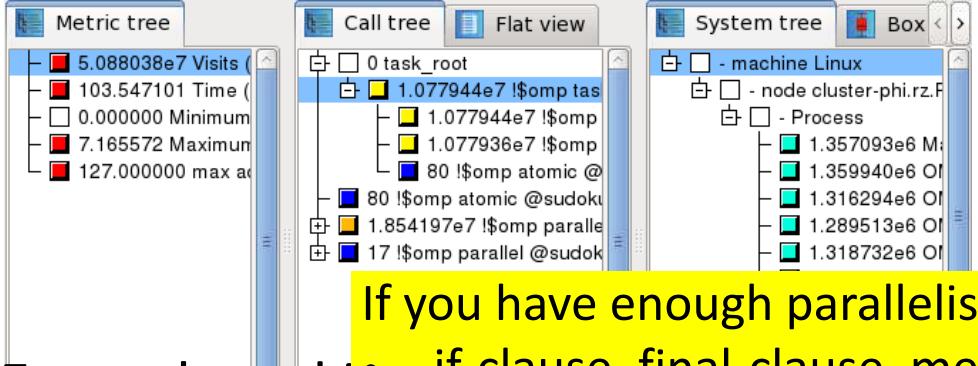
Tracing provides more details:



Tasks get much smaller down the call-stack.

# Performance Analysis

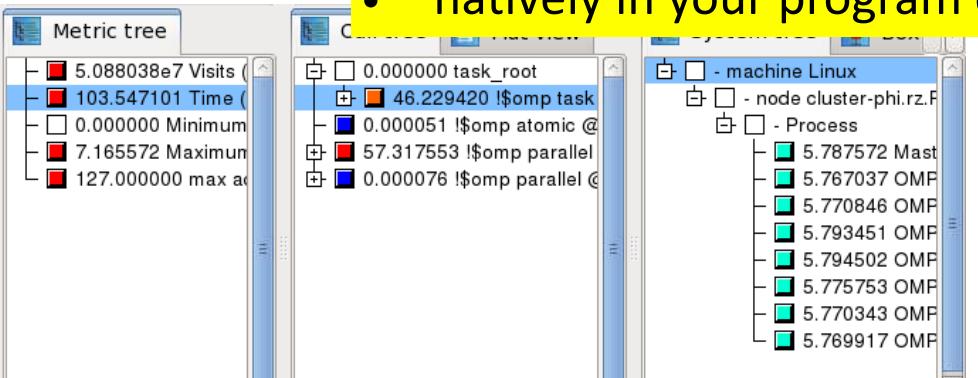
Event-based profiling provides a good overview :



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

Every thread is

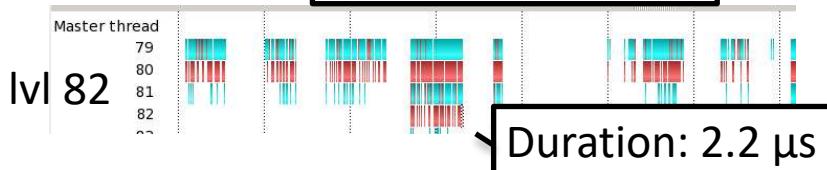
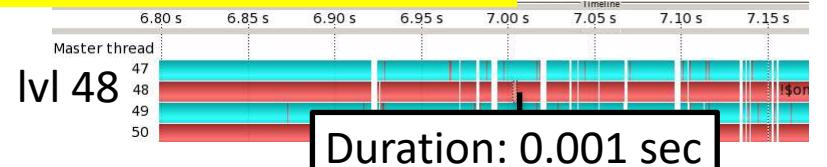
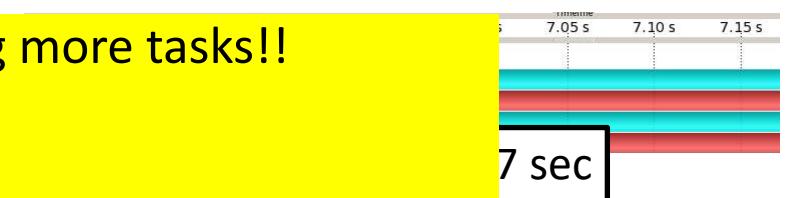
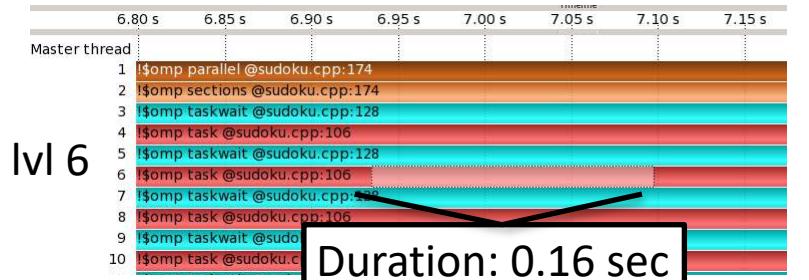
- if-clause, final-clause, mergeable-clause
- natively in your program code



... in ~5.7 seconds.

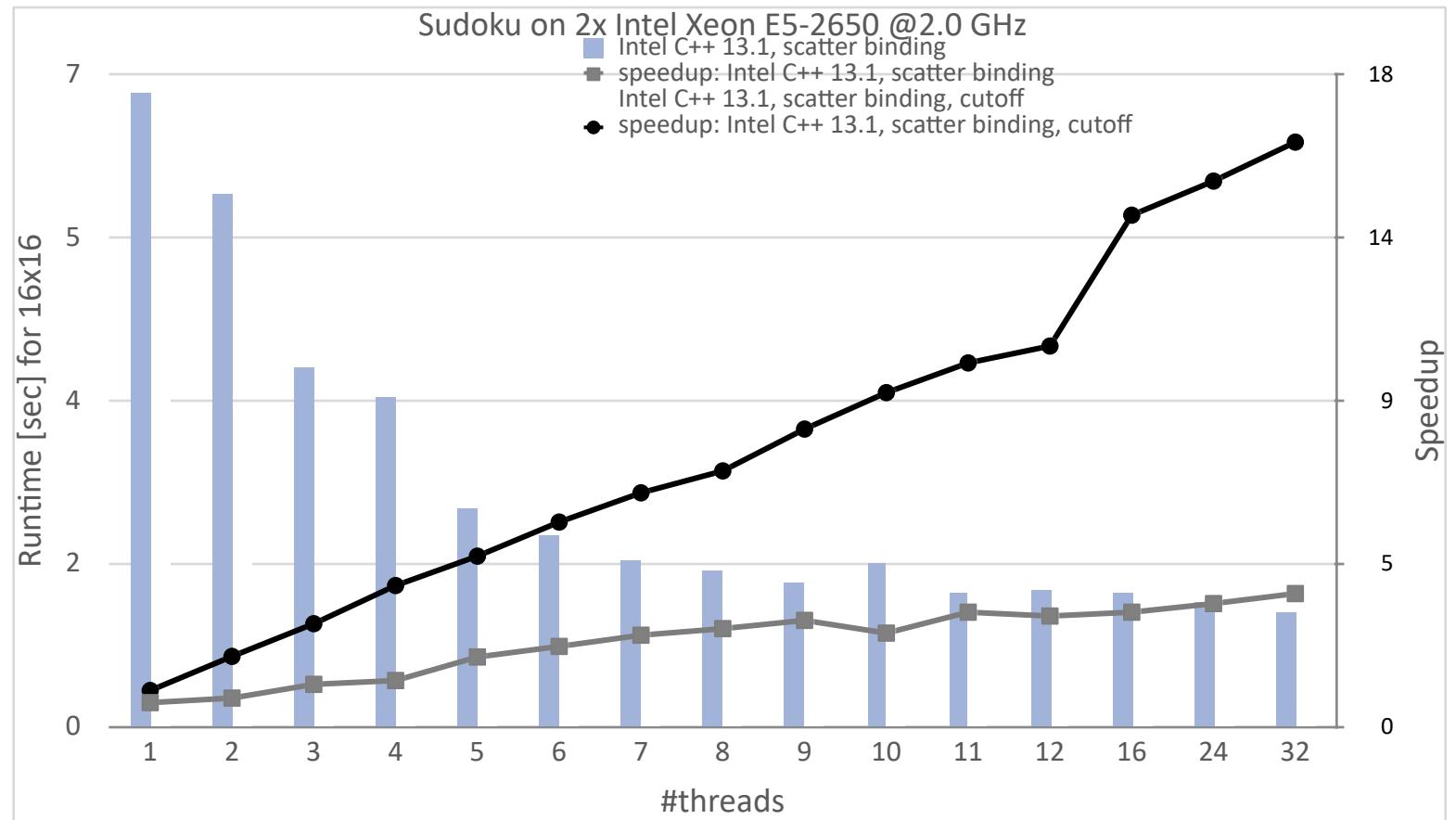
=> average duration of a task is ~4.4  $\mu$ s

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 *undeferred 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");
}
```

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

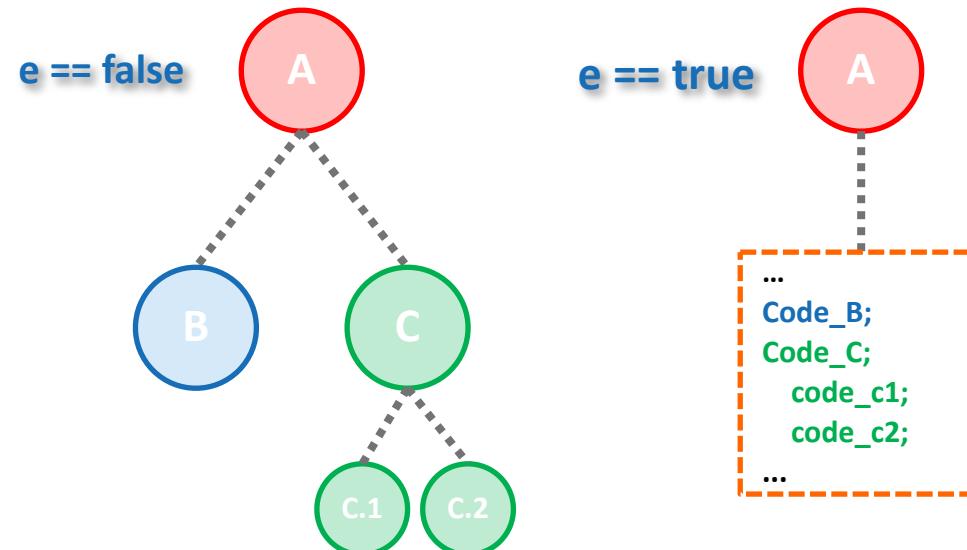
```
int foo(int x) {
    printf("entering foo function\n");
    int res = 0;
    #pragma omp task shared(res) if(false)
    {
        res += x;
    }
    printf("leaving foo function\n");
}
```

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 *unreferred* 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 *unreferred 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

# The mergeable clause

- The `mergeable` clause
  - Optimization: get rid of “data-sharing clauses are honored”
  - This optimization can only be applied in *deferred* 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 *deferred 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` =(

# *Hands-on Exercises*

# Exercises

- We have implemented a series of small hands-on examples that you can use and play with.
  - Download: `git clone https://github.com/NERSC/openmp-series-2024`
  - Subfolder: `Session-2-Tasking/exercises`, with instructions in `Exercises_OMP_2024.pdf`
  - Build: `make`
  - You can then find the compiled executable to run with the sample Slurm script
  - We use the GCC compiler mostly
- Each hands-on exercise has a folder “solution”
  - It shows the OpenMP directive that we have added
  - You can use it to cheat ☺, or to check if you came up with the same solution

# Exercises: Overview

Exercise no.	Exercise name	OpenMP Topic	Day / Order (proposal)
1	Fibonacci	Familiarize yourself with Tasking	Second day
2	For / Work-Distribution	Task + Taskloop	Second day
3	Quicksort	Tasking, Cut-off	Second day