

# Programming OpenMP

**Christian Terboven**



**Michael Klemm**



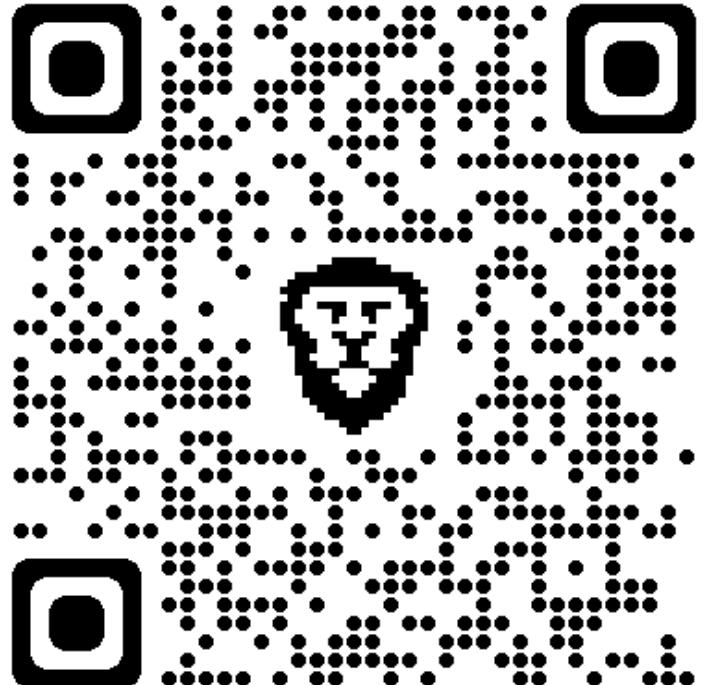
# Agenda (tentative – tell us what else you need)

	Wed 07/12	Thu 08/12
08:50 – 09:00	Time to join the training	Time to join the training
09:00 – 10:30	Introduction	SIMD and GPUs Prog. / 1
10:30 – 10:45	Break	Break
10:45 – 12:15	Tasking / 1	GPU Prog. / 2
12:15 – 13:15	Break	Break
13:15 – 14:45	Tasking / 2	Misc. OpenMP Features
14:45 – 15:00	Break	Break
15:00 – 16:30	Hands-on / 1	Hands-on / 2

Lab: hands-on time

# Material

- You can find all material on github.com:
  - Slide decks
  - Exercise tasks
  - Solutions
- <https://github.com/cterboven/OpenMP-tutorial-IT4I-2022>



# Programming OpenMP

*An Overview Of OpenMP*

**Christian Terboven**

Michael Klemm



## History

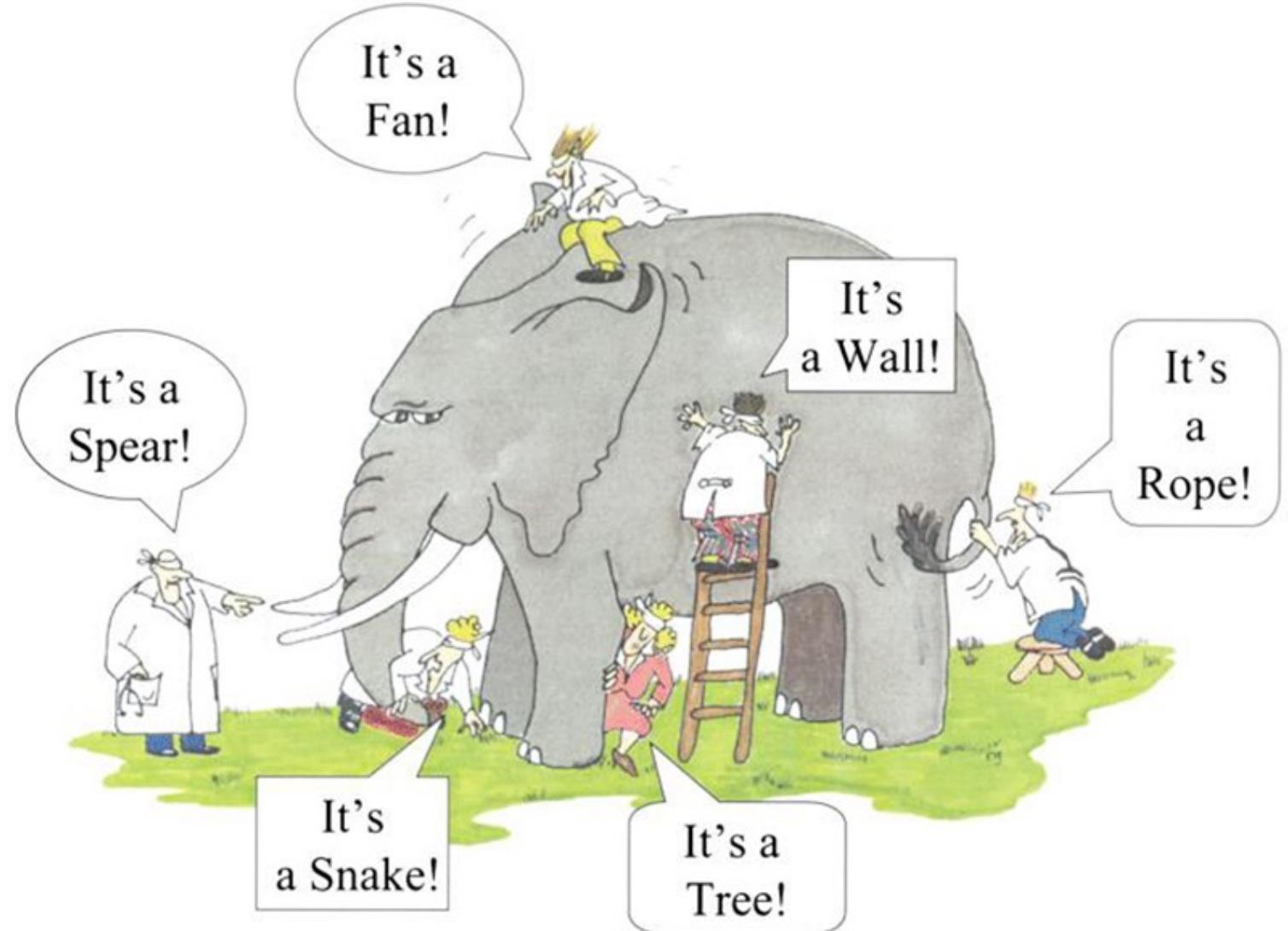
- De-facto standard for Shared-Memory Parallelization.
- 1997: OpenMP 1.0 for FORTRAN
- 1998: OpenMP 1.0 for C and C++
- 1999: OpenMP 1.1 for FORTRAN
- 2000: OpenMP 2.0 for FORTRAN
- 2002: OpenMP 2.0 for C and C++
- 2005: OpenMP 2.5 now includes both programming languages.
- 05/2008: OpenMP 3.0
- 07/2011: OpenMP 3.1
- 07/2013: OpenMP 4.0
- 11/2015: OpenMP 4.5
- 11/2018: OpenMP 5.0
- 11/2020: OpenMP 5.1
- 11/2021: OpenMP 5.2



<http://www.OpenMP.org>

## What is OpenMP?

- Parallel Region & Worksharing
- Tasking
- SIMD / Vectorization
- Accelerator Programming
- ...

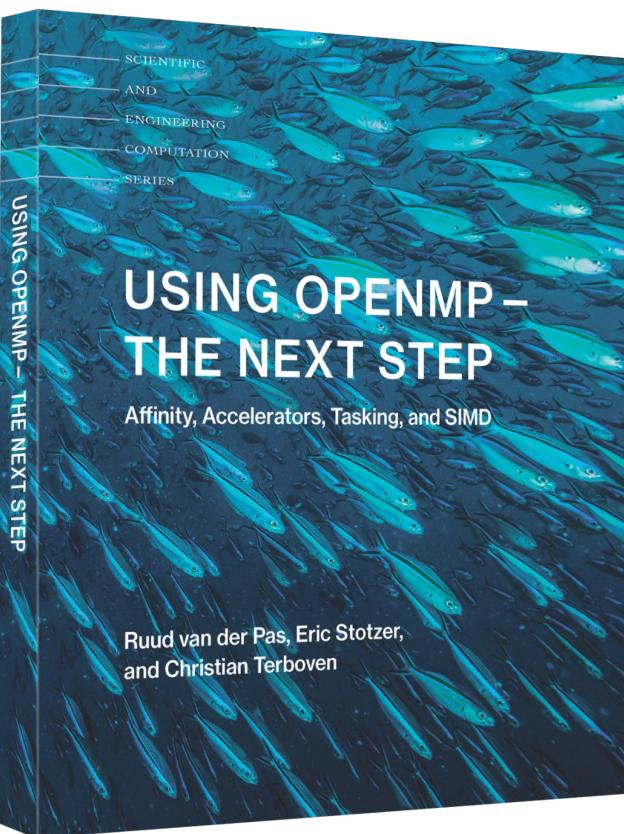


# *Get your C/C++ and Fortran Reference Guide! Covers all of OpenMP 5.1/5.2!*

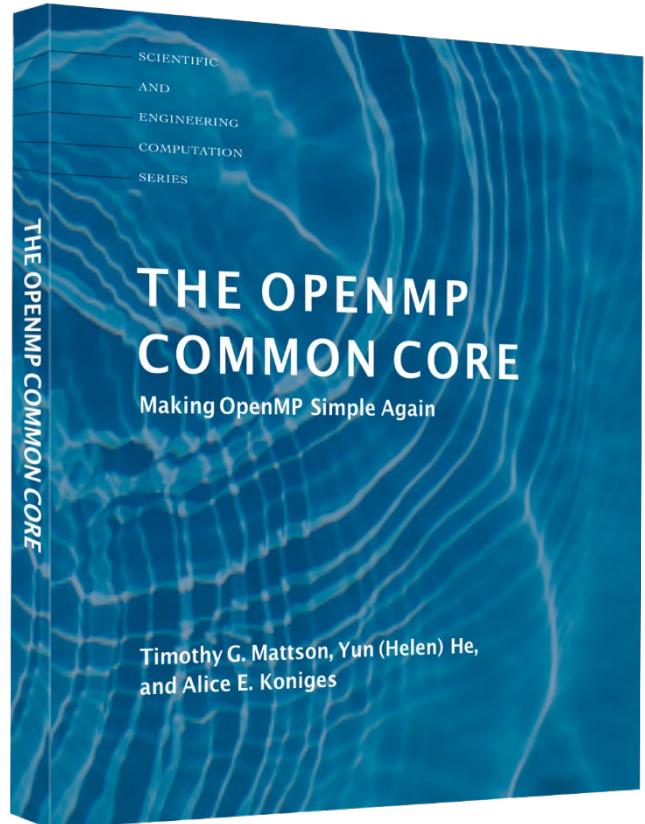
# Recent Books About OpenMP



A printed copy of the 5.1 specifications, 2020



A book that covers all of the OpenMP 4.5 features, 2017



A new book about the OpenMP Common Core, 2019

# Programming OpenMP

## *Parallel Region*

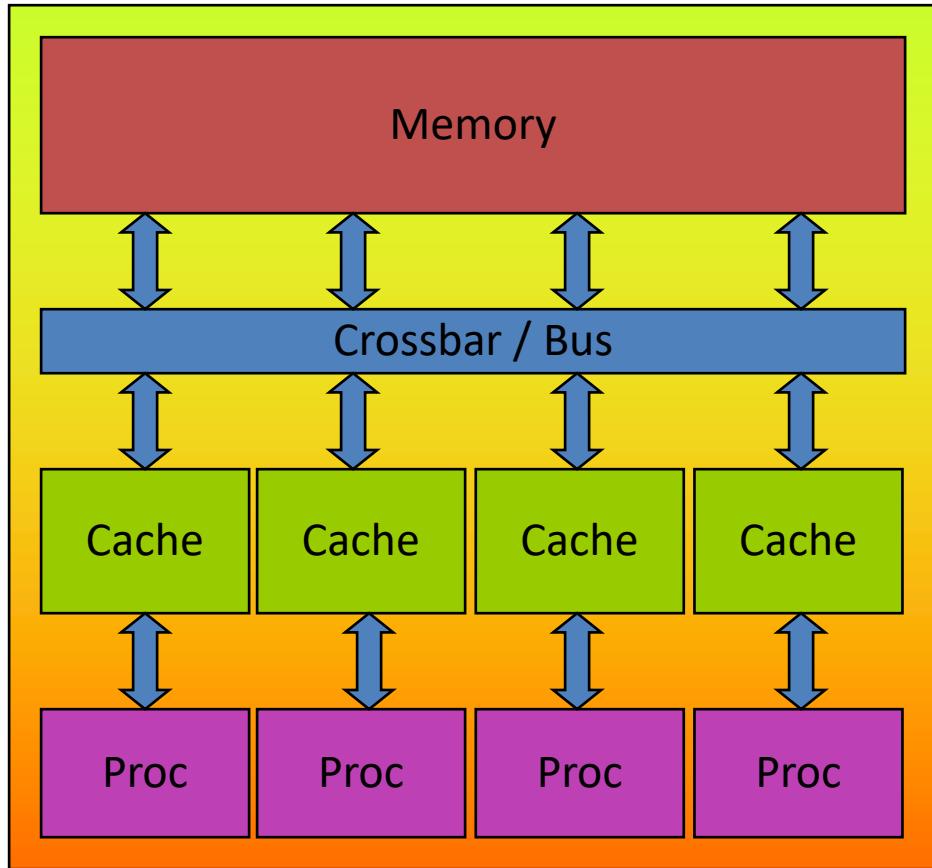
**Christian Terboven**

Michael Klemm



## OpenMP's machine model

- OpenMP: Shared-Memory Parallel Programming Model.



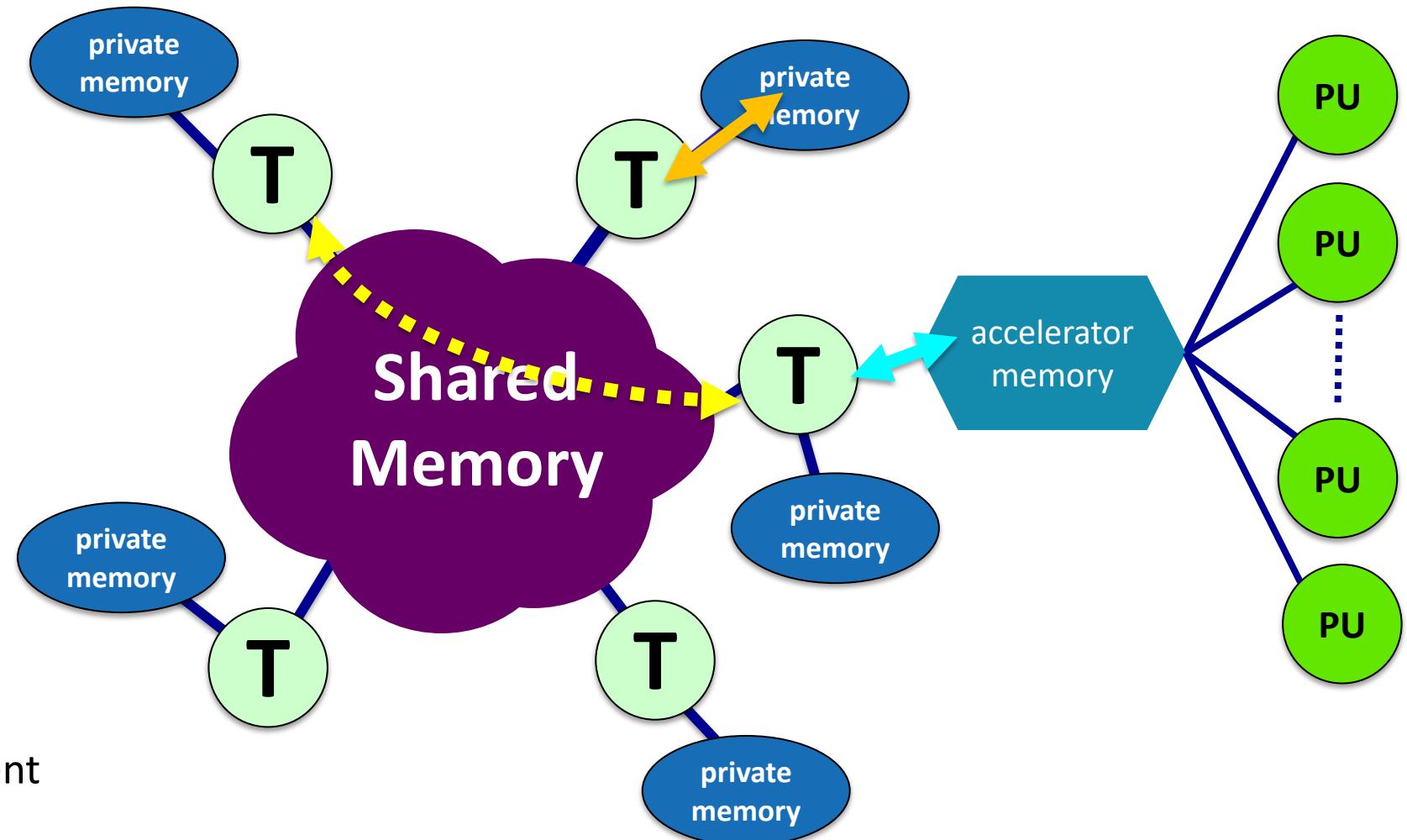
All processors/cores access a shared main memory.

Real architectures are more complex, as we will see later / as we

Parallelization in OpenMP employs multiple threads.

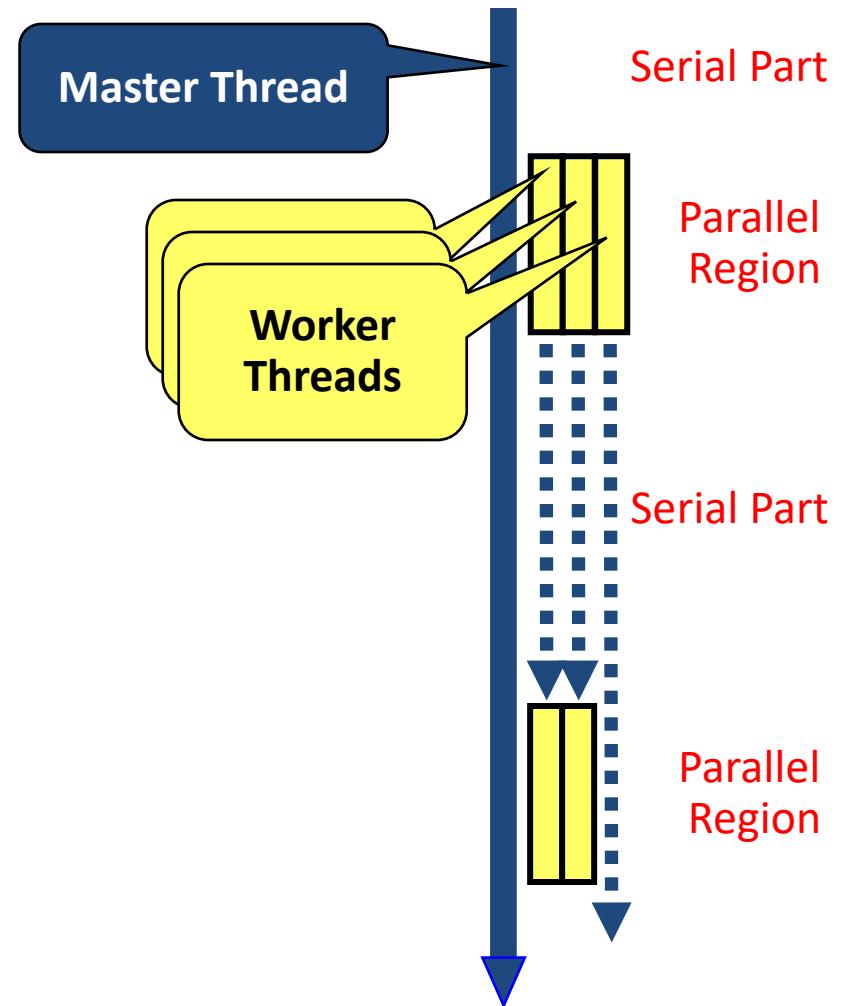
## The OpenMP Memory Model

- All threads have access to the same, globally shared memory
- Data in private memory is only accessible by the thread owning this memory
- No other thread sees the change(s) in private memory
- Data transfer is through shared memory and is 100% transparent to the application



## The OpenMP Execution Model

- OpenMP programs start with just one thread: The *Master*.
- *Worker* threads are spawned at *Parallel Regions*, together with the Master they form the *Team* of threads.
- In between Parallel Regions the Worker threads are put to sleep. The OpenMP *Runtime* takes care of all thread management work.
- Concept: *Fork-Join*.
- Allows for an incremental parallelization!



## Parallel Region and Structured Blocks

- The parallelism has to be expressed explicitly.

C/C++

```
#pragma omp parallel
{
    ...
    structured block
    ...
}
```

Fortran

```
!$omp parallel
...
structured block
...
 !$omp end parallel
```

- Structured Block*

- Exactly one entry point at the top
- Exactly one exit point at the bottom
- Branching in or out is not allowed
- Terminating the program is allowed (abort / exit)

- Specification of number of threads:*

- Environment variable: OMP\_NUM\_THREADS=...
- Or: Via num\_threads clause:  
add num\_threads (num) to the parallel construct

## Starting OpenMP Programs on Linux

- From within a shell, global setting of the number of threads:

```
export OMP_NUM_THREADS=4  
./program
```

- From within a shell, one-time setting of the number of threads:

```
OMP_NUM_THREADS=4 ./program
```

# Hello OpenMP World

# Programming OpenMP

## *Worksharing*

**Christian Terboven**

Michael Klemm



## For Worksharing

- If only the *parallel* construct is used, each thread executes the Structured Block.
- Program Speedup: *Worksharing*
- OpenMP's most common Worksharing construct: *for*

C/C++

```
int i;  
#pragma omp for  
for (i = 0; i < 100; i++)  
{  
    a[i] = b[i] + c[i];  
}
```

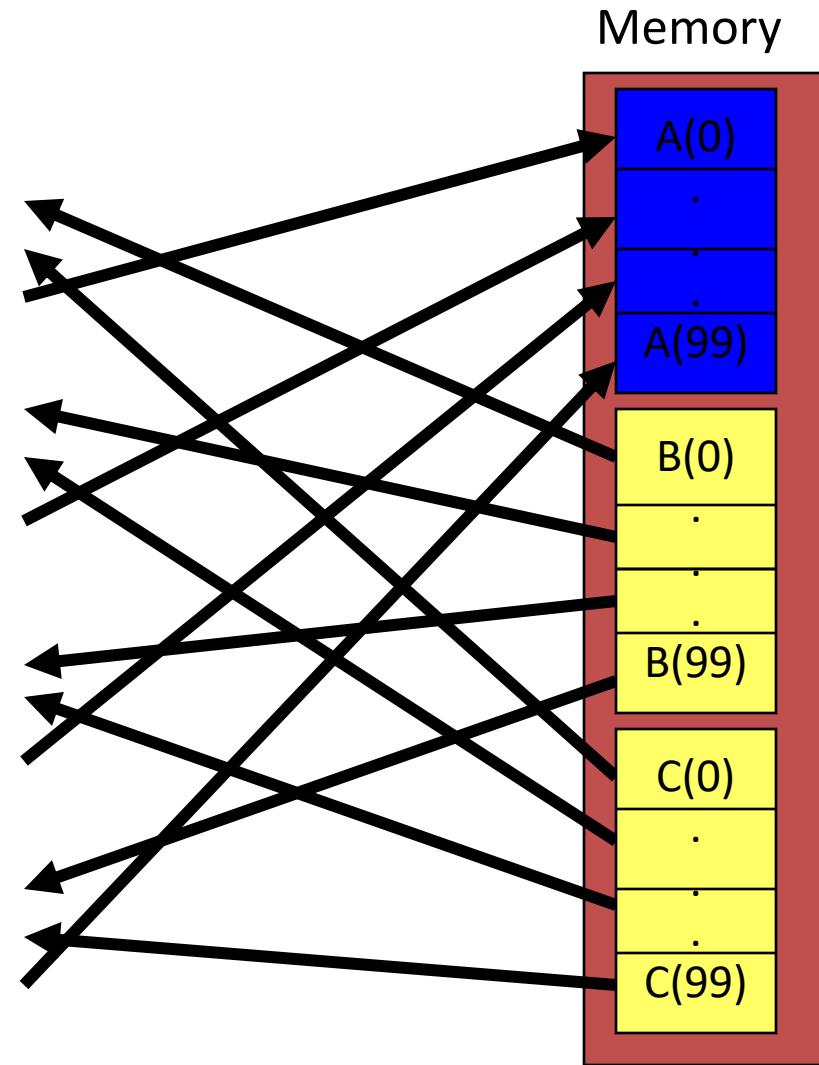
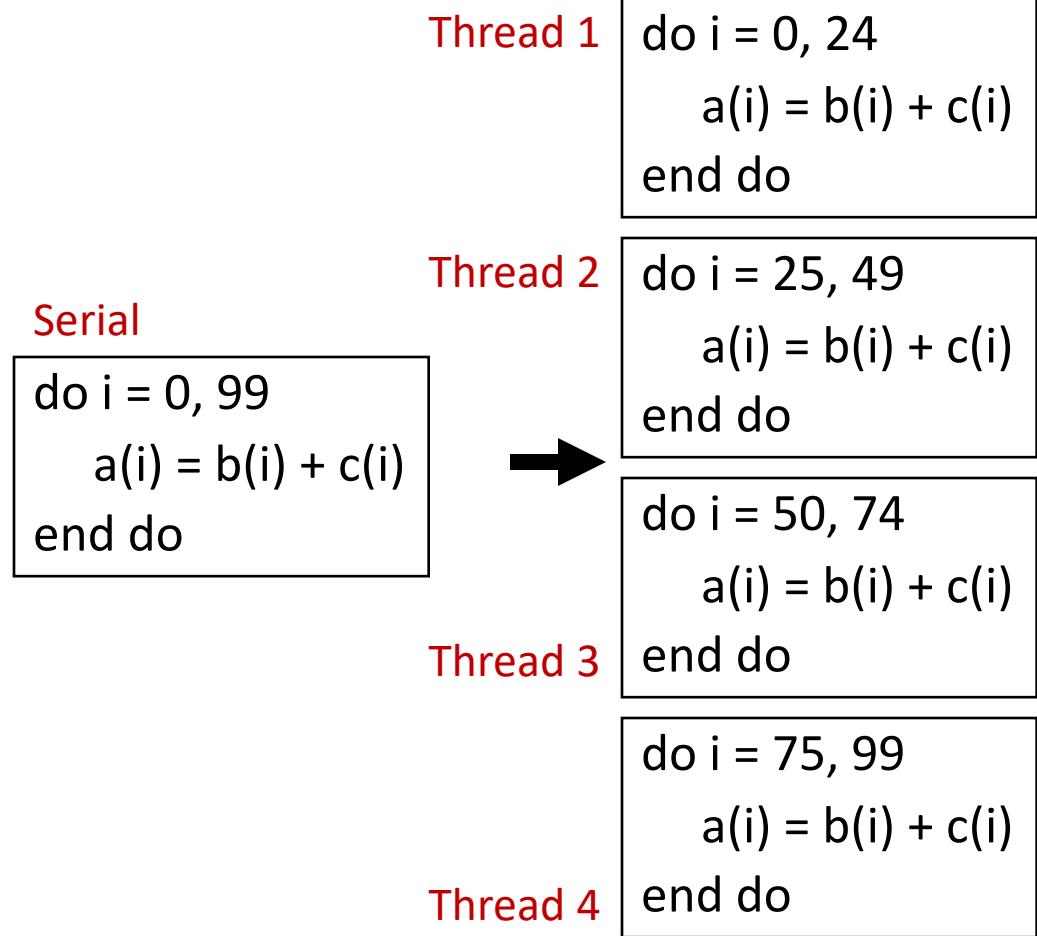
Fortran

```
INTEGER :: i  
!$omp do  
DO i = 0, 99  
    a[i] = b[i] + c[i]  
END DO
```

- Distribution of loop iterations over all threads in a Team.
  - Scheduling of the distribution can be influenced.
- 
- Loops often account for most of a program's runtime!

## Worksharing illustrated

Pseudo-Code  
Here: 4 Threads



## The Barrier Construct

- OpenMP barrier (implicit or explicit)
  - Threads wait until all threads of the current *Team* have reached the barrier

C/C++

```
#pragma omp barrier
```

- All worksharing constructs contain an implicit barrier at the end

## The Single Construct

C/C++

```
#pragma omp single [clause]
... structured block ...
```

Fortran

```
!$omp single [clause]
... structured block ...
!$omp end single
```

- The **single** construct specifies that the enclosed structured block is executed by only one thread of the team.
  - It is up to the runtime which thread that is.
- Useful for:
  - I/O
  - Memory allocation and deallocation, etc. (in general: setup work)
  - Implementation of the single-creator parallel-executor pattern as we will see later...

## The Master Construct is going to be removed with OpenMP 6.0 (2025)

C/C++

```
#pragma omp master[clause]
... structured block ...
```

Fortran

```
!$omp master[clause]
... structured block ...
!$omp end master
```

- The `master` construct specified that the enclosed structured block is executed only by the master thread of a team.
  - Note: The master construct was no worksharing construct and does not contain an implicit barrier at the end.
- Replacement: see the masked construct later on.

# Vector Addition

## Influencing the For Loop Scheduling / 1

- *for*-construct: OpenMP allows to influence how the iterations are scheduled among the threads of the team, via the *schedule* clause:
  - `schedule(static [, chunk])`: Iteration space divided into blocks of chunk size, blocks are assigned to threads in a round-robin fashion. If chunk is not specified: #threads blocks.
  - `schedule(dynamic [, chunk])`: Iteration space divided into blocks of chunk (not specified: 1) size, blocks are scheduled to threads in the order in which threads finish previous blocks.
  - `schedule(guided [, chunk])`: Similar to dynamic, but block size starts with implementation-defined value, then is decreased exponentially down to chunk.
- Default is `schedule(static)`.

## ■ Static Schedule

→ `schedule(static [, chunk])`

→ Decomposition

depending on chunksize

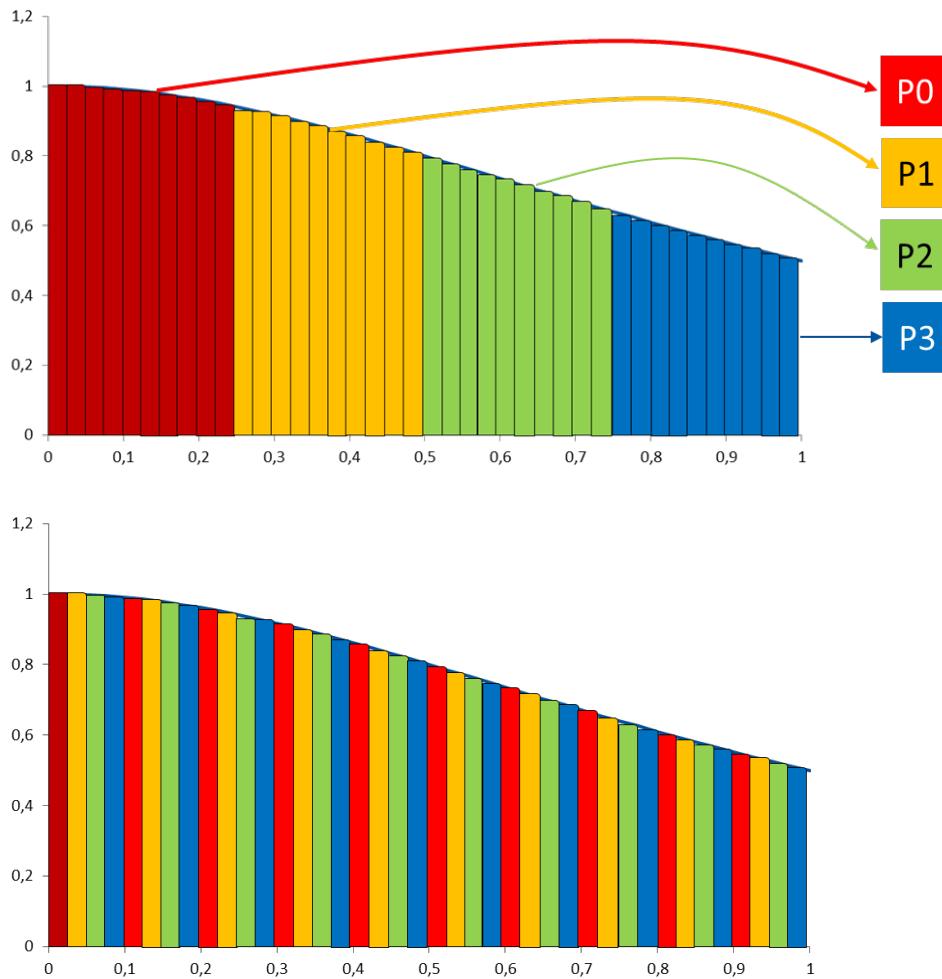
→ Equal parts of size ‘chunksize’  
distributed in round-robin  
fashion

## ■ Pros?

→ No/low runtime overhead

## ■ Cons?

→ No dynamic workload balancing



## Influencing the For Loop Scheduling / 3

- Dynamic schedule
  - `schedule(dynamic [, chunk])`
  - Iteration space divided into blocks of chunk size
  - Threads request a new block after finishing the previous one
  - Default chunk size is 1
- Pros ?
  - Workload distribution
- Cons?
  - Runtime Overhead
  - Chunk size essential for performance
  - No NUMA optimizations possible

## Synchronization Overview

- Can all loops be parallelized with `for`-constructs? No!
  - Simple test: If the results differ when the code is executed backwards, the loop iterations are not independent.  
BUT: This test alone is not sufficient:

C/C++

```
int i, int s = 0;  
  
#pragma omp parallel for  
for (i = 0; i < 100; i++)  
{  
    s = s + a[i];  
}
```

- *Data Race*: If between two synchronization points at least one thread writes to a memory location from which at least one other thread reads, the result is not deterministic (race condition).

## Synchronization: Critical Region

- A *Critical Region* is executed by all threads, but by only one thread simultaneously (*Mutual Exclusion*).

C/C++

```
#pragma omp critical (name)
{
    ... structured block ...
}
```

- Do you think this solution scales well?

C/C++

```
int i, s = 0;
#pragma omp parallel for
for (i = 0; i < 100; i++)
{
    #pragma omp critical
    {
        s = s + a[i];
    }
}
```

# Programming OpenMP

## *Scoping*

**Christian Terboven**

Michael Klemm

RWTH AACHEN  
UNIVERSITY  
OpenMP®

## Scoping Rules

- Managing the Data Environment is the challenge of OpenMP.
- *Scoping* in OpenMP: Dividing variables in *shared* and *private*:
  - *private*-list and *shared*-list on Parallel Region
  - *private*-list and *shared*-list on Worksharing constructs
  - General default is *shared* for Parallel Region, *firstprivate* for Tasks.  
Tasks are introduced later
  - Loop control variables on *for*-constructs are *private*
  - Non-static variables local to Parallel Regions are *private*
  - *private*: A new uninitialized instance is created for the task or each thread executing the construct
    - *firstprivate*: Initialization with the value before encountering the construct
    - *lastprivate*: Value of last loop iteration is written back to Master
  - Static variables are *shared*

## Privatization of Global/Static Variables

- Global / static variables can be privatized with the *threadprivate* directive
  - One instance is created for each thread
    - Before the first parallel region is encountered
    - Instance exists until the program ends
    - Does not work (well) with nested Parallel Region
  - Based on thread-local storage (TLS)
    - TlsAlloc (Win32-Threads), pthread\_key\_create (Posix-Threads), keyword `__thread` (GNU extension)

C/C++

```
static int i;  
#pragma omp threadprivate(i)
```

Fortran

```
SAVE INTEGER :: i  
!$omp threadprivate(i)
```

## Privatization of Global/Static Variables

- Global / static variables can be privatized with the *threadprivate* directive
  - One instance is created for each thread
    - Before the first parallel region is encountered
    - Instance exists until the program ends
    - Does not work (well) with nested Parallel Region
  - Based on thread-local storage (TLS)
    - TlsAlloc (Win32-Threads), pthread\_key\_create (Posix Threads), keyword \_\_thread (GNU extension)

C/C++

```
static int i;  
#pragma omp threadprivate(i)
```

Fortran

```
SAVE INTEGER :: i  
!$omp threadprivate(i)
```

Really: try to avoid the use of *threadprivate* and static variables!

# Back to our example

C/C++

```
int i, s = 0;  
#pragma omp parallel for  
for (i = 0; i < 100; i++)  
{  
    #pragma omp critical  
        { s = s + a[i]; }  
}
```

# It's your turn: Make It Scale!

```
#pragma omp parallel
```

```
{
```

```
#pragma omp for
```

```
for (i = 0; i < 99; i++)  
{
```

```
    s = s + a[i];
```

```
}
```

```
} // end parallel
```

```
do i = 0, 24  
    s = s + a(i)  
end do
```

```
do i = 0, 99  
    s = s + a(i)  
end do
```



```
do i = 25, 49  
    s = s + a(i)  
end do
```

```
do i = 50, 74  
    s = s + a(i)  
end do
```

```
do i = 75, 99  
    s = s + a(i)  
end do
```

(done)

```
#pragma omp parallel
{
    double ps = 0.0;      // private variable
#pragma omp for
for (i = 0; i < 99; i++)
{
    ps = ps + a[i];
}
#pragma omp critical
{
    s += ps;
}
} // end parallel
```

```
do i = 0, 99
s = s + a(i)
end do
```

```
do i = 0, 24
s1 = s1 + a(i)
end do
s = s + s1
```

```
do i = 25, 49
s2 = s2 + a(i)
end do
s = s + s2
```

```
do i = 50, 74
s3 = s3 + a(i)
end do
s = s + s3
```

```
do i = 75, 99
s4 = s4 + a(i)
end do
s = s + s4
```

## The Reduction Clause

- In a *reduction*-operation the operator is applied to all variables in the list. The variables have to be *shared*.
  - reduction(operator:list)
  - The result is provided in the associated reduction variable

C/C++

```
int i, s = 0;  
  
#pragma omp parallel for reduction(+:s)  
for(i = 0; i < 99; i++)  
{  
    s = s + a[i];  
}
```

- Possible reduction operators with initialization value:
  - + (0), \* (1), - (0), & (~0), | (0), && (1), || (0), ^ (0), min (largest number), max (least number)
- Remark: OpenMP also supports user-defined reductions (not covered here)

PI

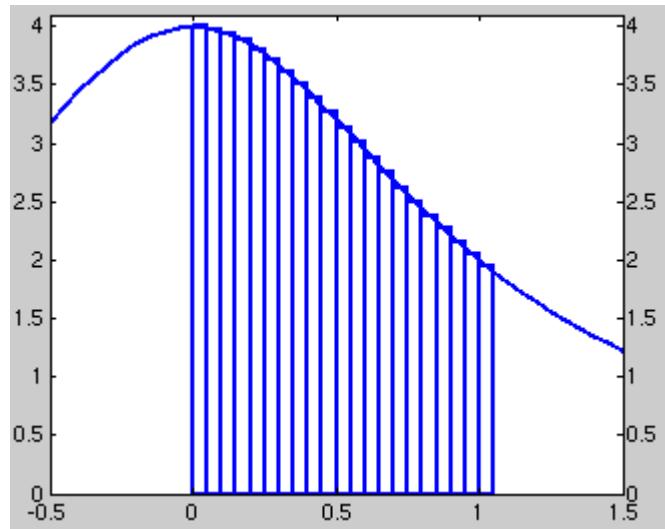
## Example: Pi (1/2)

```
double f(double x)
{
    return (4.0 / (1.0 + x*x));
}

double CalcPi (int n)
{
    const double fH = 1.0 / (double) n;
    double fSum = 0.0;
    double fX;
    int i;

#pragma omp parallel for
    for (i = 0; i < n; i++)
    {
        fX = fH * ((double)i + 0.5);
        fSum += f(fX);
    }
    return fH * fSum;
}
```

$$\pi = \int_0^1 \frac{4}{1+x^2}$$



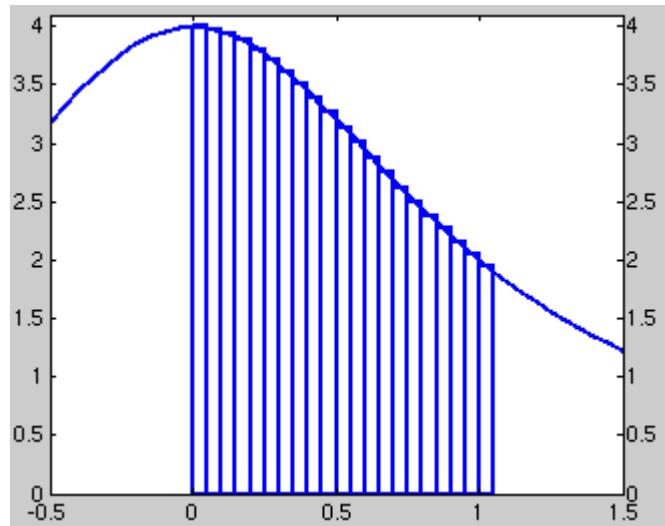
## Example: Pi (2/2)

```
double f(double x)
{
    return (4.0 / (1.0 + x*x));
}

double CalcPi (int n)
{
    const double fH = 1.0 / (double) n;
    double fSum = 0.0;
    double fX;
    int i;

#pragma omp parallel for private(fX,i) reduction(+:fSum)
    for (i = 0; i < n; i++)
    {
        fX = fH * ((double)i + 0.5);
        fSum += f(fX);
    }
    return fH * fSum;
}
```

$$\pi = \int_0^1 \frac{4}{1+x^2}$$



PI

# Programming OpenMP

## *Tasking*

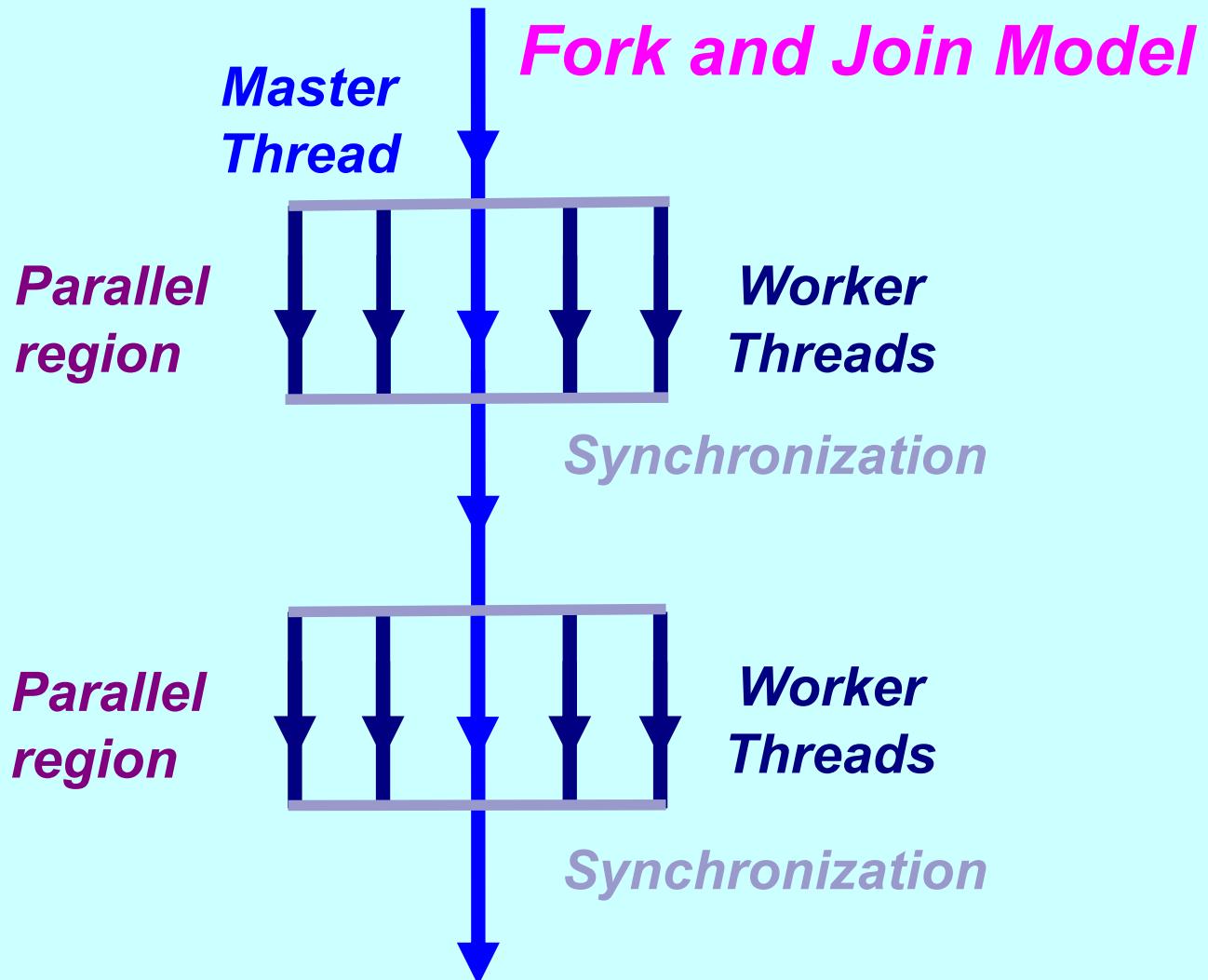
**Christian Terboven**

Michael Klemm

**RWTHAACHEN  
UNIVERSITY**  
**OpenMP**<sup>®</sup>

# OpenMP Review

# The OpenMP Execution Model



```
#pragma omp parallel  
{  
    ....  
}
```

```
#pragma omp parallel  
{  
    ....  
}
```

# The Worksharing Constructs

- ***The work is distributed over the threads***
- ***Must be enclosed in a parallel region***
- ***Must be encountered by all threads in the team, or none at all***
- ***No implied barrier on entry***
- ***Implied barrier on exit (unless the nowait clause is specified)***
- ***A work-sharing construct does not launch any new threads***

```
#pragma omp for  
{  
    ....  
}
```

```
#pragma omp sections  
{  
    ....  
}
```

```
#pragma omp single  
{  
    ....  
}
```

# The Single and Master Directives

- Single: only one thread in the team executes the code enclosed

```
#pragma omp single [private] [firstprivate] \
    [copyprivate] [nowait]
{
    <code-block>
}
```

- Master: the master thread executes the code enclosed

```
#pragma omp master
{<code-block>}
```

*There is no implied  
barrier on entry or  
exit !*



# The OpenMP Barrier

- Several constructs have an implied barrier
  - This is another safety net (has implied flush by the way)
  - the “nowait” clause
- This can help fine tuning the application
  - But you'd better know what you're doing
- The explicit barrier comes in quite handy then

```
#pragma omp barrier
```



# *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

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

- (2) Try all numbers:

- (2 a) Check Sudoku
- If invalid: skip
- If valid: Go to next field

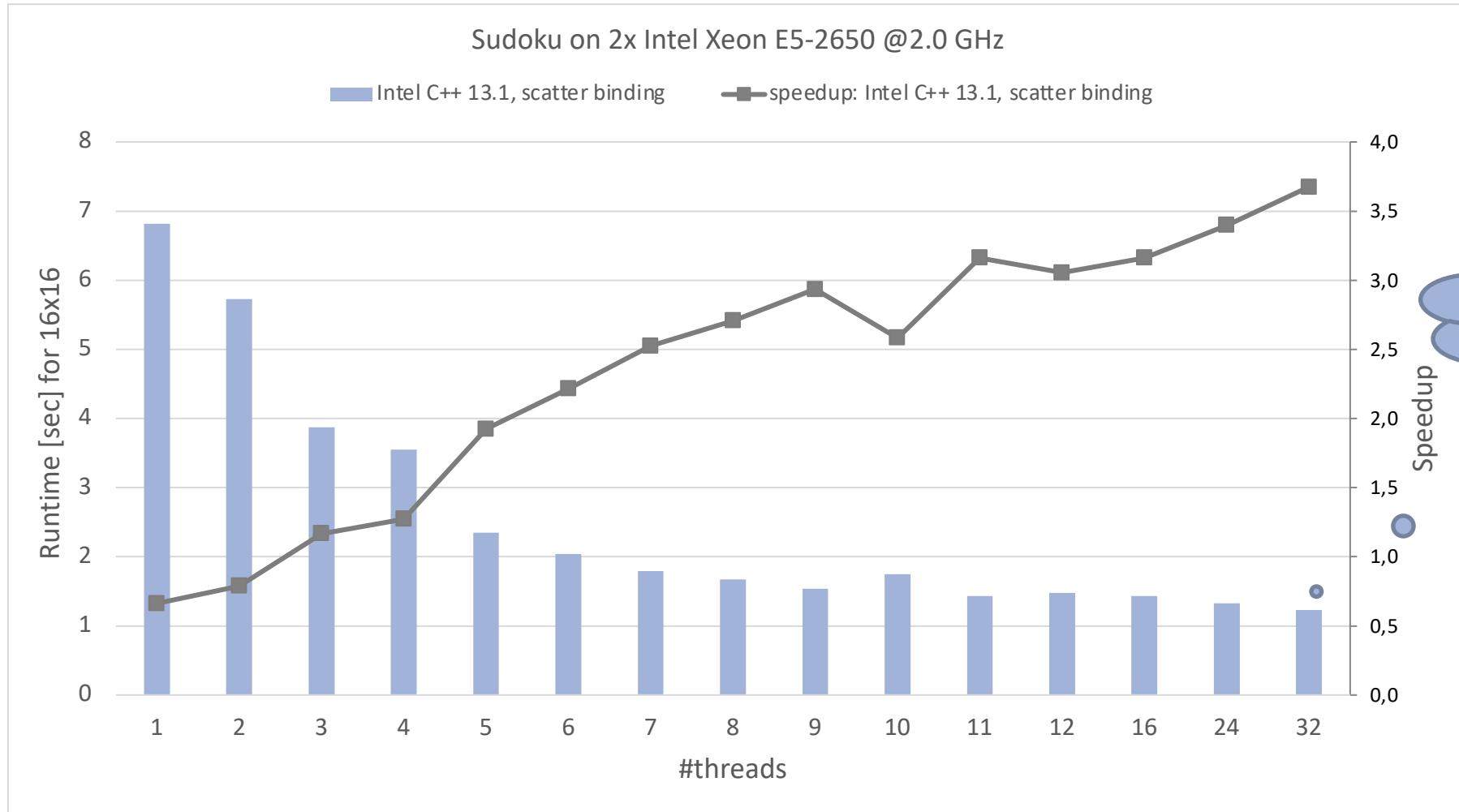
`#pragma omp task`  
needs to work on a new copy  
of the Sudoku board

- Wait for completion

`#pragma omp taskwait`  
wait for all child tasks



# Performance Evaluation



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> ); ...
}
```

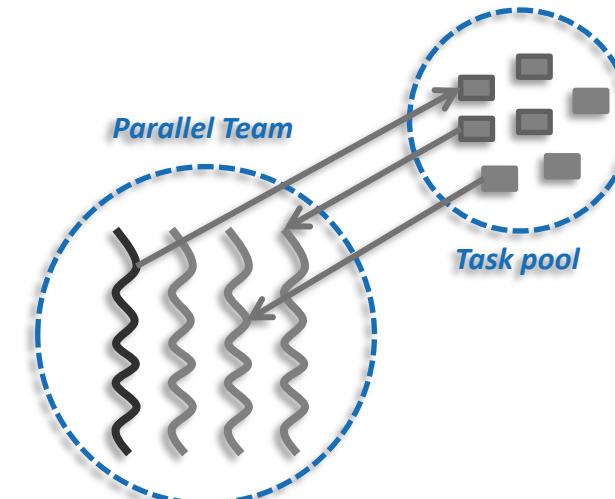
- Several scenarios are possible:

→ 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 master
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)

## Data Environment

- allocate([allocator:] list)
- detach(event-handler)

## Miscellaneous

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

## Cutoff Strategies

- depend(dep-type: list)
- untied
- priority(priority-value)
- affinity(list)

## 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

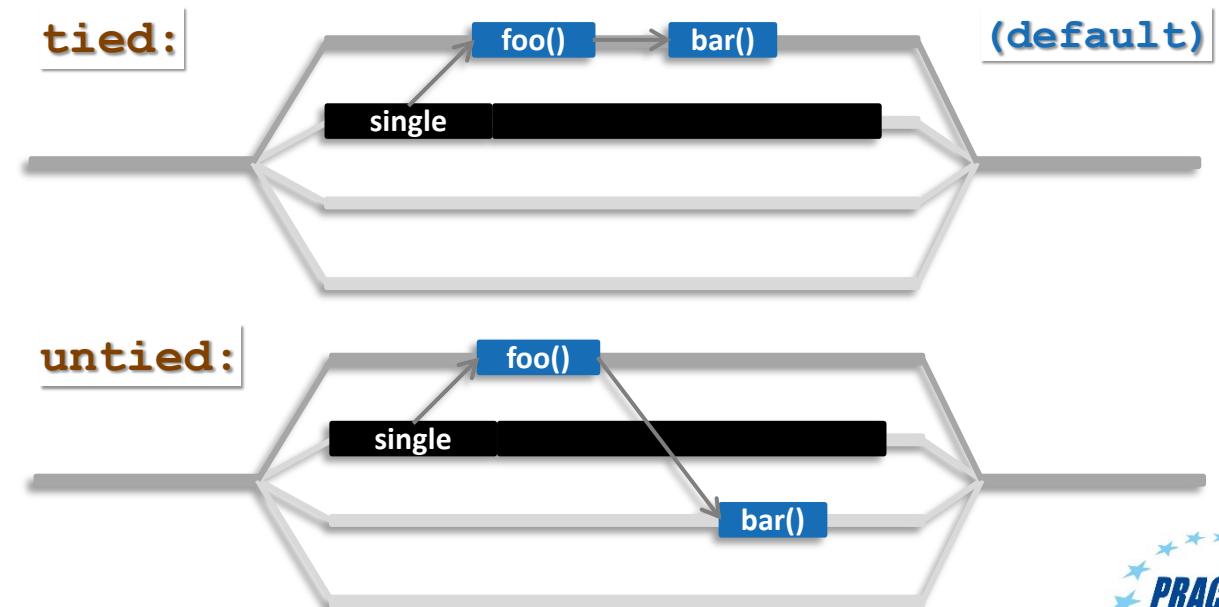
## ■ Task scheduling points (and the taskyield directive)

- 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 untied
    {
        foo();
        #pragma omp taskyield
        bar()
    }
}
```



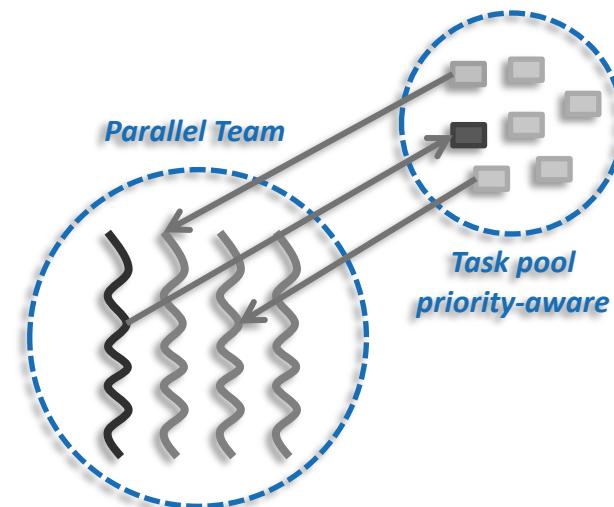
# Task scheduling: programmer's hints

- Programmers may specify a priority value when creating a task

```
#pragma omp task priority(pvalue)
{structured-block}
```

- pvalue: the higher → the best (will be scheduled earlier)
- once a thread becomes idle, gets one of the highest priority tasks

```
#pragma omp parallel
#pragma omp single
{
    for ( i = 0; i < SIZE; i++) {
        #pragma omp task priority(1)
        { code_A; }
    }
    #pragma omp task priority(100)
    { code_B; }
    ...
}
```



# 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)

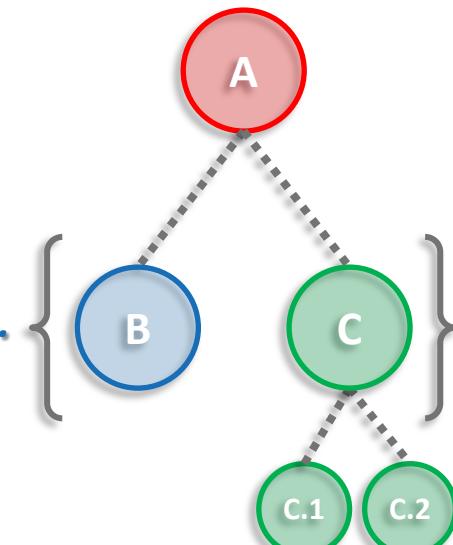
```
#pragma omp parallel
#pragma omp single
{
    #pragma omp task
    {
        #pragma omp task :B
        {
            #pragma omp task :C
            {
                ...
                #pragma omp task :C.x
            }
        }
        #pragma omp taskwait
    }
} // implicit barrier will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

:A

:B

:C

*wait for...*



# 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)

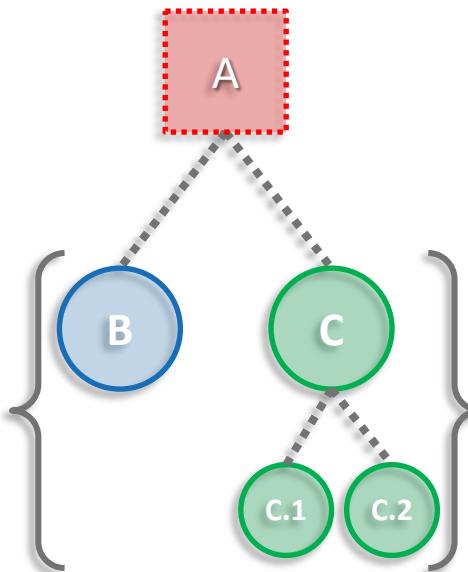
```
#pragma omp parallel
#pragma omp single
{
    #pragma omp taskgroup
    {
        #pragma omp task :A
        {
            #pragma omp task :B
            {
                ...
            }
            #pragma omp task :C
            {
                ...
                #C.1; #C.2; ...
      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

: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
}
```

## ■ 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
}
```

1

```
int *p;
p = malloc(sizeof(float)*SIZE);

#pragma omp task
{
  // *p: shared
}
```

2

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

5

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

4

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

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

3

# 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:
            // Scope of c:
            // Scope of d:
            // Scope of e:
        }
    }
}
```



# 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}
```

- Register a new reduction at [1]
- Computes the final result after [3]

## ■ The (task) in\_reduction clause [participating]

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

- Task participates in a reduction operation [2]

```
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

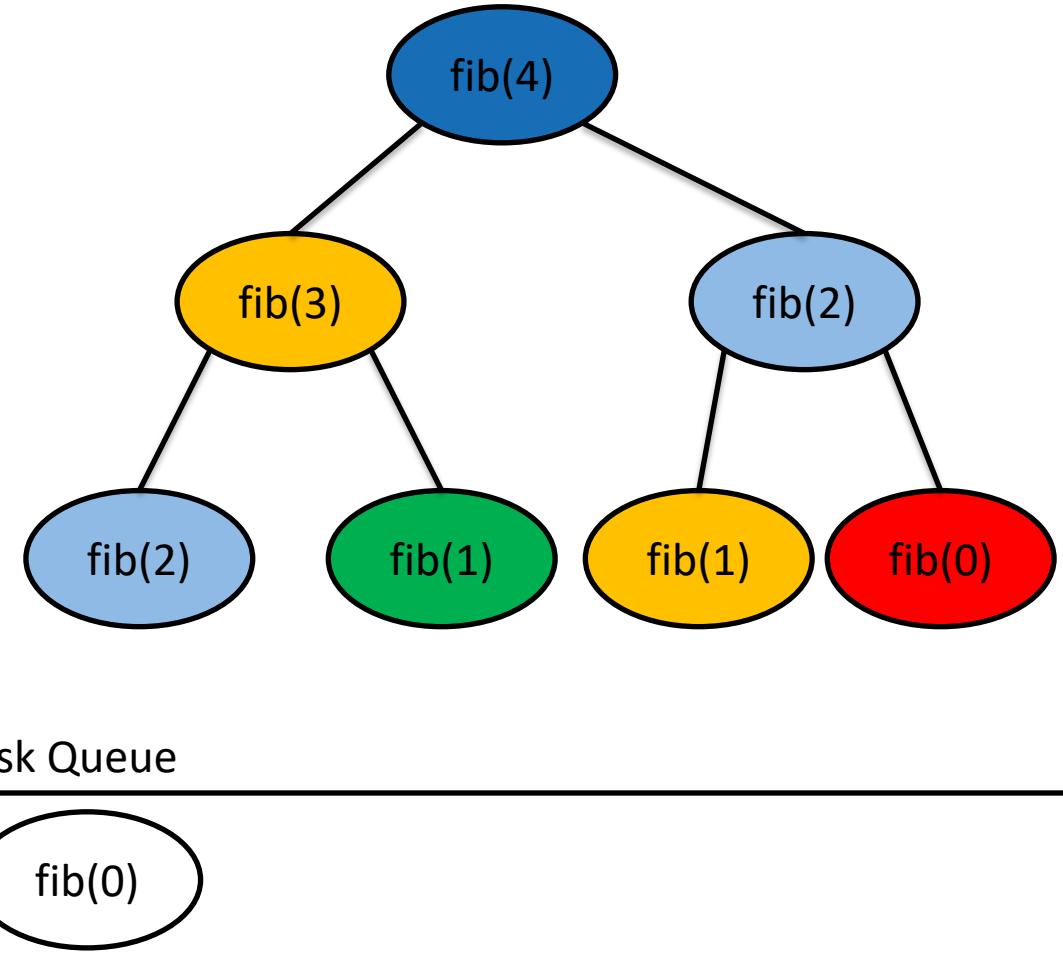
```
1 int main(int argc,
2          char* argv[])
3 {
4     [...]
5     #pragma omp parallel
6     {
7         #pragma omp single
8         {
9             fib(input);
10        }
11    }
12    [...]
13 }
```

```
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 }
```

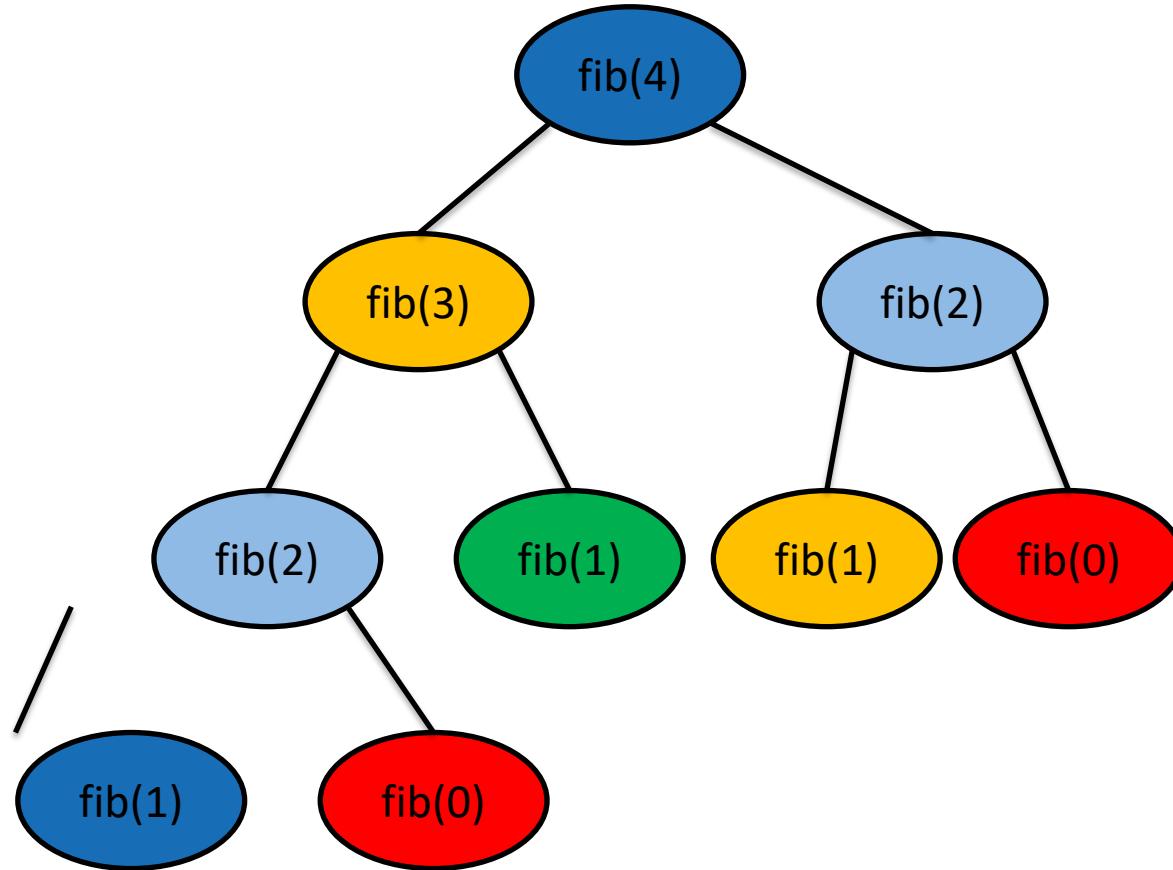
- Only one Task / Thread enters fib() from main(), it is responsible for creating the two initial work tasks
- Taskwait is required, as otherwise x and y would get lost



- T1 enters fib(4)
- 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



# Traditional Worksharing

- Worksharing constructs do not compose well
- Pathological example: parallel dgemm in MKL

```
void example() {
    #pragma omp parallel
    {
        compute_in_parallel(A);
        compute_in_parallel_too(B);
        // dgemm is either parallel or sequential,
        // but has no orphaned worksharing
        cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                    m, n, k, alpha, A, k, B, n, beta, C, n);

    }
}
```

- Writing such code either
  - oversubscribes the system,
  - yields bad performance due to OpenMP overheads, or
  - needs a lot of glue code to use sequential dgemm only for sub-matrixes



# Example: Sparse CG

```

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

```

```

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

```



# 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;
}
```

- Hiding the internal details
- 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)

## Data Environment

- grainsize(grain-size)
- num\_tasks(num-tasks)

## Chunks/Grain

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

## Cutoff Strategies

- untied
- priority(priority-value)

## Scheduler (R/H)

- collapse(n)
- nogroup
- allocate([allocator:] list)

## Miscellaneous

# Worksharing vs. taskloop constructs (1/2)

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

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

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

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

Result: x = 1024

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

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

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

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

Result: x = 16384

# Worksharing vs. taskloop constructs (2/2)

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

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

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

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

Result: x = 1024

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

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

Result: x = 1024

# Taskloop decomposition approaches

## ■ Clause: grainsize(grain-size)

- Chunks have at least grain-size iterations
- Chunks have maximum 2x grain-size iterations

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

## ■ Clause: num\_tasks(num-tasks)

- Create num-tasks chunks
- Each chunk must have at least one iteration

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

## ■ If none of previous clauses is present, the *number of chunks* and the *number of iterations per chunk* is implementation defined

## ■ Additional considerations:

- The order of the creation of the loop tasks is unspecified
- Taskloop creates an implicit taskgroup region; **nogroup** → no implicit taskgroup region is created



# 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;
}
```

## 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 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}
```

- Fortran syntax:

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

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



# Example: Sparse CG with taskloop

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

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



# Improving Tasking Performance: Task dependences



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

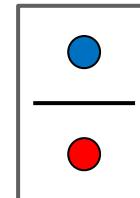
```
int x = 0;  
#pragma omp parallel  
#pragma omp single  
{  
    #pragma omp task  
    std::cout << x << std::endl;  
  
    #pragma omp taskwait  
  
    #pragma omp task  
    x++;  
}
```

OpenMP 3.1

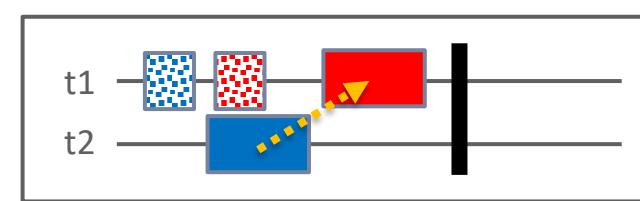
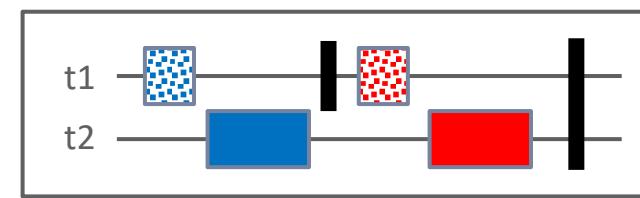
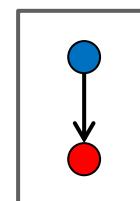
```
int x = 0;  
#pragma omp parallel  
#pragma omp single  
{  
    #pragma omp task depend(in: x)  
    std::cout << x << std::endl;
```

OpenMP 4.0

OpenMP 3.1



OpenMP 4.0



Task's creation time  
 Task's execution time



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

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

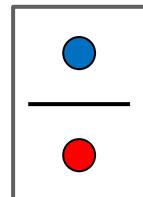
OpenMP 3.1

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

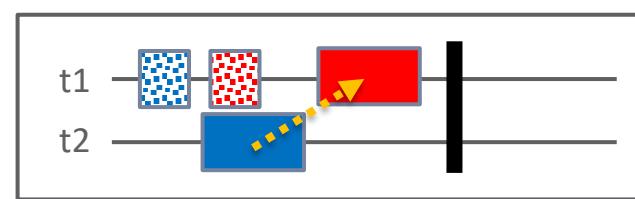
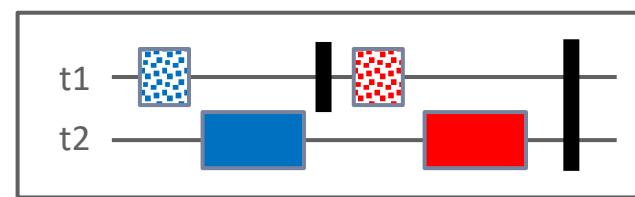
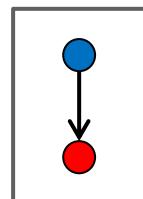
OpenMP 4.0

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

OpenMP 3.1



OpenMP 4.0



Task's creation time  
 Task's execution time

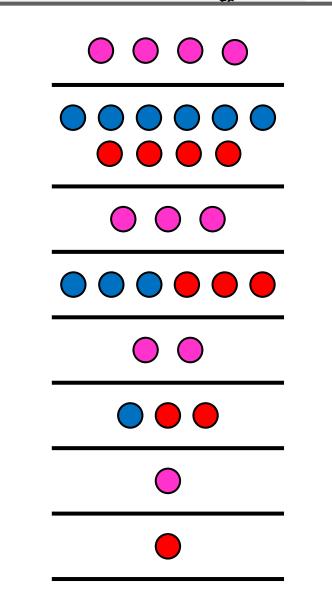


# Motivation: Cholesky factorization

```
void cholesky(int ts, int nt, double* a[nt][nt]) {
    for (int k = 0; k < nt; k++) {
        // Diagonal Block factorization
        potrf(a[k][k], ts, ts);

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

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

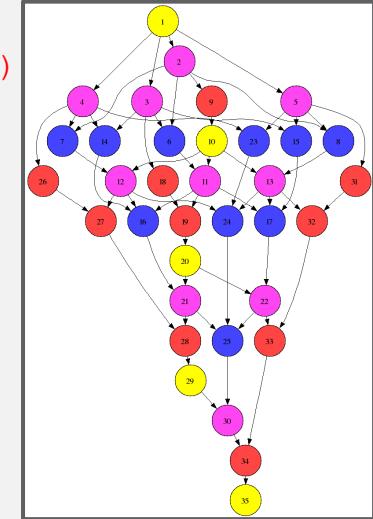


OpenMP 3.1

```
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

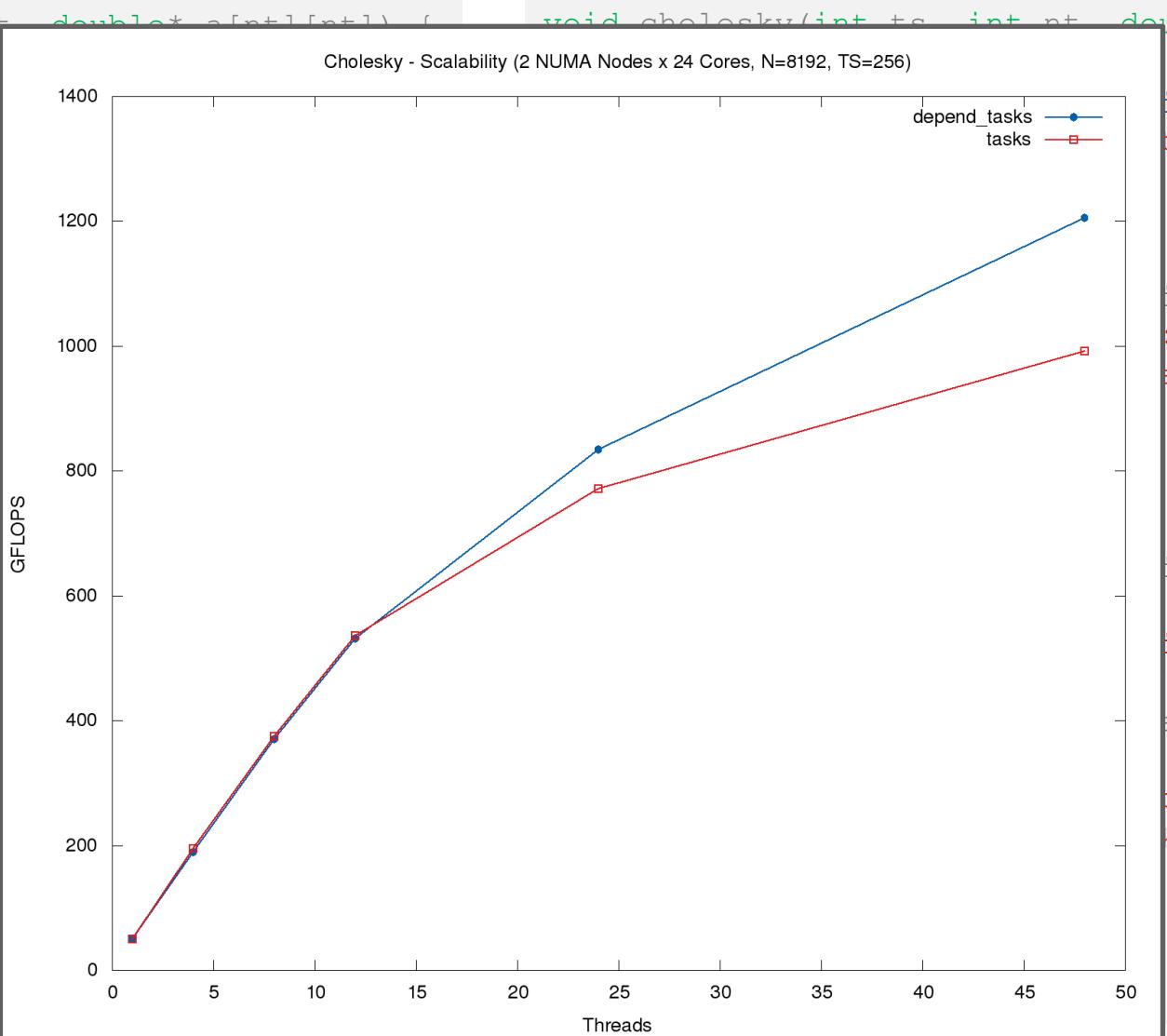


# Motivation: Cholesky factorization

```
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);

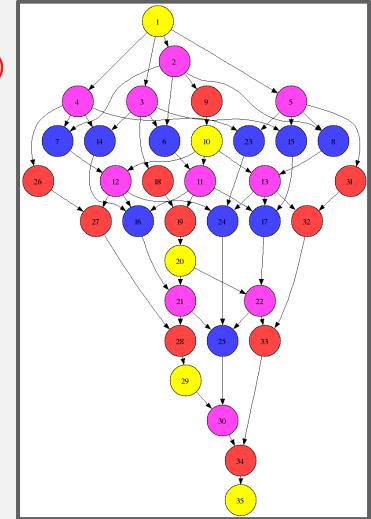
        // Triangular systems
        for (int i = k + 1; i < nt; i++) {
            #pragma omp task
            trsm(a[k][k], a[k][i]);
        }
        #pragma omp taskwait

        // 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, ts);
        }
        #pragma omp taskwait
    }
}
```



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

    out: a[i][i])
    <][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**

## 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
- 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

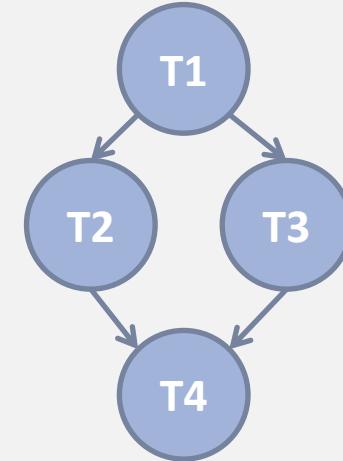
# What's in the spec: depend clause (1)

- 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

none of the list items in

none of the list items in

# 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);

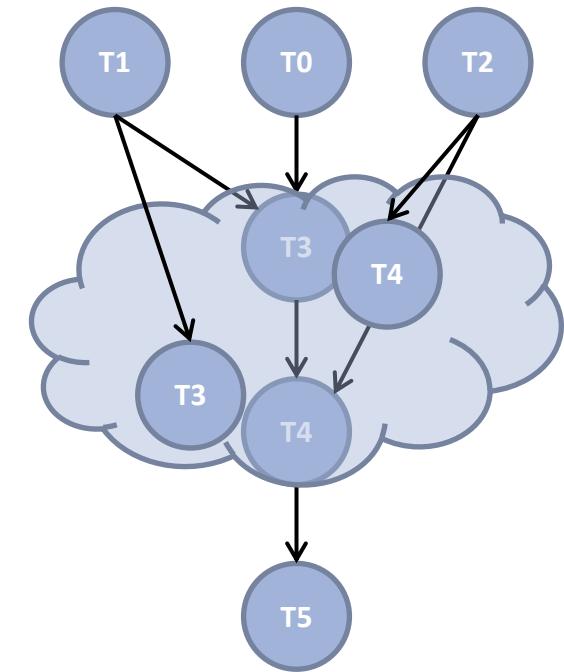
    #pragma omp task depend(out: y)      //T2
    short_computation(y);

    #pragma omp task depend(in: x) depend(mutexinoutset:T3res) //T3
    res += x;

    #pragma omp task depend(in: y) depend(mutexinoutset:T4res) //T4
    res += y;

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

```



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

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

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

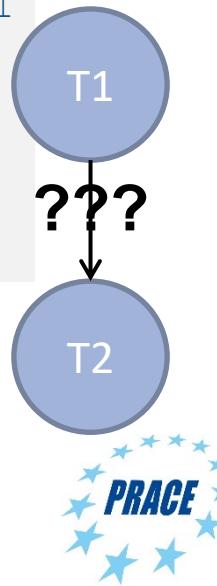
- List items used in the depend clauses [...] must indicate **identical** or **disjoint storage**

```
//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)

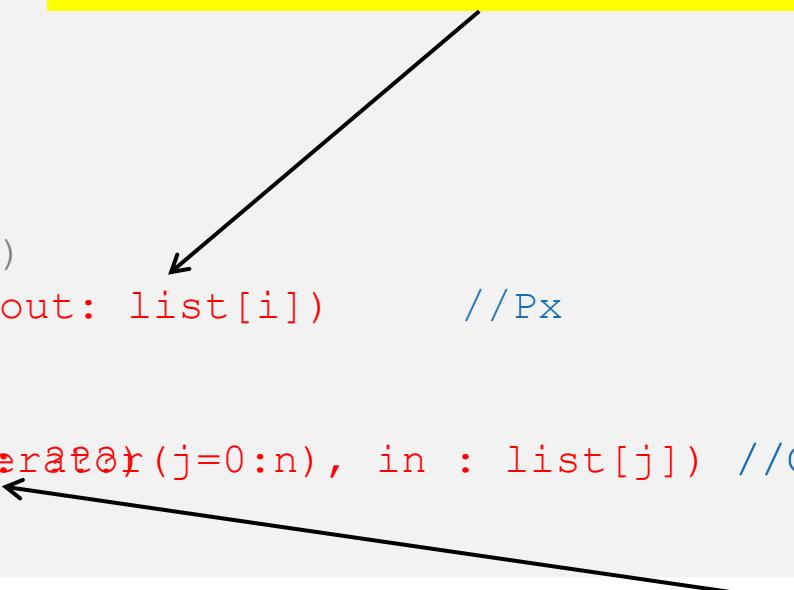
## ■ Iterators + deps: a way to define a dynamic number of dependences

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

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



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

# *Use case*

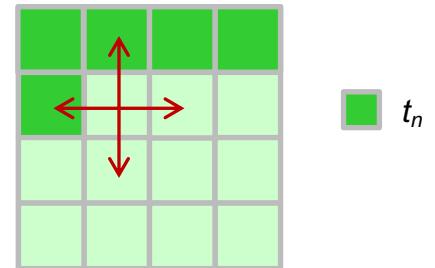


# Use case: intro to Gauss-seidel

```
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
            }
        }
    }
}
```

## 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

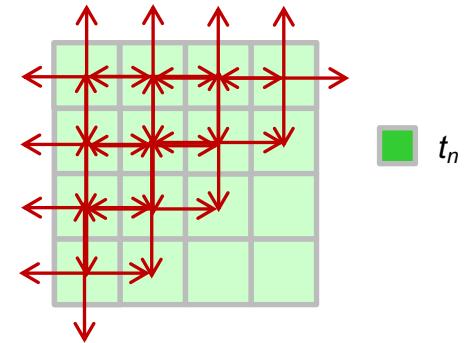


# Use case: Gauss-seidel (2)

```
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  $t$



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
        }
    }
}
```

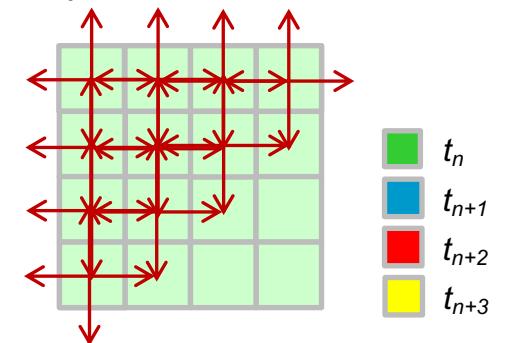


# Use case : Gauss-seidel (4)

```
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
            }
        }
    }
}
```

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

*multiple time iterations*



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



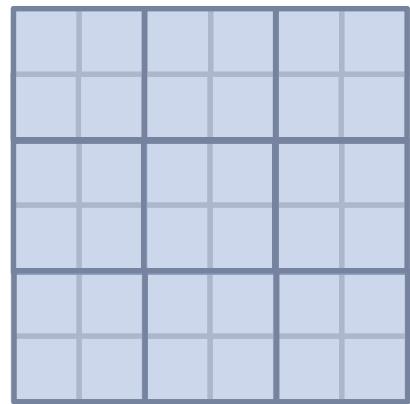
# Use case : Gauss-seidel (5)

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

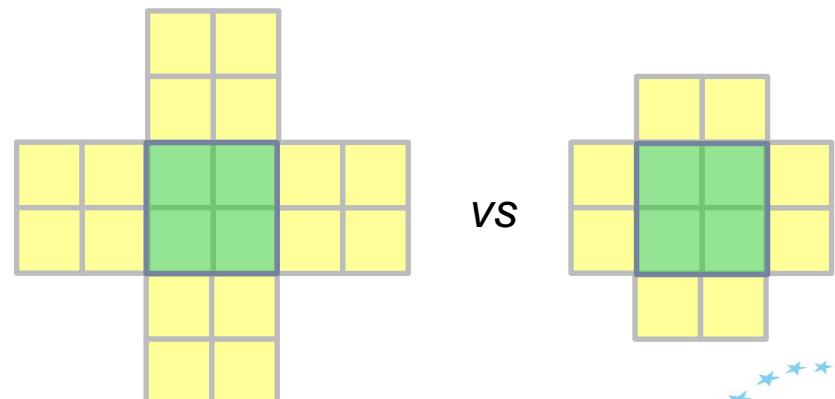
    #pragma omp parallel
    #pragma omp single
    for (int t = 0; t < tsteps; ++t)
        for (int ii=1; ii < size-1; ii+=TS)
            for (int jj=1; jj < size-1; jj+=TS) {
                #pragma omp task depend(inout: p[ii:TS][jj:TS])
                depend(in: p[ii-TS:TS][jj:TS], p[ii+TS:TS][jj:TS],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



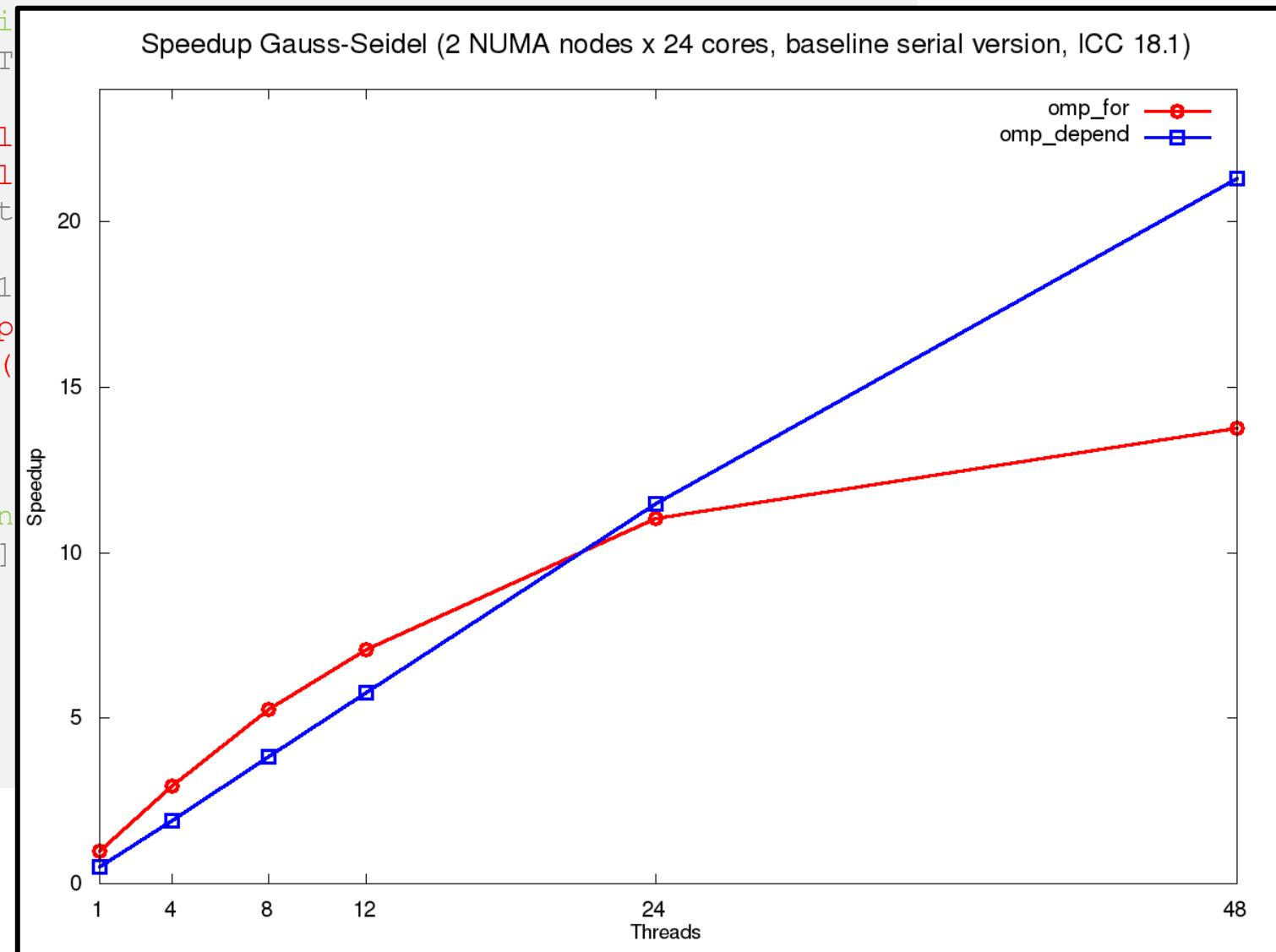
Q: Why do the input dependences depend on the whole block rather than just a column/row?



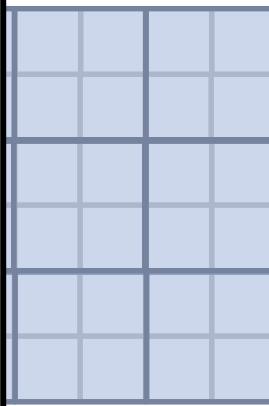
# Use case : Gauss-seidel (5)

```
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]
}
}
}
```

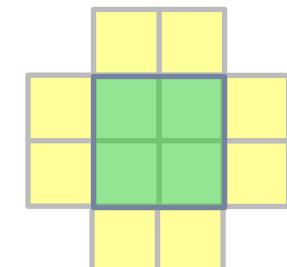


matrix region



the input dependences  
use whole block rather  
than a column/row?

vs



# 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

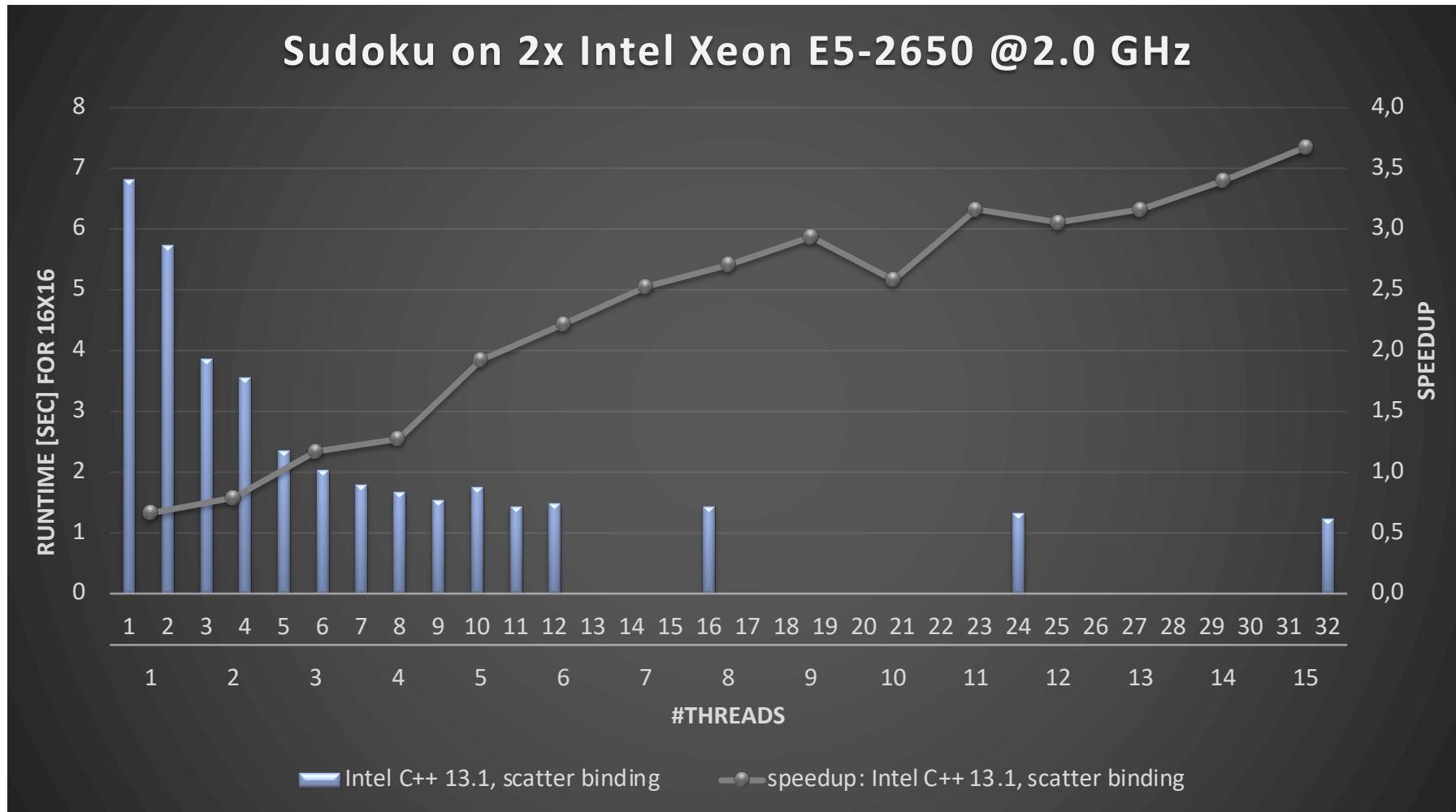
first call contained in a  
`#pragma omp parallel`  
`#pragma omp single`  
 such that one tasks starts the  
 execution of the algorithm

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

`#pragma omp taskwait`  
 wait for all child tasks

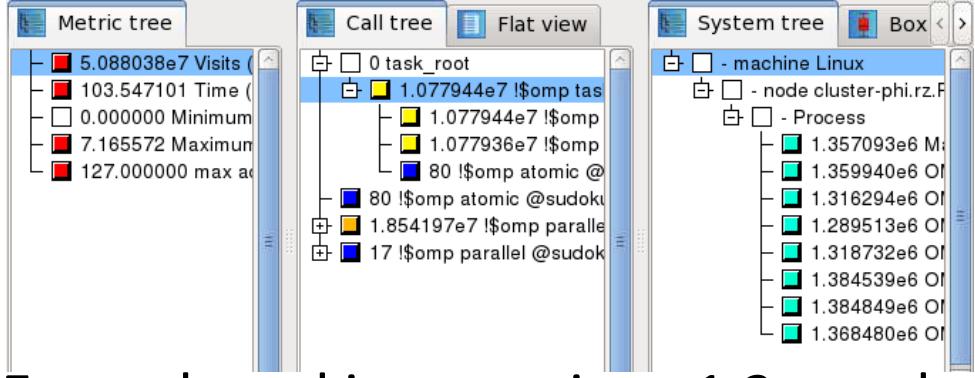


# Performance Evaluation

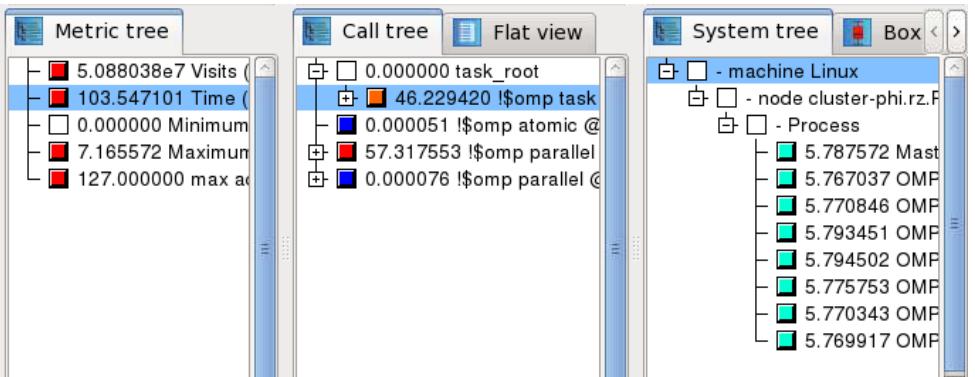


# Performance Analysis

Event-based profiling provides a good overview :



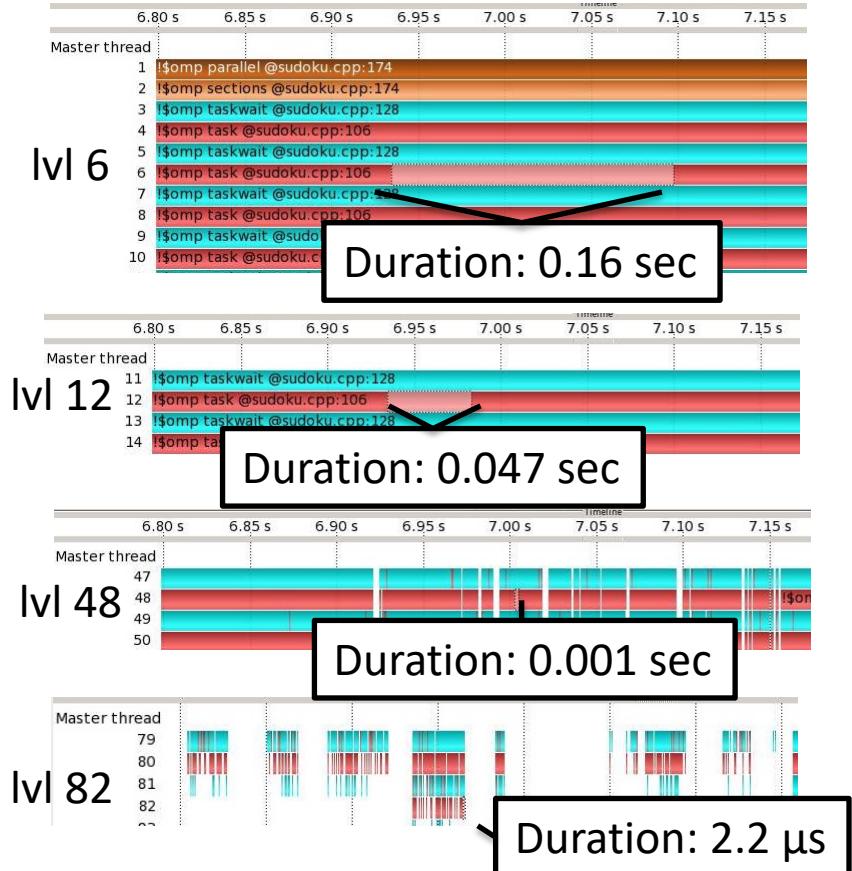
Every thread is executing ~1.3m tasks...



... in ~5.7 seconds.

=> average duration of a task is ~4.4 µs

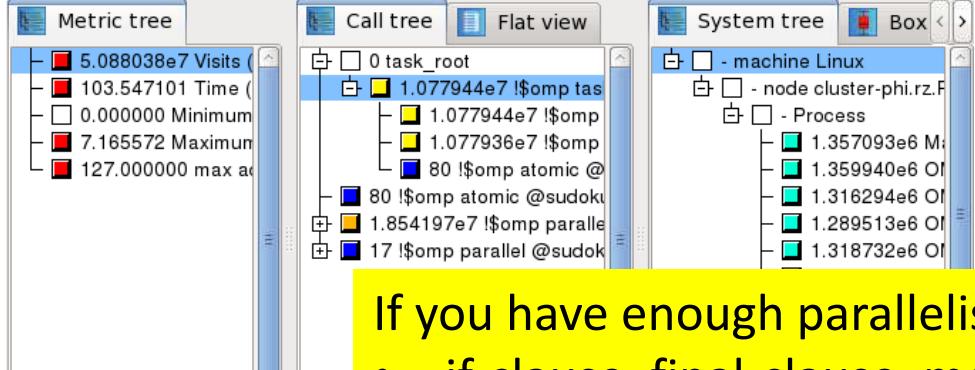
Tracing provides more details:



Tasks get much smaller down the call-stack.

# Performance Analysis

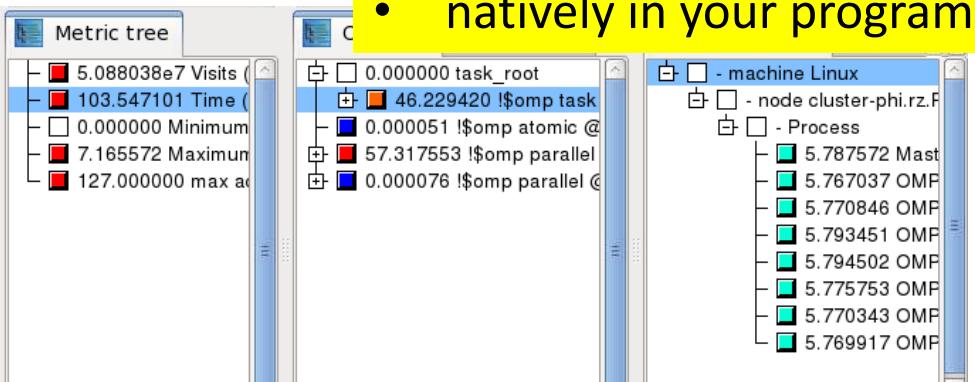
Event-based profiling provides a good overview :



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

Every thread is doing something

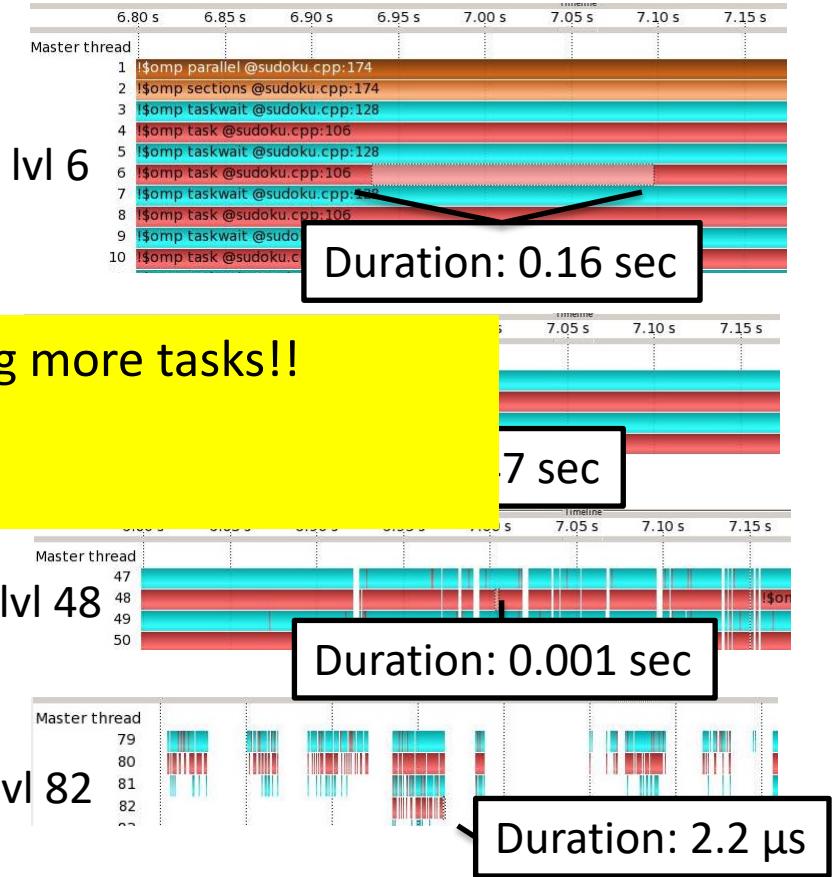
- if-clause, final-clause, mergeable-clause
- natively in your program code



... in ~5.7 seconds.

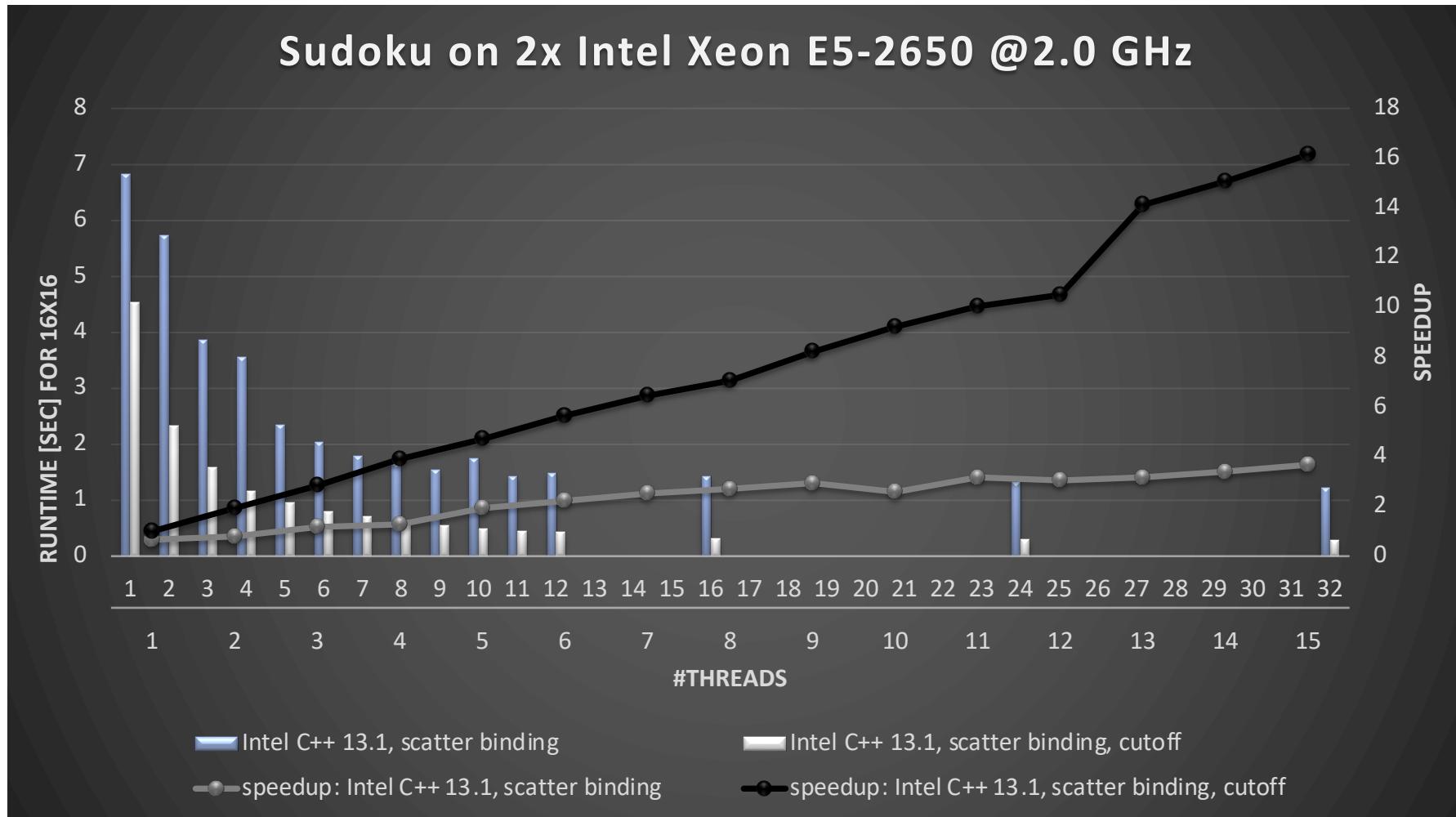
=> average duration of a task is ~4.4 µs

Tracing provides more details:



Tasks get much smaller down the call-stack.

# Performance Evaluation (with cutoff)



# The if clause

- Rule of thumb: the `if (expression)` clause as a “switch off” mechanism
  - Allows lightweight implementations of task creation and execution but it reduces the parallelism
- If the expression of the `if` clause evaluates to `false`
  - the encountering task is suspended
  - the new task is executed immediately (task dependences are respected!!)
  - the encountering task resumes its execution once the new task is completed
  - This is known as *undelayed task*
- Even if the expression is false, data-sharing clauses are honored

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

Really useful to debug tasking applications!



# The final clause

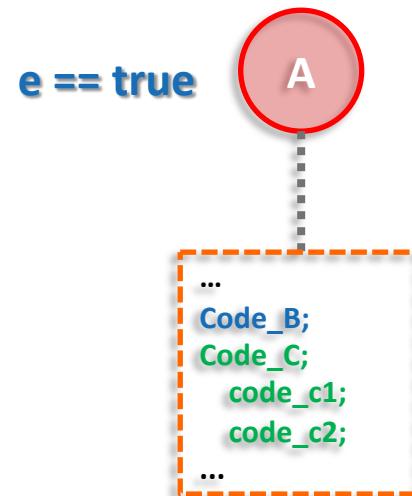
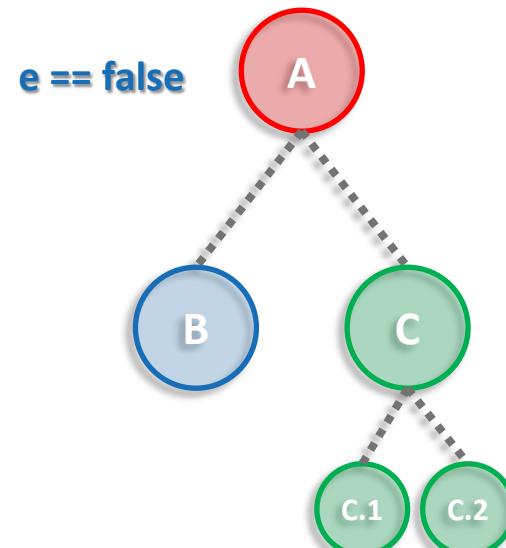
## ■ The final (expression) clause

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

## ■ If the expression of the final clause evaluates to true

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

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



## ■ Data-sharing clauses are honored too!

# The mergeable clause

## ■ The mergeable clause

- Optimization: get rid of “data-sharing clauses are honored”
- This optimization can only be applied in *undeferred* or *included* tasks

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

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

## ■ A *merged task* is:

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

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

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



# Programming OpenMP

*Using OpenMP Compilers*

Christian Terboven

Michael Klemm



# Production Compilers w/ OpenMP Support

- GCC
- clang/LLVM
- Intel Classic and Next-gen Compilers
- AOCC, AOMP, ROCmCC
- IBM XL
- ... and many more
  
- See <https://www.openmp.org/resources/openmp-compilers-tools/> for a list

# Compiling OpenMP

- Enable OpenMP via the compiler's command-line switches

- GCC: -fopenmp
- clang: -fopenmp
- Intel: -fopenmp or -qopenmp (classic) or -fiopenmp (next-gen)
- AOCC, AOCL, ROCmCC: -fopenmp
- IBM XL: -qsmp=omp

- Switches have to be passed to both compiler and linker:

```
$ gcc [...] -fopenmp -o matmul.o -c matmul.c
$ gcc [...] -fopenmp -o matmul matmul.o
$./matmul 1024
Sum of matrix (serial): 134217728.000000, wall time 0.413975, speed-up 1.00
Sum of matrix (parallel): 134217728.000000, wall time 0.092162, speed-up 4.49
```

# Programming OpenMP

## *Hands-on Exercises*

Christian Terboven

Michael Klemm



# Webinar Exercises

- We have implemented a series of small hands-on examples that you can use and play with.
  - Download: `git clone https://github.com/cterboven/OpenMP-tutorial-PRACE-2022.git`
  - Build: `make` (in the corresponding subdirectories)
  - You can then find the compiled code in the “bin” folder to run it
  - We use the GCC compiler mostly, some examples require Intel’s Math Kernel Library
- 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
- Also provided: basic exercises in the `openmp-simple-exercises.tar` archive
  - Instructions contained in the archive: `Exercises_OMP_2021.pdf`

# Programming OpenMP

## *Hands-on Exercises*

**Christian Terboven**

Michael Klemm



# *Solution of Homework Assignments*

# Example: sin-cos

```
double do_some_computation(int i) {
    double t = 0.0; int j;
    for (j = 0; j < i*i; j++) {
        t += sin((double)j) * cos((double)j);
    }
    return t;
}

int main(int argc, char* argv[]) {
    const int dimension = 500;
    int i;
    double result = 0.0;
    double t1 = omp_get_wtime();
    #pragma omp parallel for schedule(dynamic) reduction(+:result)
    for (i = 0; i < dimension; i++) {
        result += do_some_computation(i);
    }
    double t2 = omp_get_wtime();
    printf("Computation took %.3lf seconds.\n", t2 - t1);
    printf("Result is %.3lf.\n", result);
    return 0;
}
```

# Example: matmul

```
void matmul_seq(double * C, double * A, double * B, size_t n) { ... }

void matmul_par(double * C, double * A, double * B, size_t n) {
#pragma omp parallel for shared(A,B,C) firstprivate(n) \
    schedule(static) // collapse(2)
    for (size_t i = 0; i < n; ++i) {
        for (size_t k = 0; k < n; ++k) {
            for (size_t j = 0; j < n; ++j) {
                C[i * n + j] += A[i * n + k] * B[k * n + j];
            }
        }
    }
}

void init_mat(double * C, double * A, double * B, size_t n) { ... }

void dump_mat(double * mtx, size_t n) { ... }
double sum_mat(double * mtx, size_t n) { ... }

int main(int argc, char *argv[]) { ... }
```

# Example: cholesky

```
void cholesky(int ts, int nt, double* Ah[nt][nt]) {  
    for (int k = 0; k < nt; k++) {  
        LAPACKE_dpotrf(LAPACK_COL_MAJOR, 'L', ts, Ah[k][k], ts);  
  
        #pragma omp parallel for  
        for (int i = k + 1; i < nt; i++) {  
            cblas_dtrsm(CblasColMajor, CblasRight, CblasLower, CblasTrans,  
                         CblasNonUnit, ts, ts, 1.0, Ah[k][k], ts, Ah[k][i], ts);  
        }  
  
        #pragma omp parallel for  
        for (int i = k + 1; i < nt; i++) {  
            for (int j = k + 1; j < i; j++) {  
                cblas_dgemm(CblasColMajor, CblasNoTrans, CblasTrans, ts, ts, ts, -1.0,  
                            Ah[k][i], ts, Ah[k][j], ts, 1.0, Ah[j][i], ts);  
            }  
            cblas_dsyrk(CblasColMajor, CblasLower, CblasNoTrans, ts, ts, -1.0,  
                        Ah[k][i], ts, 1.0, Ah[i][i], ts);  
        }  
    }  
}
```

Blocked matrix  
w/ block size  $ts$