Mastering Tasking with OpenMP

Christian Terboven

Michael Klemm

Stephen Olivier

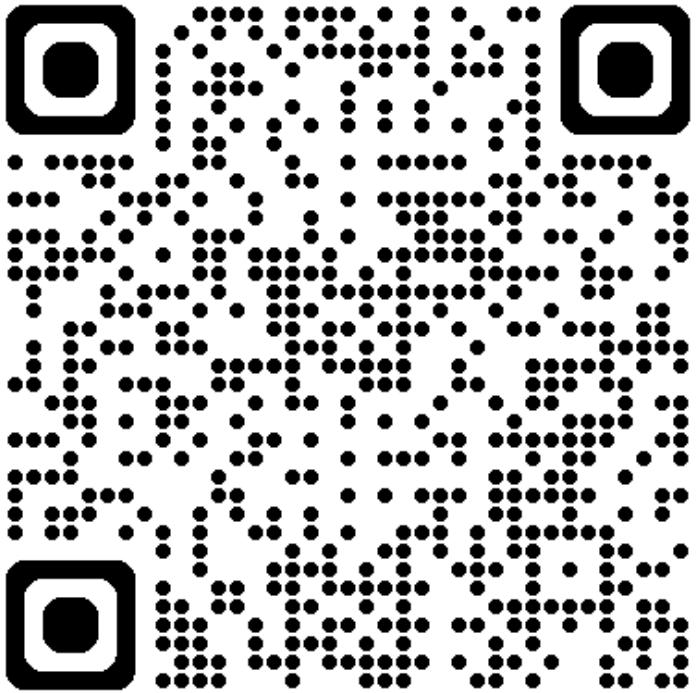
Bronis R. de Supinski

Members of the OpenMP Language Committee

Agenda

- OpenMP Overview ~20 min.
  - → Welcome, Basics, Memory Model
- OpenMP Tasking Model ~50 min.
  - → Tasking, Data Sharing, Taskloop
- Improving Tasking Performance ~50 min.
  - → Dependences, Cut-Off, Affinity
- OpenMP Free-agent Threads ~30 min.
- Task iterations ~15 min.
- Future OpenMP Directions ~15 min.
  - → OpenMP 6.0 and Beyond

Material and Evaluation



Updated slides: https://bit.ly/sc25-tsk-omp

Please participate in the tutorial‘s evaluation (link will be provided later)!

Acknowledgements

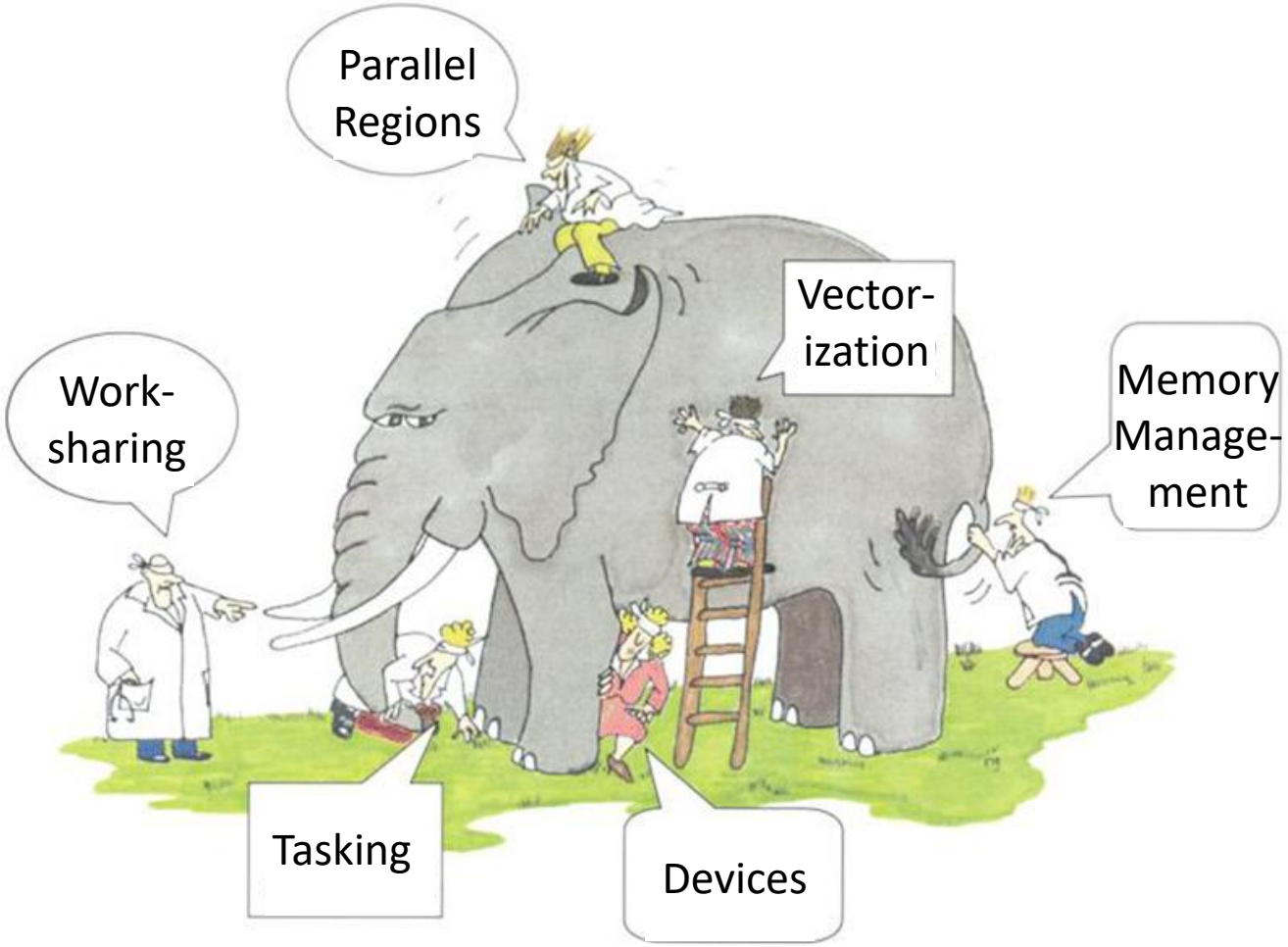
Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

OpenMP Overview

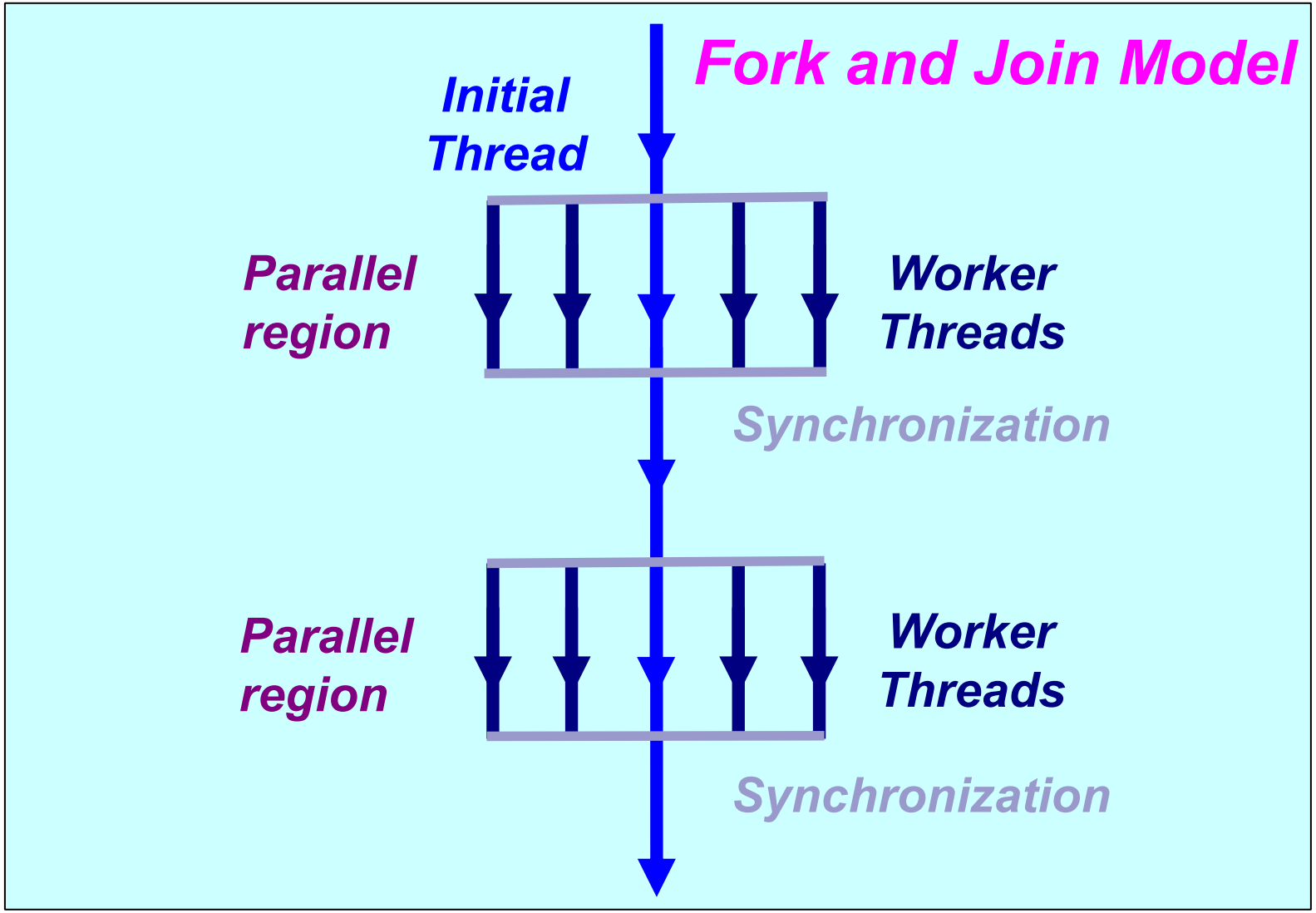
What is OpenMP?

- De-facto standard Application Programming Interface (API) to write <u>shared memory parallel</u> applications in C, C++, and Fortran
- Consists of Compiler Directives, Runtime routines and Environment variables

- Version 5.0 has been released at SC18
- Version 5.2 has been released at SC21
- Version 6.0 has been released at SC24



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
{
        ....
}
```

Single, Masked and Master / 1

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

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

There is no implied barrier on entry, but there is one on exit !

- Masked: rule-based selection of threads for region execution

```
#pragma omp masked [filter(integer-expression)]
{<code-block>}
```

Single, Masked and Master / 2

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

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

- Masked: rule-based selection of threads for region execution

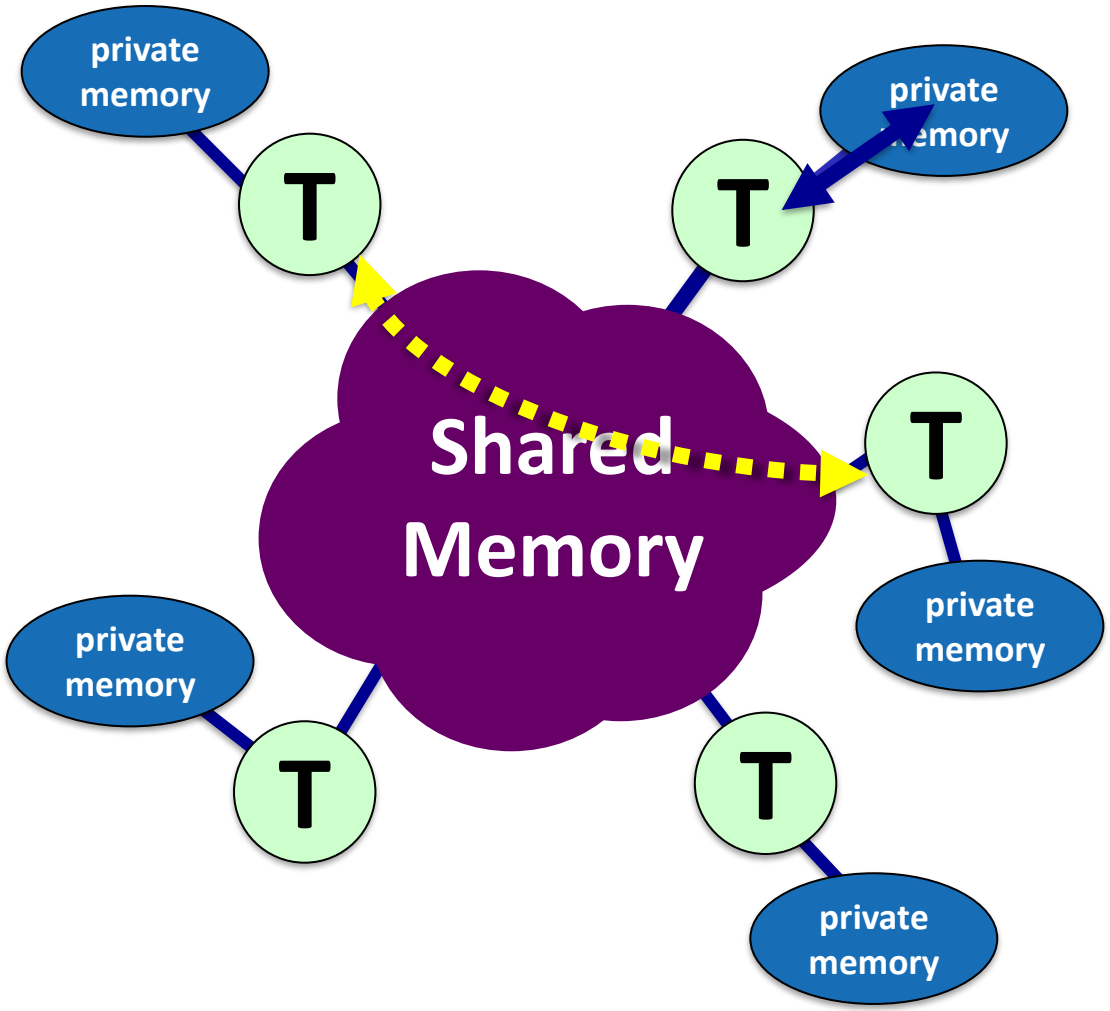
```
#pragma omp masked [filter(integer-expression)]
{<code-block>}
```

→Replacement of Master:

```
#pragma omp masked [filter(0)]
{<code-block>}
```

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*



Tasking Motivation

Sudoko 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

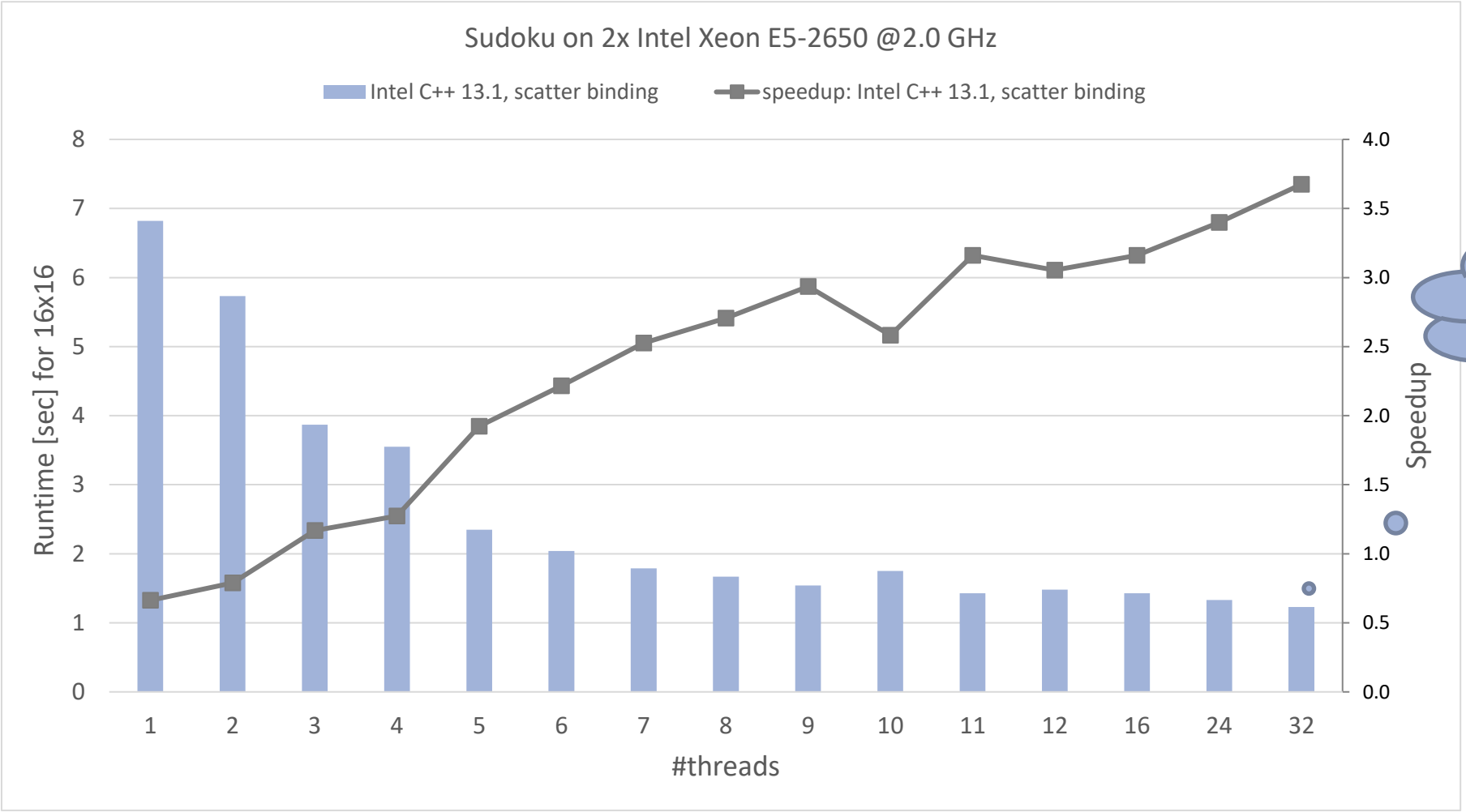
- (2) Try all numbers:
  - (2 a) Check Sudoku
    - If invalid: skip
    - If valid: Go to next field

```
#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 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

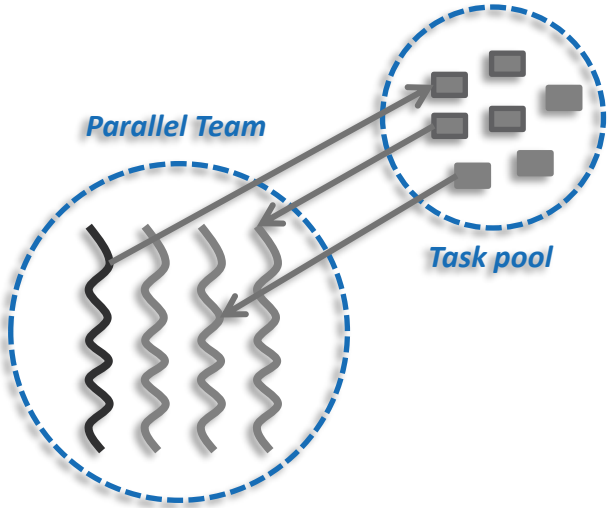
```
while ( <expr> ) {
    ...
}
```

  - → recursive functions

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

- Example (unstructured parallelism)

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



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

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)	**Synchronization**
→ 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 ☹

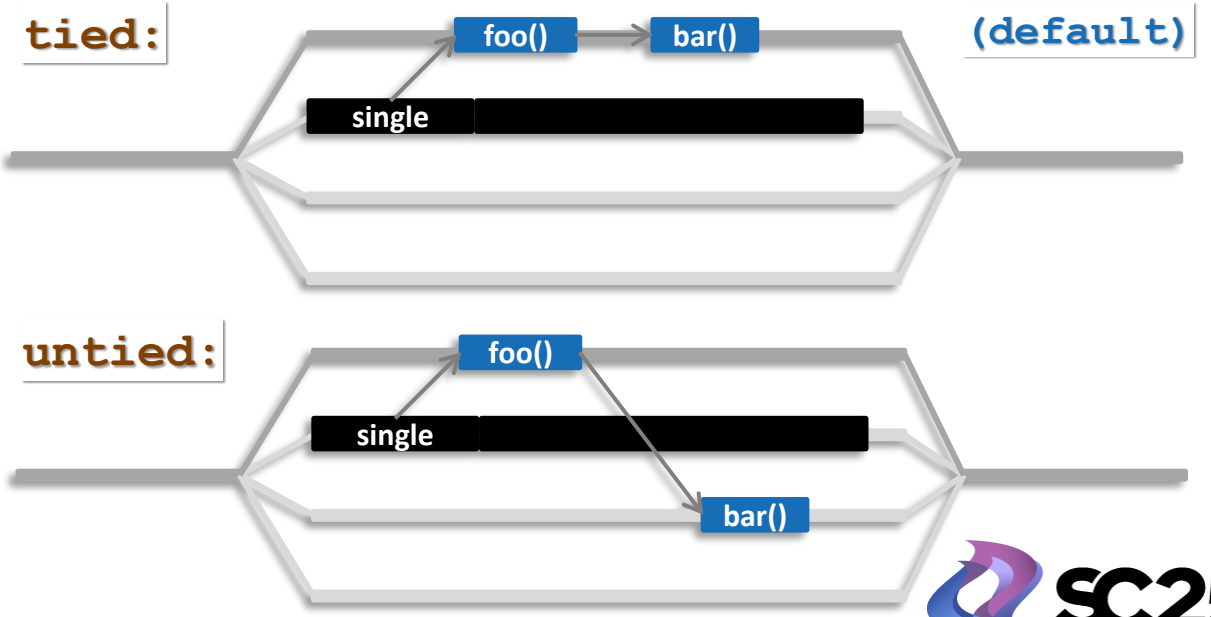
Task scheduling: taskyield directive

- 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

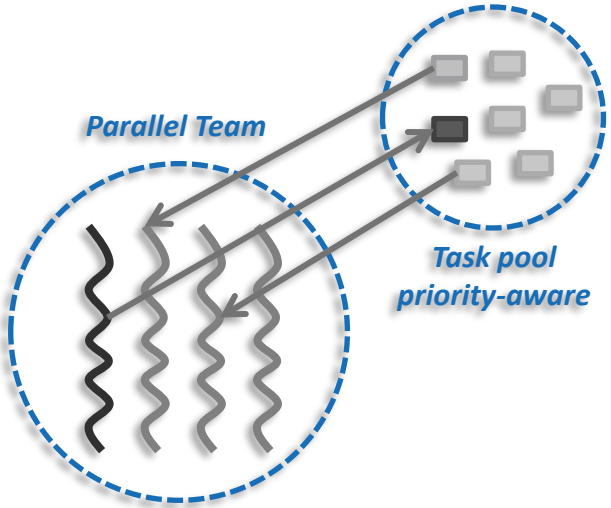
- Programmers may specify a priority value when creating a task

```
#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; }
  ...
}
```



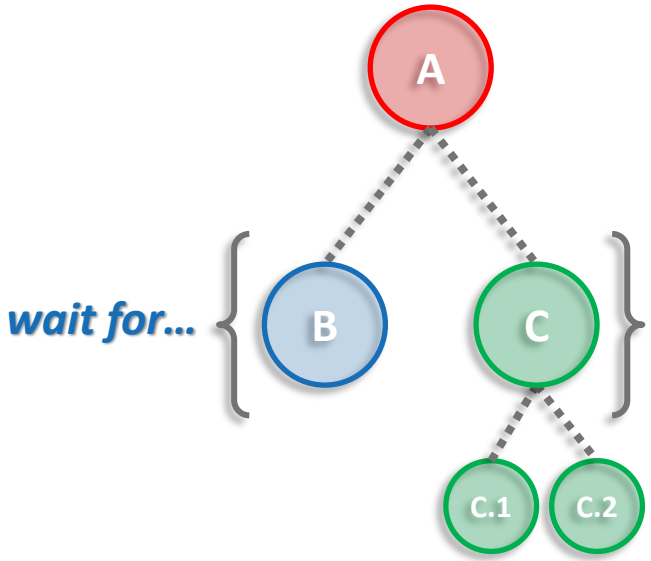
Task synchronization: taskwait directive

- 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                    : A
   {
      #pragma omp task                 : B
      { … }
      #pragma omp task                 : C
      { … #C.1; #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…

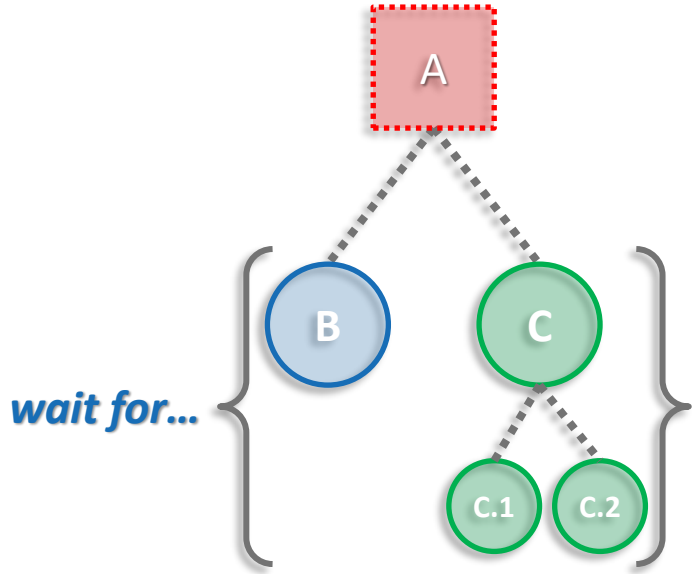
Task synchronization: taskgroup construct

- 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          : A
   {
      #pragma omp task            :B
      { … }
      #pragma omp task            :C
      { … #C.1; #C.2; …}

   } // end of taskgroup
}
```



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)…

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

5

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

4

```
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

```
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**

```
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:
      }
   }
}
```

- → Pre-determined rules (can 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

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 Use Cases

Tasking Use Case: Fibonacci (Recursion)

```
int comp_fib_numbers ( int n) {
  int fn1, fn2;

  if ( n == 0 || n == 1 ) return(n);

  #pragma omp task shared(fn1)
  fn1 = comp_fib_numbers(n-1);

  #pragma omp task shared(fn2)
  fn2 = comp_fib_numbers(n-2);

  #pragma omp taskwait

  return(fn1 + fn2);
}
```

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

Tasking Use Case: Cholesky (Synchronization)

```
void cholesky(int ts, int nt, double* a[nt][nt]) {
  for (int k = 0; k < nt; k++) {
    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
  }
}
```

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

Tasking Use Case: saxpy (Blocking/Tiling)

```
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 → to fine
  - → whole loop → no parallelism
- Manually transform the code
  - → blocking techniques
- Improving programmability
  - → OpenMP taskloop

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 → to 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**

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

Worksharing vs. taskloop constructs (1/2)

```fortran
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

```fortran
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)

```fortran
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

```fortran
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

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

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

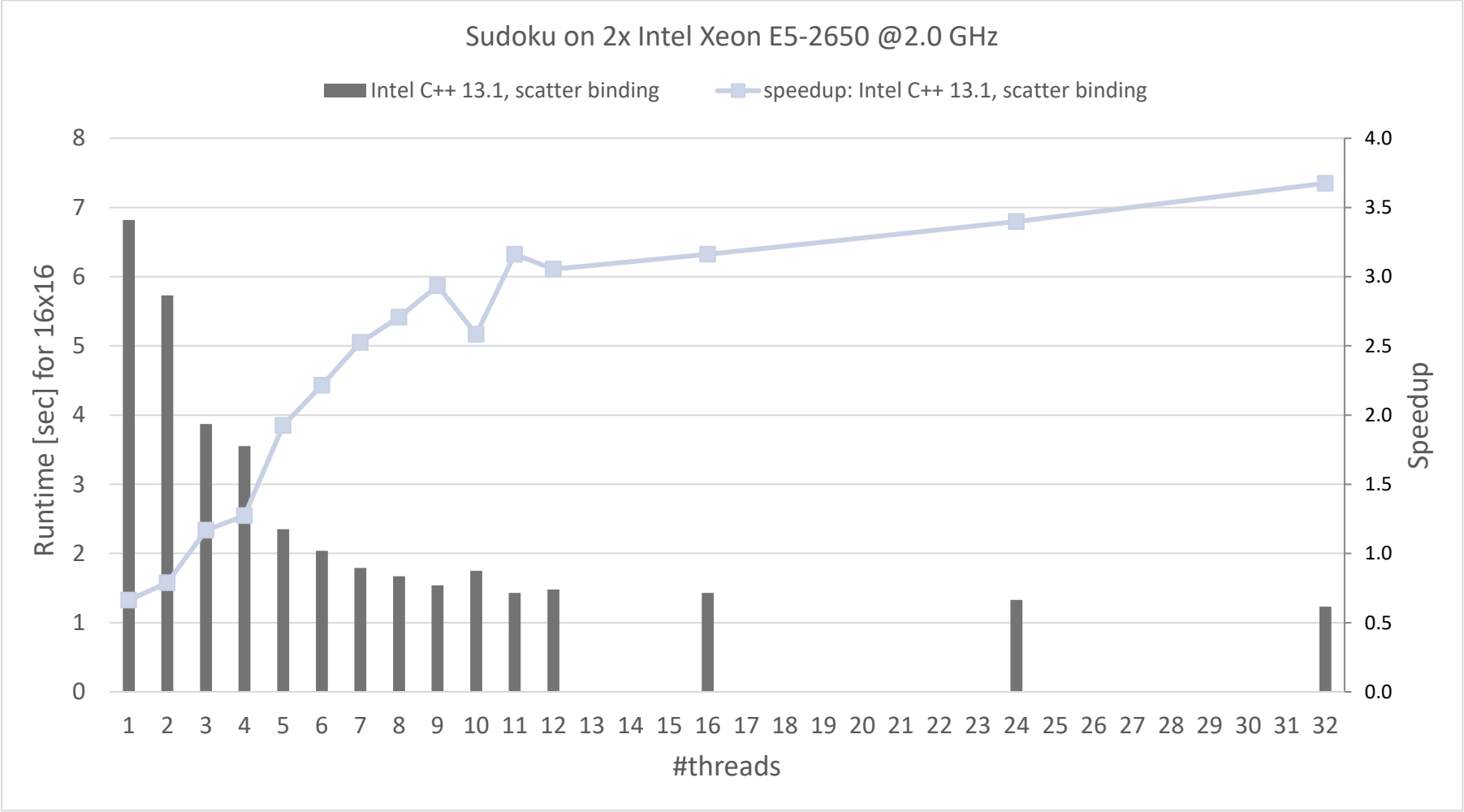
- (2) Try all numbers:
  - (2 a) Check Sudoku
    - If invalid: skip
    - If valid: Go to next field

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

- Wait for completion

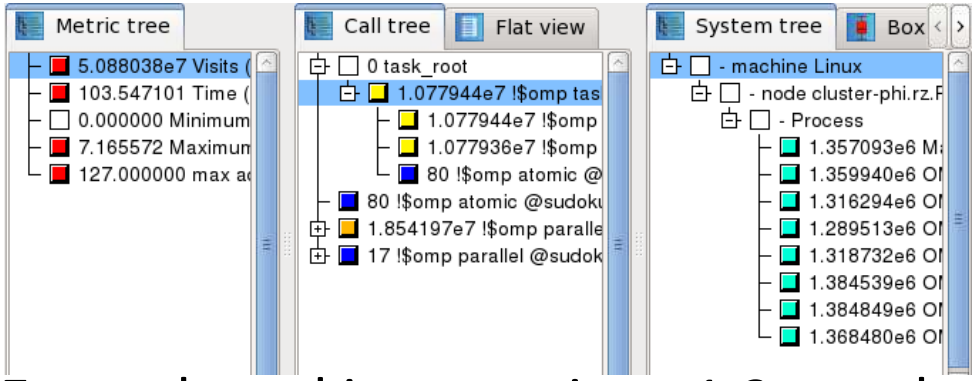
```
#pragma omp taskwait
```
wait for all child tasks

Performance Evaluation

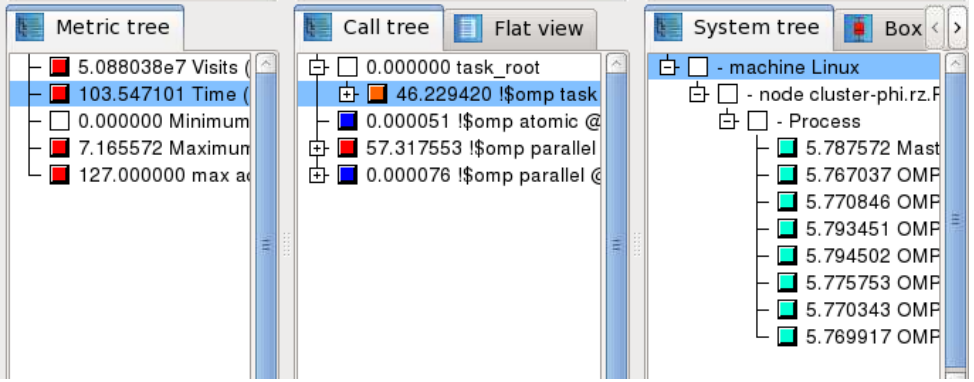


Performance Analysis

Event-based profiling provides a good overview :



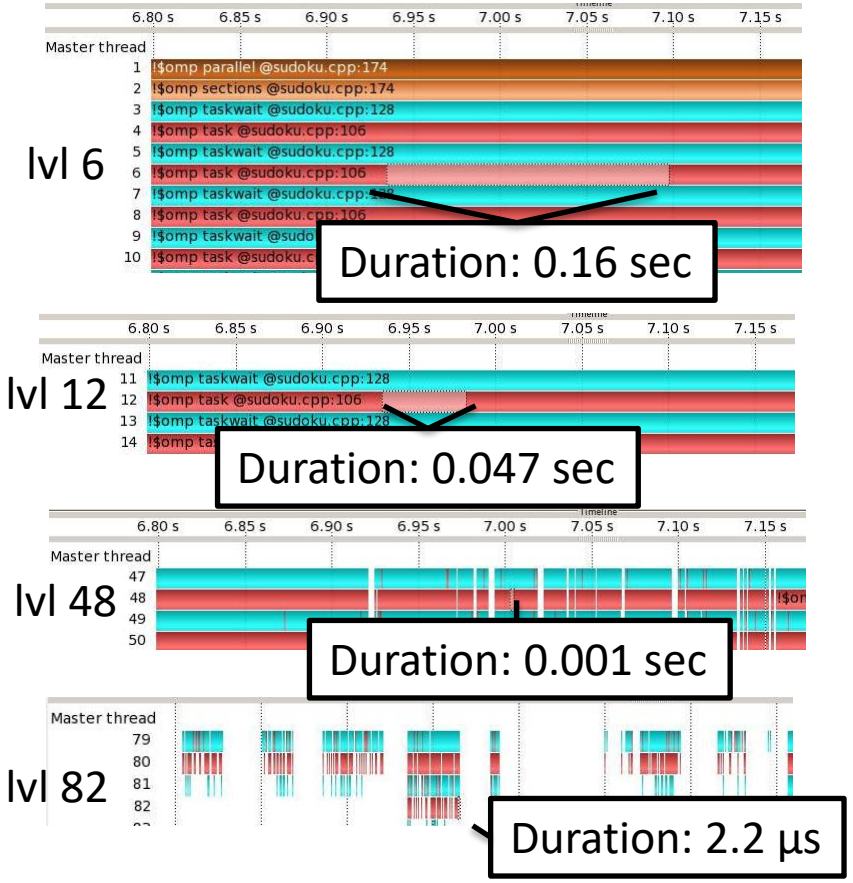
Every thread is executing ~1.3m tasks…



… in ~5.7 seconds.

=> average duration of a task is ~4.4 µs

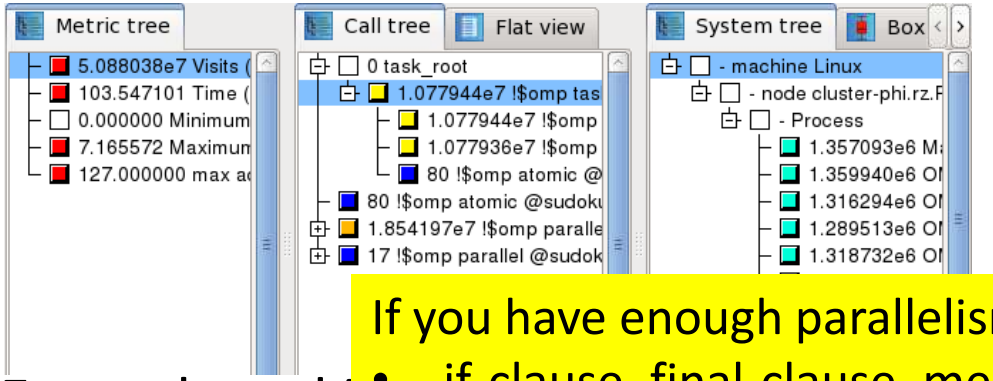
Tracing provides more details:



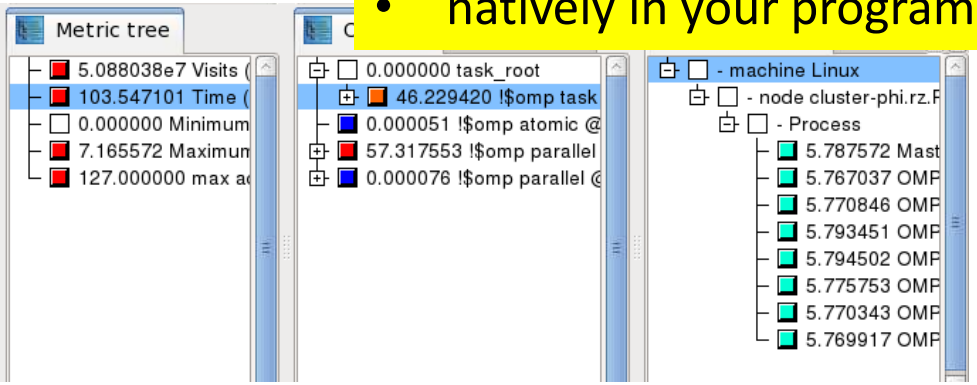
Tasks get much smaller down the call-stack.

Performance Analysis

Event-based profiling provides a good overview :



Every thread i



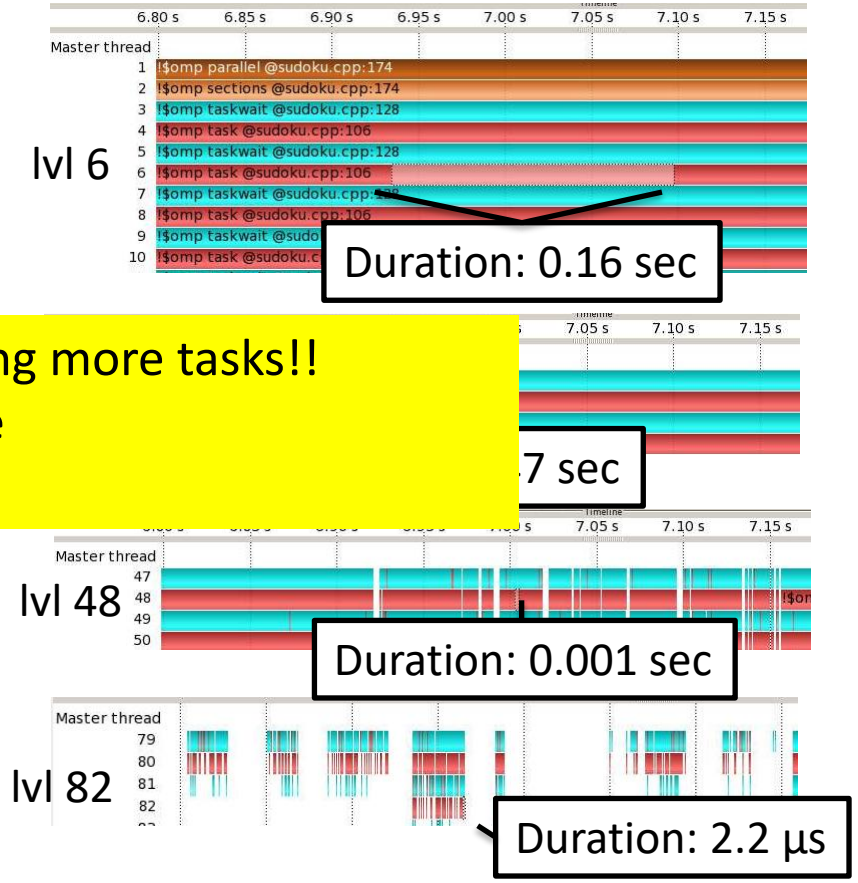
If you have enough parallelism, stop creating more tasks!!

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

… in ~5.7 seconds.

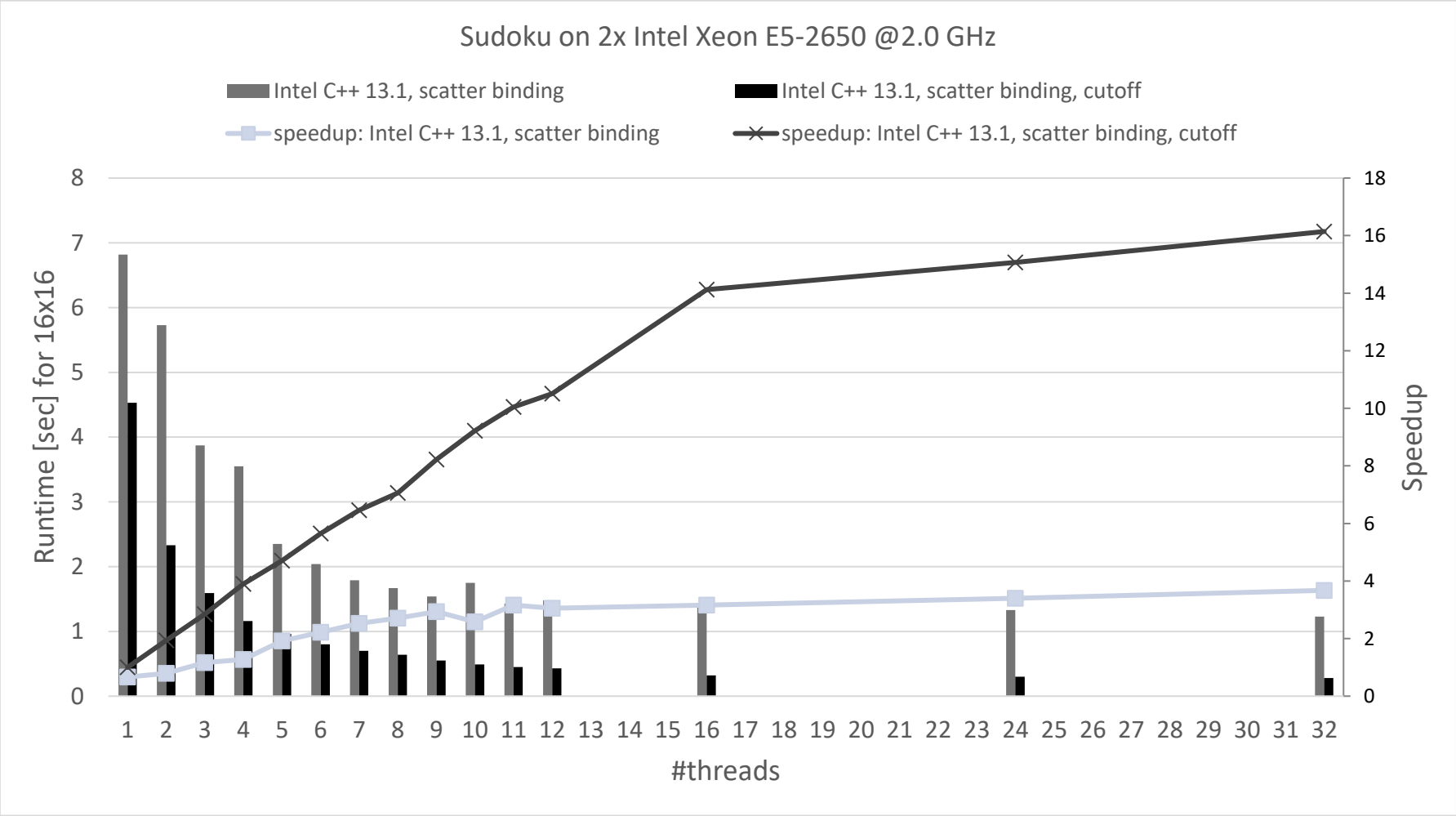
=> average duration of a task is ~4.4 µs

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 *undeferred task*

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

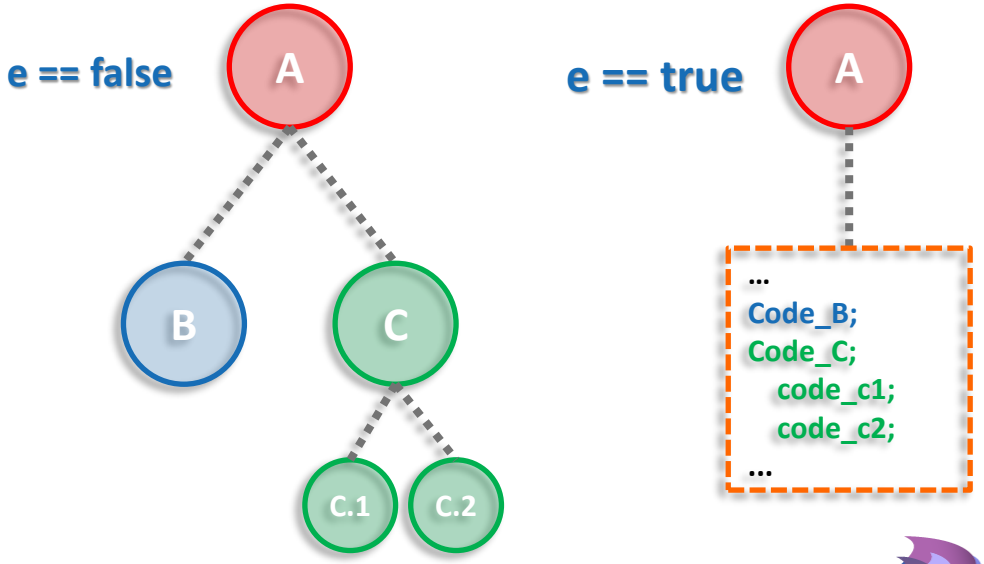
Really useful to debug tasking applications!

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

The `final` clause

- The `final(expression)` clause
  - → Nested tasks / recursive applications
  - → allows to avoid future task creation → reduces overhead but also reduces parallelism
- If the `expression` of the `final` clause evaluates to `true`
  - → The new task is created and executed normally but in its context all tasks will be executed immediately by the same thread (*included tasks*)

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



- Data-sharing clauses are honored too!

The `mergeable` clause

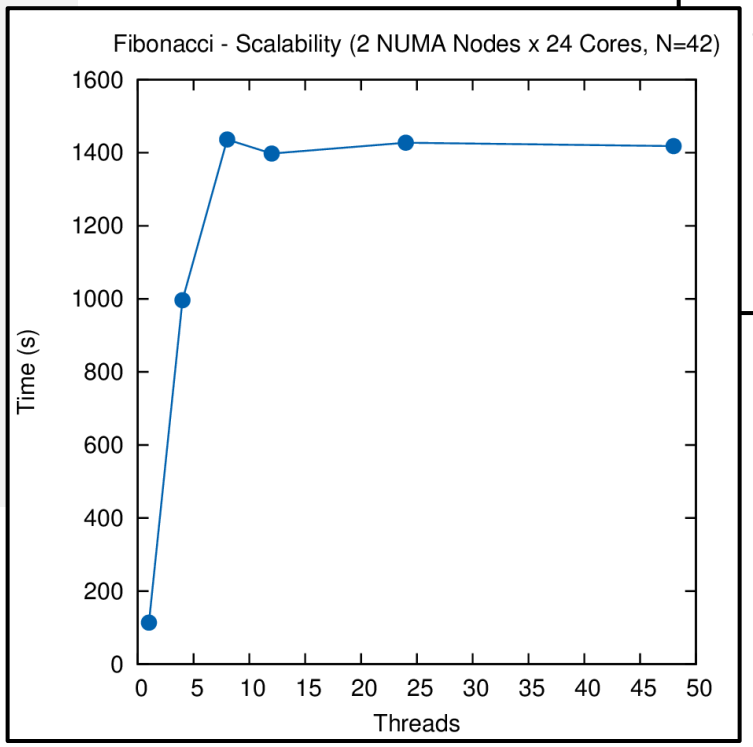
- The `mergeable` clause
  - → Optimization: get rid of “data-sharing clauses are honored”
  - → This optimization can only be applied in *undeferred* or *included tasks*
- A Task that is annotated with the `mergeable` clause is called a *mergeable task*
  - → A task that may be a *merged task* if it is an *undeferred task* or an *included task*
- A *merged task* is:
  - → A task for which the data environment (inclusive of ICVs) may be the same as that of its generating task region
- A good implementation could execute a merged task without adding any OpenMP-related overhead

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

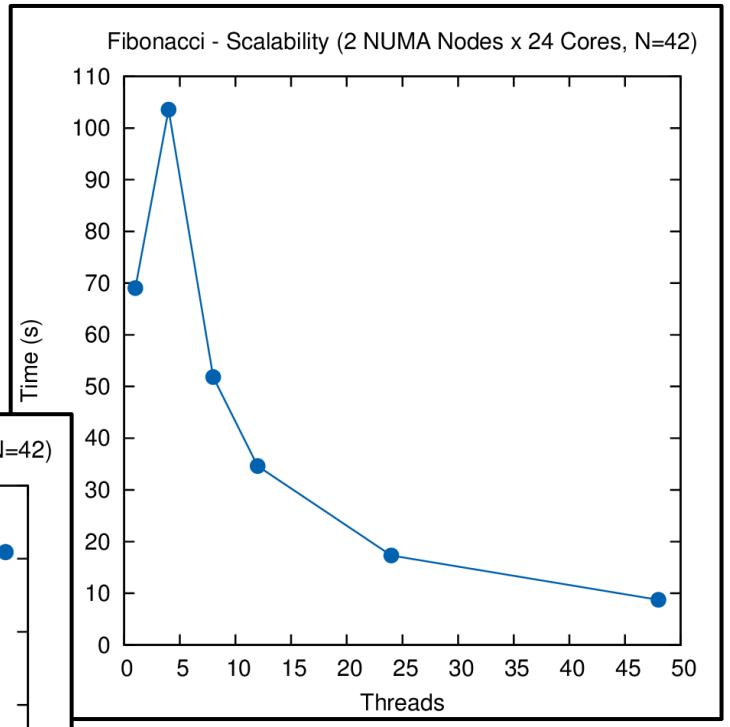
Example: Fibonacci

Fibonacci: without cutoff

```
int fib(int n) {
  if (n < 2)
    return n;

  int res1, res2;
  #pragma omp task shared(res1)
  res1 = fib(n-1);

  #pragma omp task shared(res2)
  res2 = fib(n-2);

  #pragma omp taskwait

  return res1 + res2;
}
```



gcc 7.2.0

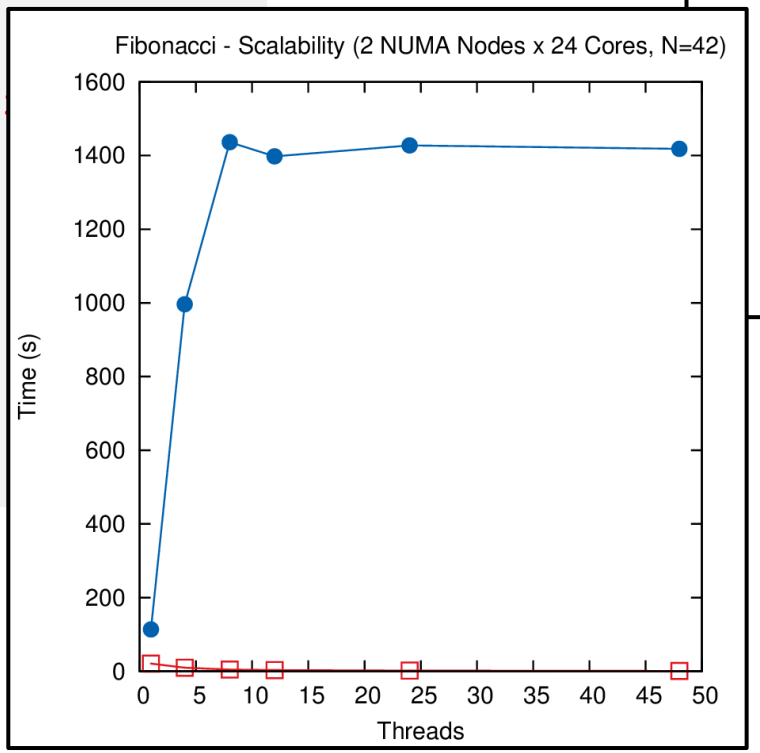


icc 2018.0

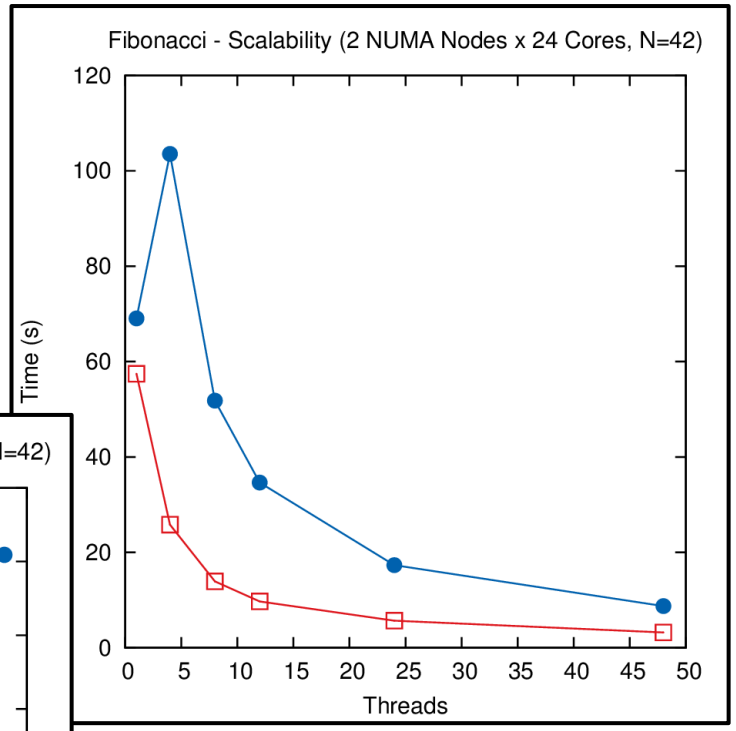
no_cutoff

Fibonacci: `if` clause

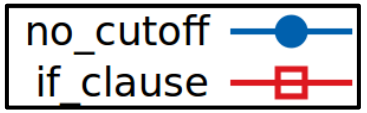
```
int fib(int n) {
  if (n < 2)
    return n;

  int res1, res2;
  #pragma omp task shared(res1) if(n > 30)
  res1 = fib(n-1);

  #pragma omp task shared(res2) if(
  res2 = fib(n-2);

  #pragma omp taskwait

  return res1 + res2;
}
```



gcc 7.2.0

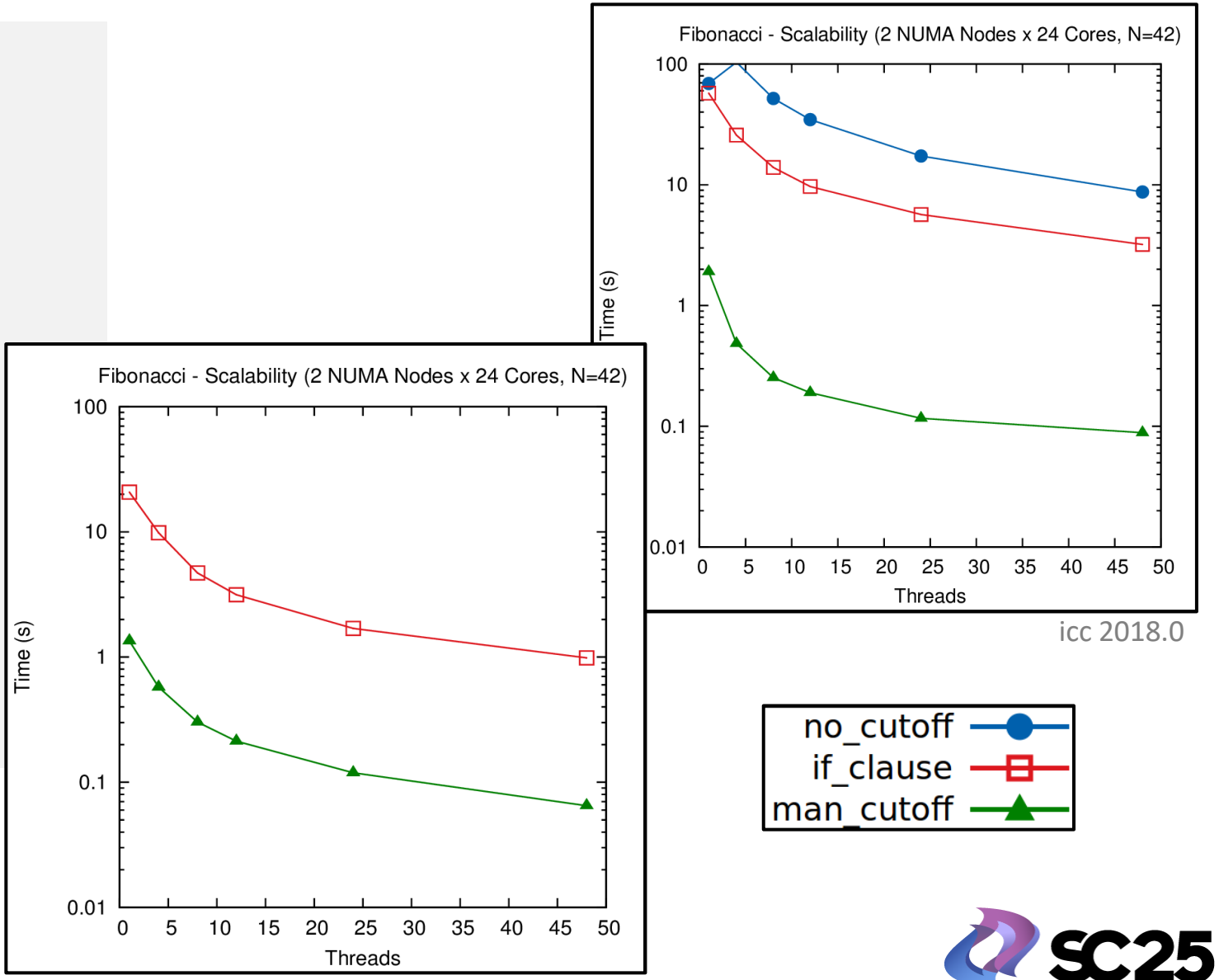


icc 2018.0



Fibonacci: manual optimization

```
int fib(int n) {
  if (n < 30)
    return fib_serial(n);

  int res1, res2;
  #pragma omp task shared(res1)
  res1 = fib(n-1);

  #pragma omp task shared(res2)
  res2 = fib(n-2);

  #pragma omp taskwait

  return res1 + res2;
}
```



icc 2018.0

gcc 7.2.0

Improving Tasking Performance: Task Affinity

Motivation

- Techniques for process binding & thread pinning available
  - → OpenMP thread level: `OMP_PLACES & OMP_PROC_BIND`
  - → OS functionality: `taskset -c`

OpenMP Tasking:

- In general: Tasks may be executed by any thread in the team
  - → Missing task-to-data affinity may have detrimental effect on performance

OpenMP 5.0:

- `affinity` clause to express affinity to data

`affinity` clause

- New clause: `#pragma omp task affinity (list)`
  - → Hint to the runtime to execute task closely to physical data location
  - → Clear separation between dependencies and affinity
- Expectations:
  - → Improve data locality / reduce remote memory accesses
  - → Decrease runtime variability
- Still expect task stealing
  - → In particular, if a thread is under-utilized

Code Example

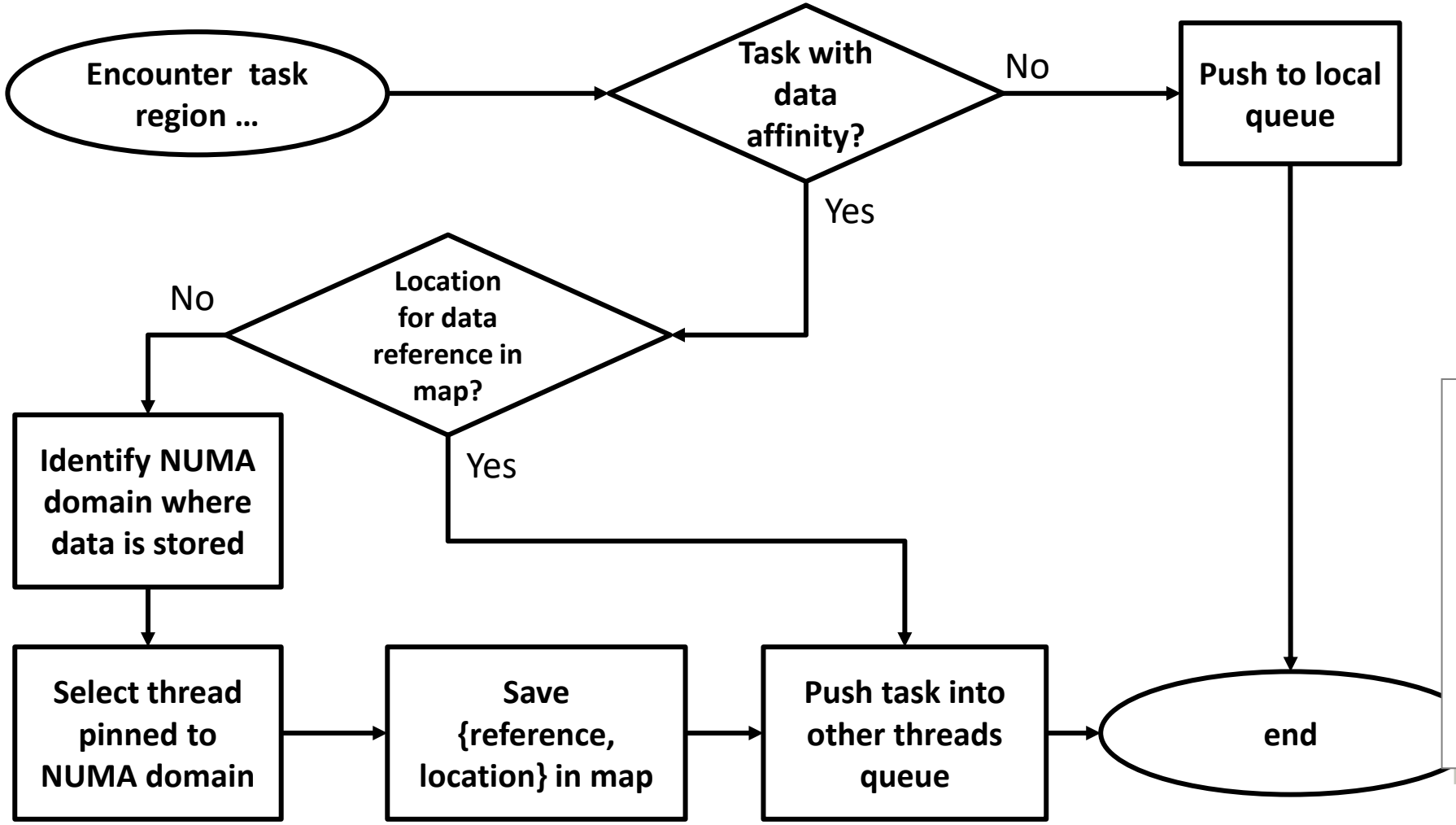
- Excerpt from task-parallel STREAM

```
#pragma omp task \
    shared(a, b, c, scalar) \
    firstprivate(tmp_idx_start, tmp_idx_end) \
    affinity( a[tmp_idx_start] )
{
   int i;
   for(i = tmp_idx_start; i <= tmp_idx_end; i++)
        a[i] = b[i] + scalar * c[i];
}
```

→Loops have been blocked manually (see `tmp_idx_start/end`)

→Assumption: initialization and computation have same blocking and same affinity

Selected LLVM implementation details

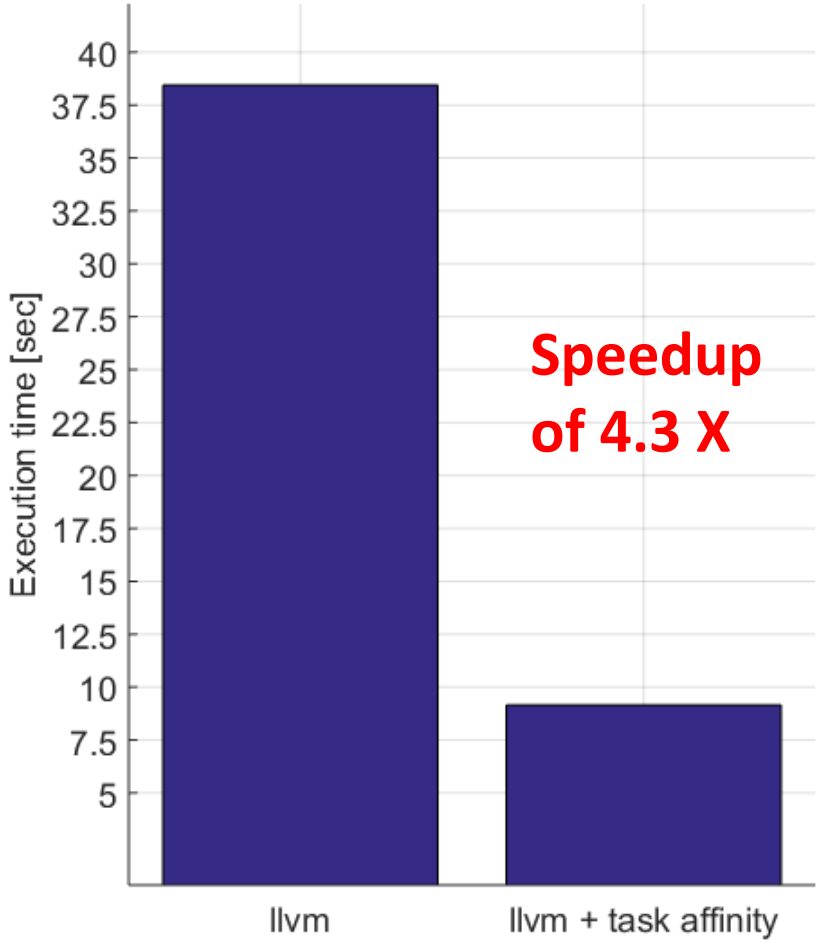


A map is introduced to store location information of data that was previously used

Jannis Klinkenberg, Philipp Samfass, Christian Terboven, Alejandro Duran, Michael Klemm, Xavier Teruel, Sergi Mateo, Stephen L. Olivier, and Matthias S. Müller. **Assessing Task-to-Data Affinity in the LLVM OpenMP Runtime**. Proceedings of the 14th International Workshop on OpenMP, IWOMP 2018. September 26-28, 2018, Barcelona, Spain.

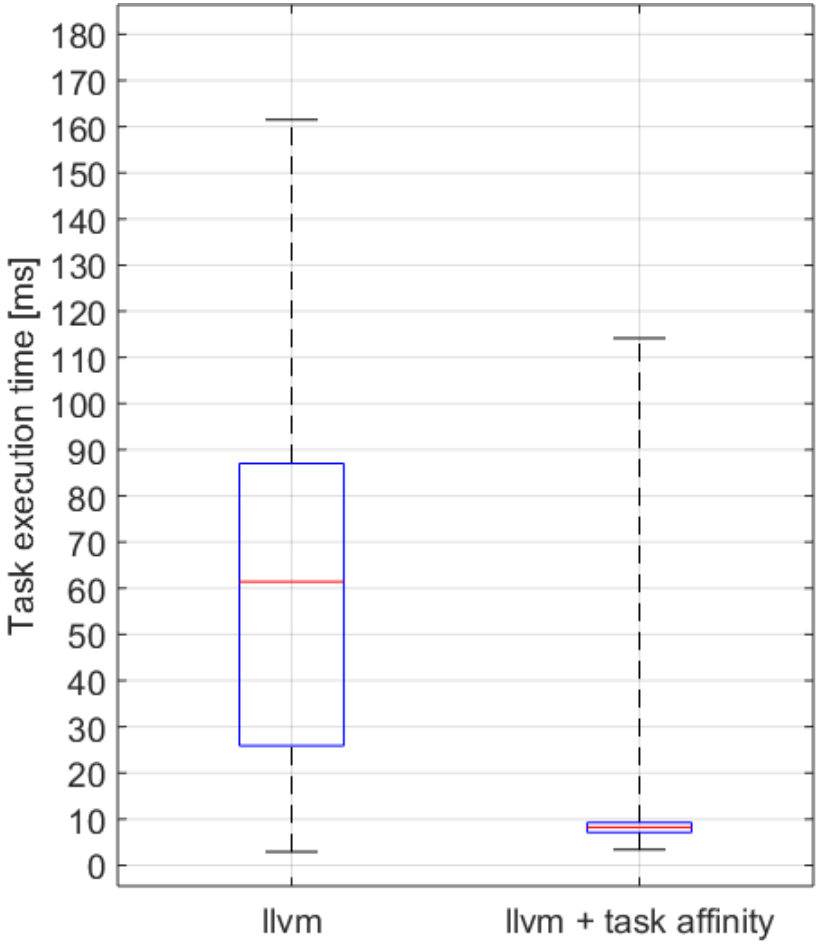
Evaluation (Merge-Sort, from paper above)

Program runtime
Median of 10 runs



Speedup of 4.3 X

Distribution of single task execution times



LIKWID: reduction of remote data volume from 69% to 13%

Summary

- Requirement for this feature: thread affinity enabled

- The `affinity` clause helps, if
  - → tasks access data heavily
  - → single task creator scenario, or task not created with data affinity
  - → high load imbalance among the tasks

- Different from thread binding: task stealing is absolutely allowed

Improving Tasking Performance: Task dependences

Motivation

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

OpenMP 3.1

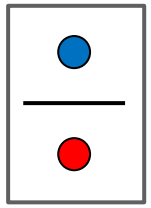
```
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task
  std::cout << x << std

  #pragma omp taskwait

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

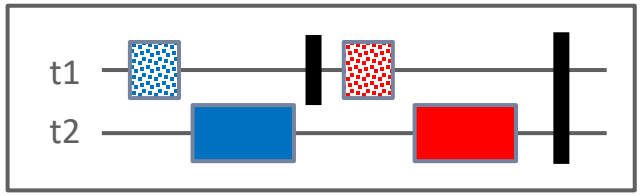
OpenMP 4.0

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

                                end(inout: x)
  x++;
}
```

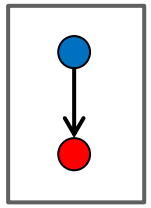
Task dependences can help us to remove "strong" synchronizations, increasing the look ahead and, frequently, the paralelism!!!!

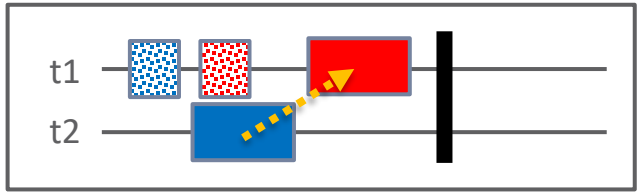
OpenMP 3.1





OpenMP 4.0



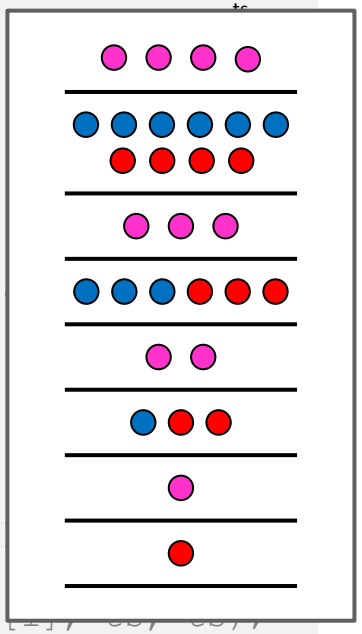


Task's creation time

Task's execution time

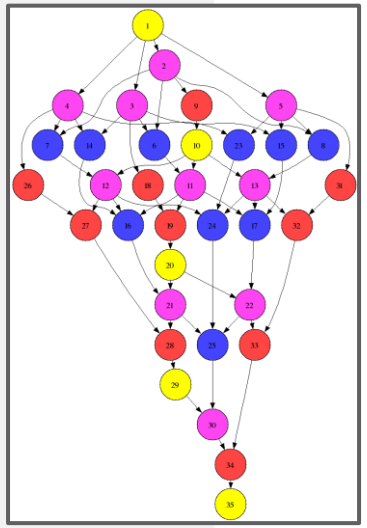
Motivation: Cholesky factorization

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



OpenMP 3.1

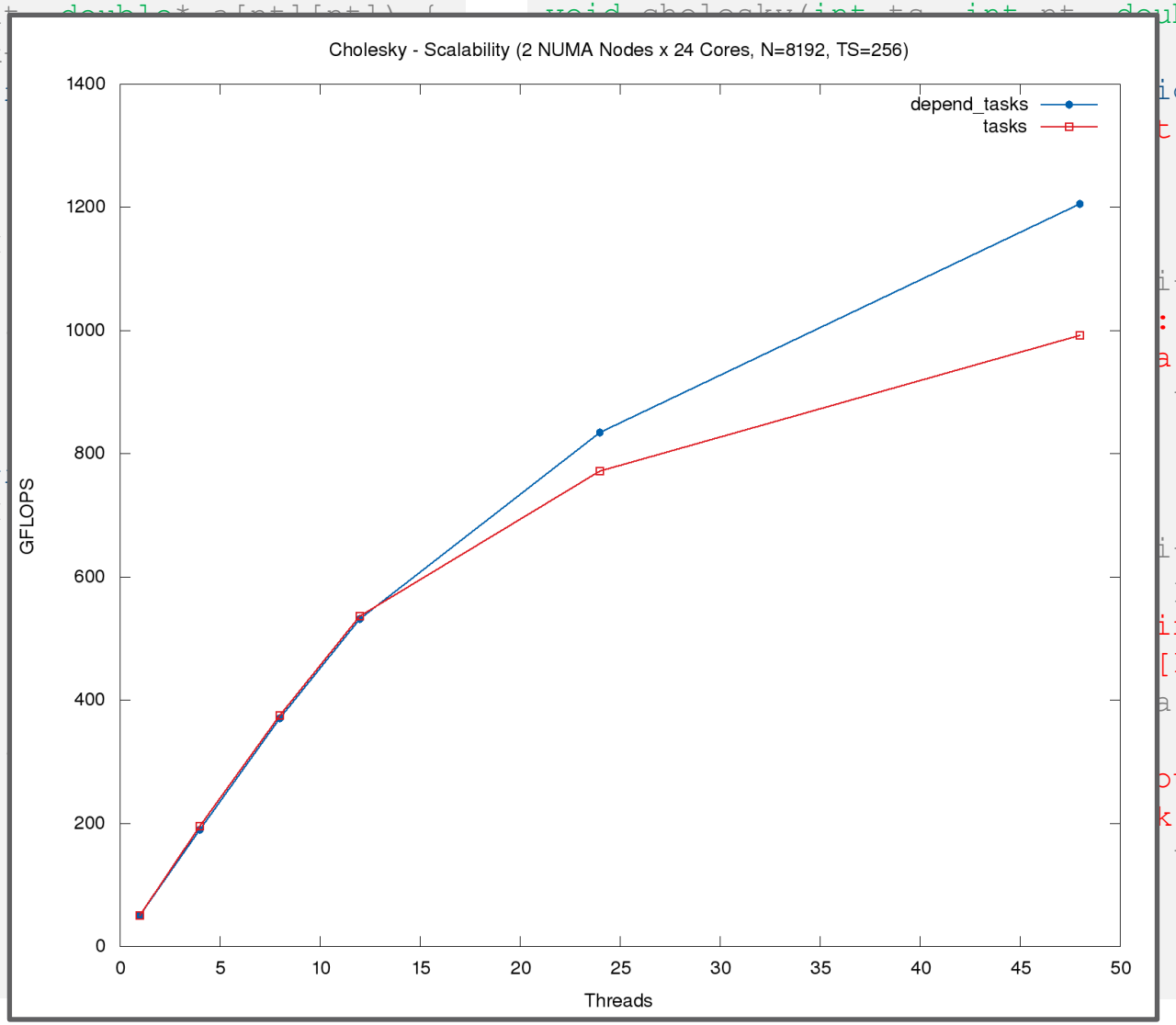
```
void cholesky(int ts, int nt, double* a[nt][nt]) {
  for (int k = 0; k < nt; k++) {
    // Diagonal Block factorization
    #pragma omp task depend(inout: a[k][k])
    potrf(a[k][k], ts, ts);

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

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



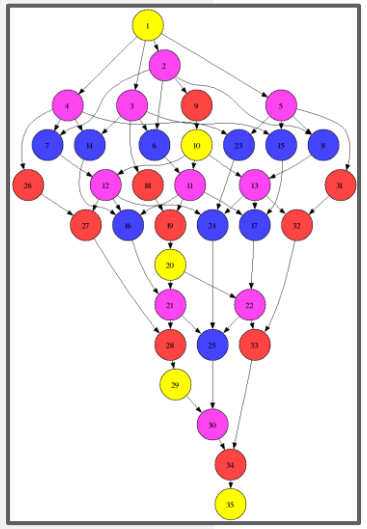
OpenMP 4.0

Motivation: Cholesky factorization

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

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

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




```
void cholesky(int ts, int nt, double* a[nt][nt]) {
                                                  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

Using 2017 Intel compiler

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`
- Dependable objects

OpenMP 5.1

- New dependency type: `inoutset`

What's in the spec: syntax depend clause

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

where:

→ `depend-modifier` is used to define iterators

→ `dependency-type` may be: `in`, `out`, `inout`, `inoutset`, `mutexinoutset` and `depobj`

→ A `list-item` may be:

- C/C++: A `lvalue` expr or an array section `depend(in: x, v[i], *p, w[10:10])`
- Fortran: A variable or an array section `depend(in: x, v(i), w(10:20))`

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

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

- If a task defines an `in` dependence over a list-item
  - → the task will depend on all previously generated sibling tasks that reference that list-item in an `out` or `inout` dependence

- If a task defines an `out`/`inout` dependence over list-item
  - → the task will depend on all previously generated sibling tasks that reference that list-item in an `in`, `out` or `inout` dependence

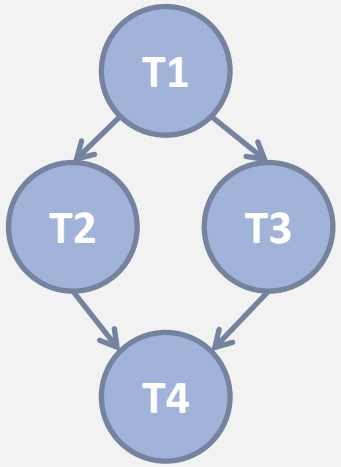
What's in the spec: sema depend clause (1)

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

- If a task defir
  - → the task will c ... tem in an `out` or `inout` deper

- If a task defir
  - → the task will c ... tem in an `in`, `out` or `inout` deper

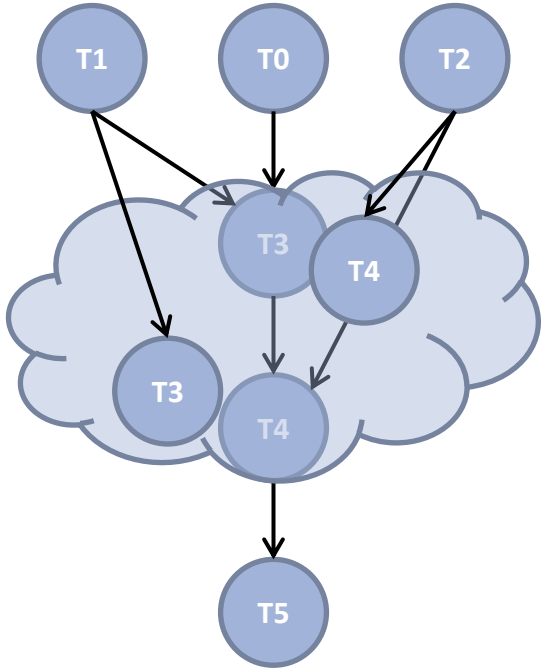
```
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(inout: x) //T1
  { ... }

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

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

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



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

- Set types: `inoutset` & `mutexinoutset`

```
int x = 0, y = 0, res = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(out: res)  //T0
   res = 0;

  #pragma omp task depend(out: x)  //T1
  long_computation(x);

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

  #pragma omp task depend(in: x) depend(mutexinoutset: 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;
}
```



1. *inoutset property*: tasks with a `mutexinoutset` dependence create a cloud of tasks (an inout set) that synchronizes with previous & posterior tasks that dependent on the same list item

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

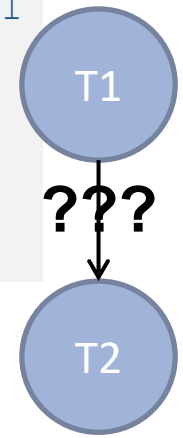
What's in the spec: sema depend clause (3)

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

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

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

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

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

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



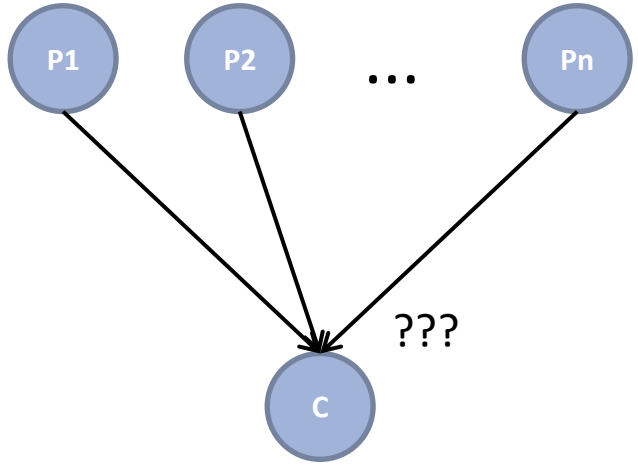
What's in the spec: sema depend clause (4)

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

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

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

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

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

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



Philosophy

Philosophy: data-flow model

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

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

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

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

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

OK, but it always prints ‘0’ :(

We have a data-race!!

Philosophy: data-flow model (2)

- Properly combining dependences and data-sharings allow us to define a **task data-flow model**
  - → Data that is read in the task → input dependence
  - → Data that is written in the task → output dependence

- A task data-flow model
  - → Enhances the **composability**
  - → **Eases the parallelization** of new regions of your code

Philosophy: data-flow model (3)

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

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

```
//test1_v2.cc
```

```
//test1_v3.cc
```

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

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

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

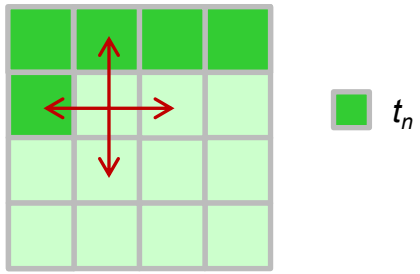
Use case

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

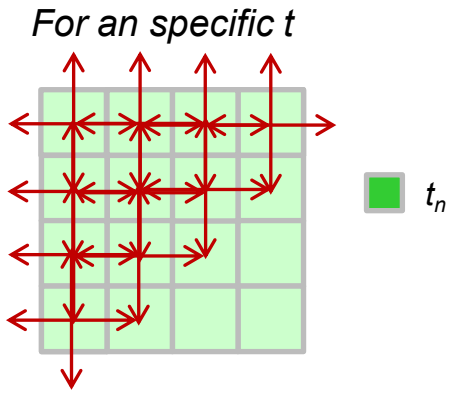


Each cell depends on:
- two cells (north & west) that are computed in the current time step, and
- two cells (south & east) that were computed in the previous time step

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
      }
    }
  }
}
```

1st parallelization strategy



We can exploit the wavefront to obtain parallelism!!

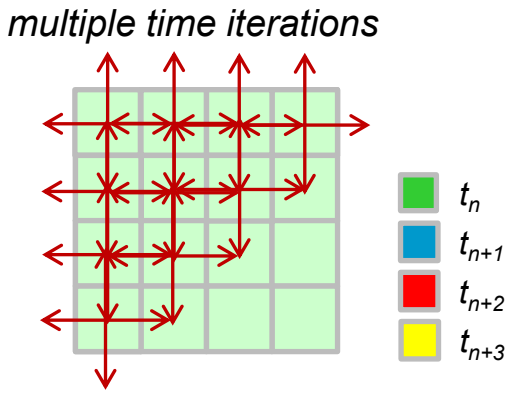
Use case : Gauss-seidel (3)

```
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
      }
    }
  }
}
```

2nd parallelization strategy

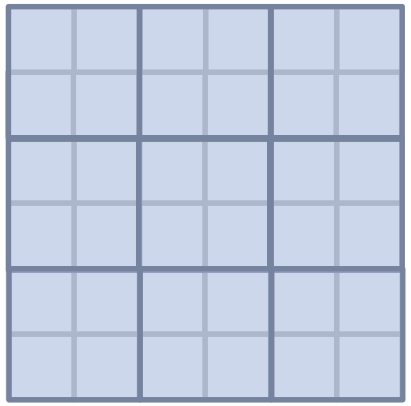


We can exploit the wavefront of multiple time steps to obtain MORE parallelism!!

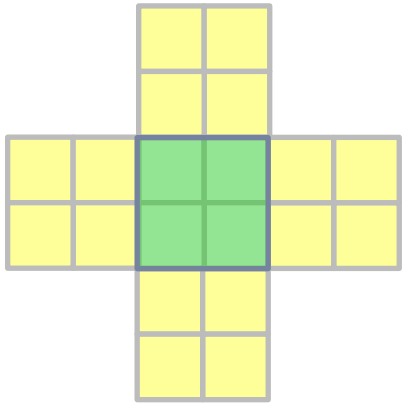
Use case : Gauss-seidel (5)

```
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]);
        }
      }
}
```

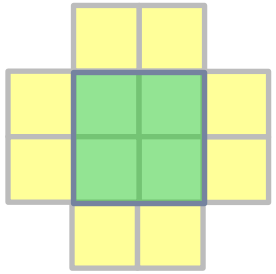
inner matrix region



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

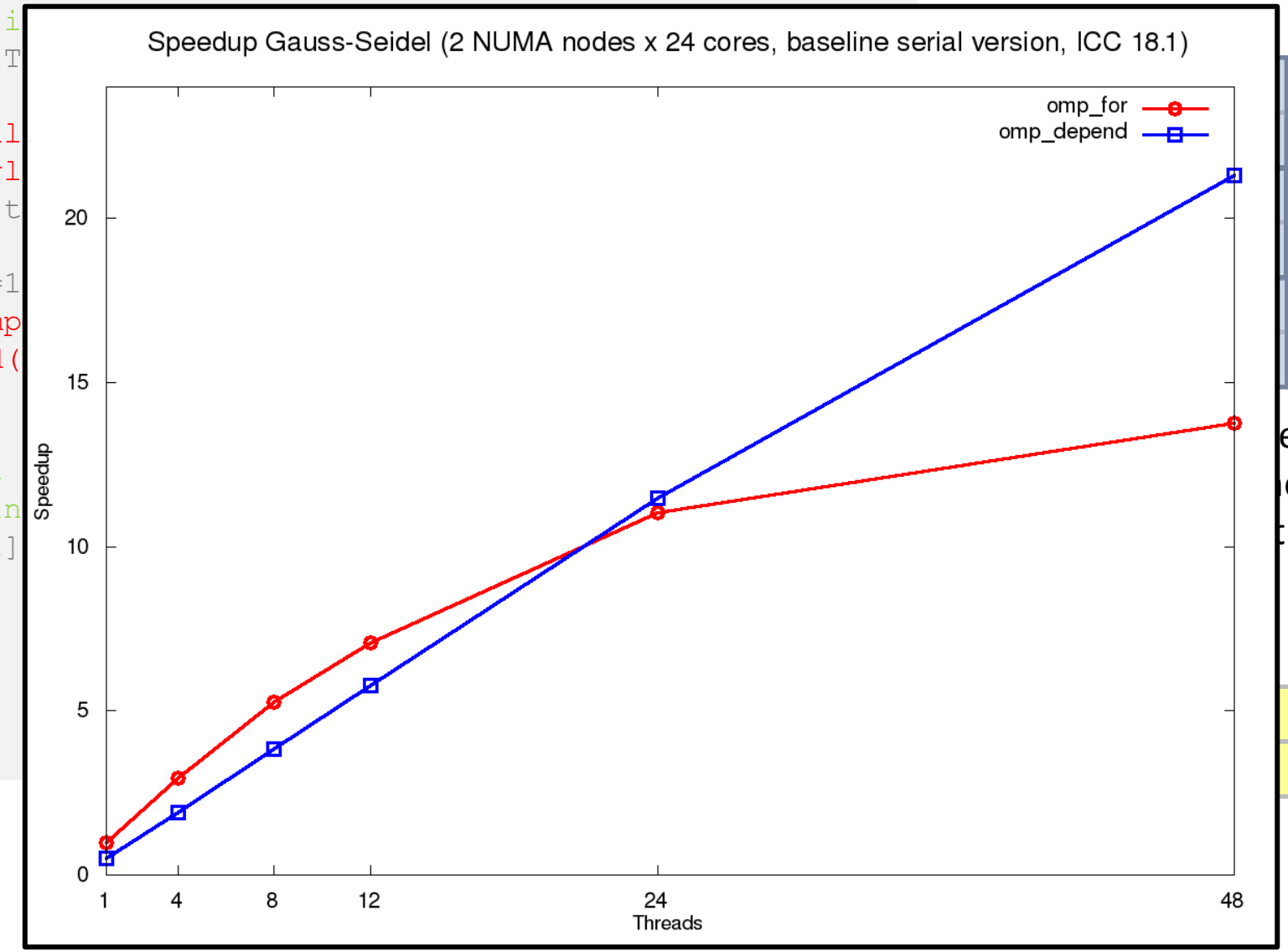


vs

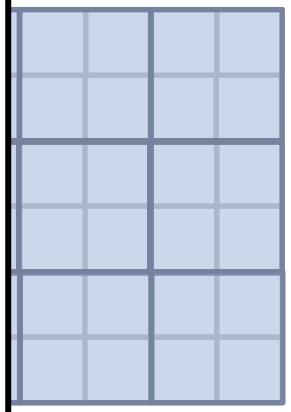


Use case : Gauss-seidel (5)

```
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]

        }
      }
}
```

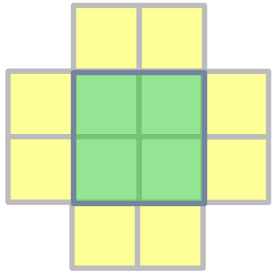


matrix region



e input dependences
e whole block rather
t a column/row?

vs



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;
}
```

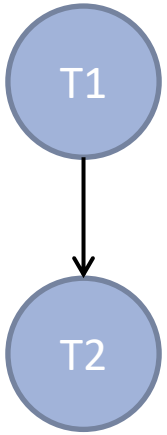
Advanced features: dependable objects (1)

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

Advanced features: dependable objects (2)

- Offer a way to manually handle dependences

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

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




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

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

  #pragma omp depobj(obj) update(in)

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

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

Task iterations

(OpenMP 6.0 Feature)

Task iteration construct

- It controls the per-iteration task-execution attributes of the generated tasks
  - → Subsidiary directive of the **`taskloop`** construct; associated to the innermost enclosing **`taskloop`**

```
#pragma omp task_iteration [clause[[,] clause]...]
{loop-body}
```

- → if(scalar-expression)
- → depend(dep-type: list)
- → affinity(list)

- → For each generated task in the enclosing directive the behavior is as if it was specified with the corresponding clauses
- → Clauses are instantiated for each instance of the loop-iteration variables; where the if clause evaluate true

- Task iteration restrictions:
  - → Must appear in one of the taskloop-affected loops; affected can be several if taskloop includes collapse
  - → Must precede all statements and directives (other than task_iteration) in that loop body
  - → If present, no intervening code may occur between collapsed loops

Task iteration (Taskloop + Tasks)

```
#pragma omp taskloop nogroup
for (int i=0; i<n; i++) {
   #pragma omp task_iteration depend(inout: A[i])
   A[i] += <expression>;
}

#pragma omp task depend(in: A[0])
Compute (<…params…>, A[0]);

#pragma omp task depend(in: A[n-1])
Compute (<…params…>, A[n-1]);
```

→ Generated tasks will have extra information @ loop body

→ These two tasks will depend on the "corresponding" generated tasks in the previous taskloop

Task iteration (Taskloop + Taskloop)

```
#pragma omp taskloop nogroup
for (int i=0; i<n; i++) {
   #pragma omp task_iteration depend(inout: A[i])
   A[i] += <expression>;
}
```

→ Generated tasks will have extra information @ loop body

```
#pragma omp taskloop
for (int i=0; i<n; i++) {
   #pragma omp task_iteration depend(in: A[i])
   B[i] = f(A[i]);
}
```

→ Generated tasks will have extra information @ loop body

Task iteration (w/ the strict modifier)

```
#pragma omp taskloop grainsize(strict: 256) nogroup
for (int i=0; i<n; i++) {
   #pragma omp task_iteration depend(inout: A[i]) \
               if((i%256) == 0)
   A[i] += <expression>;
}
```

→ Forcing chunk size

→ It will only consider the “first” iteration of each chunk/task

```
#pragma omp taskloop grainsize(strict: 256)
for (int i=0; i<n; i++) {
   #pragma omp task_iteration depend(in: A[i]) \
               if((i%256) == 0)
   B[i] = f(A[i]);
}
```

→ It will only consider the “first” iteration of each chunk/task

Final task iteration considerations

- For the associated taskloop the creation order is the increasing collapsed iteration order wrt their assigned chunks
- General task dependence restrictions still apply
  - → List items must indicate **identical** or **disjoint** storage locations
  - → Cannot be zero-length array sections,…
- Further interesting use cases (not shown in the examples)
  - → Expressing loop carried dependences
  - → Combining with the taskloop collapse clause
- **Affinity:** same examples could be rewritten with the affinity clause

Free-agent threads

(OpenMP 6.0 Feature)

Recall the tasking execution model

- Supports unstructured parallelism
  - → unbounded loops

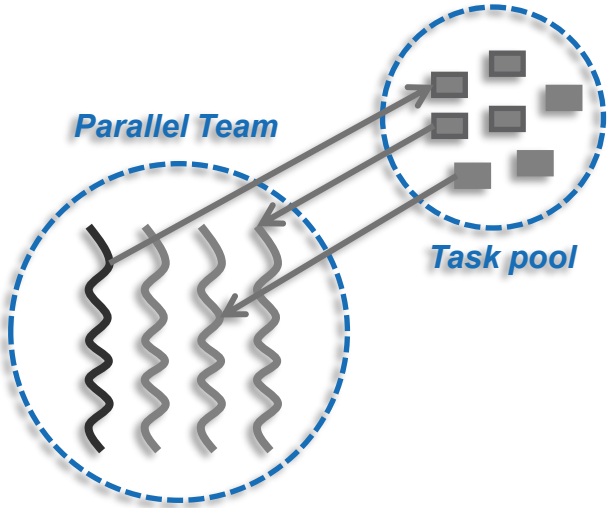
```
while ( <expr> ) {
    ...
}
```

  - → recursive functions

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

- Example (unstructured parallelism)

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



- Why are the **`parallel`** and **`single`** directives needed?
  - → Otherwise all threads in the team generate (duplicate) tasks
  - → Only threads in the team may execute tasks

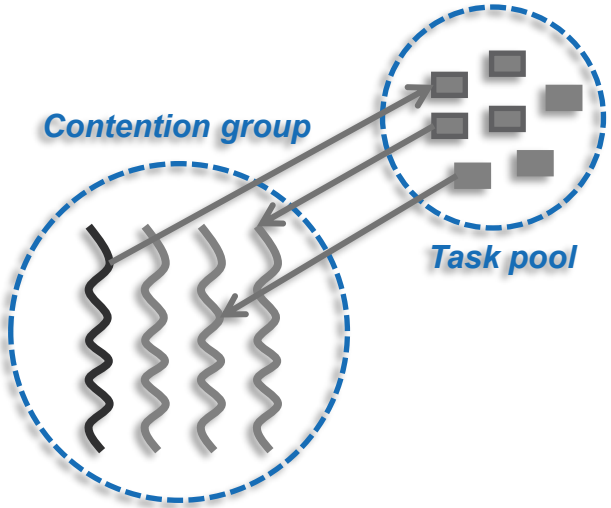
Is restricting tasks to a team good?

- Positive aspects
  - → Simplifies resource management
  - → Clear semantics with respect to other teams
- Negative aspects
  - → Ignores unutilized resources
  - → Complicates code structure for task-only programs

- Example (no **`parallel`** directive needed)

```
while (elem != NULL) {
   #pragma omp task threadset(omp_pool)
      compute(elem);
   elem = elem->next;
}
```

- Alternative starting in OpenMP 6.0: free-agent threads
  - → Unassigned threads in contention group may execute tasks
  - → Can provide parallelism in the implicit parallel region
  - → Exploits unused resources, common practice of parked threads



Some details for free-agent threads

- Existing behavior is preserved by default
  - → As if **`threadset`** clause is specified with value of **`omp_team`**

```
#pragma omp task threadset(omp_team)
{structured-block}
```

  - → Tasks are still tied by default so free-agent thread executes the task completely if at all
  - → Task synchronization (e.g., dependences, **`taskwait`** and **`taskgroup`**) unchanged
- Can use environment variables to control ICVs to reserve threads

```
setenv OMP_THREADS_RESERVE "structured(2),free_agent(2)"
```

  - → At least two threads available for structured parallelism, at least two available to act as free-agents
  - → Minimum for structured parallelism is one (the initial thread)
  - → Sum of reservations should not exceed *thread-limit-var* ICV

Future Directions

OpenMP 6.0 includes many major new features

- Officially released on November 14, 2024
  - → Reflects three years of work since release of OpenMP 5.2
  - → Includes 416 enacted issues, covering a wide range of content and complexity
- Free-agent threads significantly change execution model, implementations
- New concept for task dependences: transparent tasks
  - → Enables asynchronous `target data` (also enables other future extensions)
- User-defined induction and `induction` clause expand parallelism support
- Many significant device support improvements (e.g., `workdistribute`)
- Several additional (sequential) loop transforming directives
- Supported compound constructs are now defined based on a grammar
- Significant improvements to usability and correctness of specification

OpenMP tasking advances have pervasive impact

- Other major additions to 6.0 include:
  - → Support for dependences and affinity of tasks generated by `taskloop` directives
  - → A new `taskgraph` directive that enables optimized task generation
- Task-generating constructs are fundamental to OpenMP offload model
  - → Most device constructs (e.g., `target` and `target_update` directives) already generate them
  - → Another major change: `target_data` is now a dependence sequence of three tasks
    - → Middle task is transparent by default
    - → The construct now is also a `taskgroup` region by default
    - → Can specify `no_wait` and `no_group` to rely only on dependences for ordering
- Other constructs (e.g., `parallel` and `teams`) are composed of implicit tasks
  - → While not adopted for 6.0, expect to add `transparent` clause to many of them eventually
  - → Will enable `no_wait` to be supported for `parallel` construct

Topics

- Current OpenMP Language Committee Activities
- OpenMP Organizational Overview
- Final Review of OpenMP 5.0, 5.1, 5.2 and 6.0 (included for reference)

OpenMP Language Committee Current Activities: TR14 and OpenMP 6.1

OpenMP 6.1 will refine and amend OpenMP 6.0

- Significant progress has already been made
  - → 18 issues have been adopted, mostly covering small updates to 6.0 additions
  - → Language committee face-to-face meeting week after next will result in many more issues moving forward
- Targeting some significant improvements for device support
  - → Support for dynamic groupprivate memory (e.g., small, optimized GPU memory pool) (done!)
  - → Support for explicit control of pointer attachment (done!)
  - → Improved support for implicit `declare target` in Fortran
  - → Beginning work on “kernel language”, which will provide more low-level device control
- Expect continued refinement in many other areas
  - → More loop transformations, refinements of other ones
  - → Working on mechanism to control OpenMP defaults used for a translation unit
  - → Considering additional extensions that build on transparent tasks (e.g., `parallel nowait`)
  - → Many other small changes, particularly related to tasking and tool support, are likely

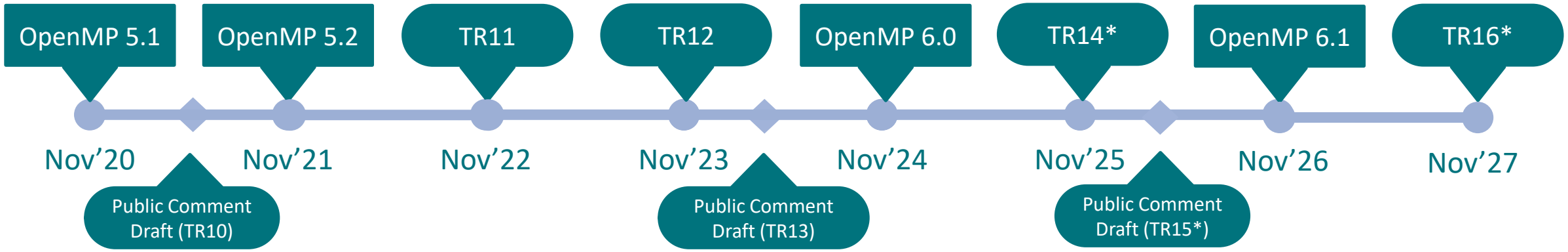
Things likely to be deferred to beyond 6.1

- True support for using multiple devices
  - → Device-to-device scoping support for atomic and other memory operations
  - → Support for bulk launch
  - → Support to update data on multiple devices (broadcast/multicast, other collectives)
  - → Support for work distribution across devices
  - → Considering relaxing restrictions on nested `target` regions
- Support for pipelining, data-flow, other parallelization models
- Support for event-based parallelism
- Characterizing loop-based work distribution constructs as transformations
- Efficient use of multiple compilation units (i.e., support for efficient IPO)

OpenMP Organizational Overview

OpenMP Roadmap

- OpenMP has a well-defined roadmap:
  - → 5-year cadence for major releases
  - → One minor release in between
  - → OpenMP 5.2 was added as a second minor release before OpenMP version 6.0
  - → (At least) one Technical Report (TR) with feature previews in every year



* Numbers assigned to TRs may change if additional TRs are released.

Development Process of the Specification

- Modifications to the OpenMP specification follow a (strict) process:

- Release process for specifications:

User Outreach & Education

Check out openmp.org/news/events-calendar/

Help Us Shape the Future of OpenMP

- OpenMP continues to grow
  - → 32 members currently
- You can contribute to our annual releases
- Attend IWOMP, understand and shape research directions
- OpenMP membership types now include less expensive memberships
  - → Please let us know if you would be interested

Final Review of OpenMP 5.0, 5.1, 5.2 and 6.0 Included for Reference

Ratified OpenMP 5.0 in November 2018, Ratified OpenMP 5.1 in November 2020

- OpenMP 5.0
  - → Addressed several major open issues for OpenMP
  - → Included 293 passed tickets
- OpenMP 5.1
  - → Includes many refinements to 5.0 additions
  - → Included 254 passed GitHub issues
- OpenMP 5.2
  - → Mostly address quality of specification issues but also refines 5.0 and 5.1 additions
  - → Included 131 passed GitHub issues

Major new features in OpenMP 5.0

- Significant extensions to improve usability
  - → OpenMP contexts, `metadirective` and `declare variant`
  - → Addition of `requires` directive, including support for unified shared memory
  - → Memory allocators and support for deep memory hierarchies
  - → Descriptive `loop` construct
  - → Ability to quiesce OpenMP threads
  - → Support to print/inspect affinity state
  - → Release/acquire semantics added to memory model
  - → Support for C/C++ array shaping
- First (OMPT) and third (OMPD) party tool support

Major new features in OpenMP 5.0

- Some significant extensions to existing functionality
  - → Verbosity reducing changes such as implicit `declare target` directives
  - → User defined mappers provide deep copy support for map clauses
  - → Support for reverse offload
  - → Support for task reductions , including on `taskloop` construct, task affinity, new dependence types, depend objects and detachable tasks
  - → Allows `teams` construct outside of `target` construct (i.e., on host)
  - → Supports collapse of non-rectangular loops
  - → Scan extension of reductions
- Major advances for base language normative references
  - → Completed support for Fortran 2003
  - → Added initial support of Fortran 2008, C11, C++11, C++14 and C++17

OpenMP 5.0 clarifications and enhancements

- Supports collapse of imperfectly nested loops
- Supports `!=` on C/C++ loops
- Adds `conditional` modifier to `lastprivate`
- Support use of any C/C++ *lvalue* in `depend` clauses
- Permits `declare target` on C++ classes with virtual members
- Clarification of `declare target` C++ initializations
- Adds `task` modifier on many `reduction` clauses
- Adds `depend` clause to `taskwait` construct

OpenMP 5.1 refines existing functionality

- Adds full support for C11, C++11, C++14, C++17, C++20 and Fortran 2008 and partial support for Fortran 2018
- Extends directive syntax to C++ attribute specifiers
- The `scope` construct supports reductions within parallel regions
  - → Christian discussed this enhancement in another session
- Extends `atomic` construct to support compare-and-swap, min and max
  - → Detailed these enhancements in another session
- Adds many clauses and clause modifiers including:
  - → `nowait` to `taskwait` construct
  - → `strict` modifier to clauses on the `taskloop` construct

OpenMP 5.1 refines existing functionality

- Support for mapping (translated) function pointers
- Device-specific environment variables to control their ICVs
- `nothing` directive supports `metadirective` clarity and completeness
- Several new runtime routines, including more memory allocation flavors
- Deprecations include:
  - → The `master` affinity policy and `master` construct
  - → Cray pointers
  - → Many enum values, most related to OMPT (first-party tool interface)

OpenMP 5.1 adds some significant extensions

- The `interop` construct
  - → Improves native device support (e.g., CUDA streams)
  - → Also supports interoperability with CPU-based libraries (e.g., TBB)
- The new `dispatch` construct, improved `declare variant` directive
  - → Enable use of variants with device-specific arguments
  - → Elision of “unrecognized” code

OpenMP 5.1 adds some significant extensions

- The `assume` directive
  - → Supports optimization hints based on invariants
  - → Supports promise to limit OpenMP usage to (optimizable) subsets
- Loop transformation directives: The `tile` and `unroll` directives
  - → Control use of traditional sequential optimizations
  - → Ensure that they are applied when, where appropriate relative to parallelization

The `error` directive supports user-defined warnings and errors

- Use `error` directive to interact with the compiler

```
#pragma omp error [at(compilation|execution)] [severity(fatal|warning)] \
                  [message(msg-string)]
  structured-block
```

- Compiler displays `msg-string` as part of implementation-defined message
- The `at` clause determines when the effect of the directive occurs
  - → `compilation`: If encountered during compilation in a declarative context (useful along with `metadirective`) or is reachable at runtime
  - → `execution`: If the code location is encountered during execution (similar to `assert()`)
- The `severity` clause determines compiler action
  - → `warning`: Print message only (default)
  - → `fatal`: Stop compilation or execution

The `masked` construct supports filtering execution per thread

- Use `masked` construct to limit parallel execution (low cost: no data environ.)

```
#pragma omp masked [filter(integer-expression)]
  structured-block
```

- Encountering thread executes if `filter` clause matches its thread number
- Default (i.e., no clause) is equivalent to deprecated `master` construct
- Future (i.e., OpenMP 6.0) enhancements planned
  - → Define concept of thread groups, a subset of the threads in a team
  - → Extend `masked` to `filter` based on thread groups or booleans (via clause modifier)
  - → `filter` clause added to other constructs, relying on thread group concept

OpenMP 5.2 improves quality of the specification

- Large portions of specification now generated from JSON-based database
  - → Section headers and directive and clause format
  - → Cross references, index entries, hyperlinks and many other document details
  - → Long-term plan will capture sufficient information in database to generate much more, including grammar, quick reference guide, and header and runtime library routine stub files
- Improves specification of OpenMP syntax
  - → Ensuring syntax of directives and clauses is well-specified and consistent
  - → Ensuring restrictions are consistent and not just implied by syntax
  - → Deprecating one-off syntax choices, many other inconsistencies (12 new deprecation entries)
  - → Makes C++ attribute syntax a first-class citizen
- Many other minor improvements

OpenMP 6.0 includes many major new features

- Free-agent threads significantly change execution model, implementations
- New concept for task dependences: transparent tasks
  - → Enables asynchronous `target data` (also enables other future extensions)
- The `target_data` directive is now a dependence sequence of three task
- Support for dependences and affinity of tasks generated by `taskloop`
- The `taskgraph` directive enables optimized task generation
- User-defined induction and `induction` clause expand parallelism support
- Many significant device support improvements (e.g., `workdistribute`)
- Several additional (sequential) loop transforming directives
- Supported compound constructs are now defined based on a grammar
- Significant improvements to usability and correctness of specification

Induction: Parallelization despite dependences

```
xi = x0;
result = 0.0;
#pragma omp parallel for reduction(+: result) induction(step(x), *: xi)
for (I = 0; I < N; i++) {
  result += c[i] * xi;
  xi *= x;
}
```

- Simple inductions are similar to reductions, particulary with use of `inscan`
  - → Avoids complexity needed to avoid serialization for parallel scan computations
- User-defined induction greatly expands expressible loop parallelism
  - → Can define complex functions to perform computations with dependences
  - → Can use `collector` clause to specify closed form function to enable starting at arbitrary iterations (typically used for start of chunks but can allow arbitrarily)

What is the effect of the following code?

```
// assume in main with initialization omitted
// assume no OpenMP directives omitted

TS = 4096;
#pragma omp taskloop grainsize(TS)
for (i = 0; i < SIZE; i++) {
  A[i] = A[i] * B[i] * s;
}
```

- Pre-6.0 need `parallel masked` directive so multiple threads execute tasks

```
// assume in main with initialization omitted
// assume no OpenMP directives omitted

TS = 4096;
#pragma omp parallel masked
#pragma omp taskloop grainsize(TS)
for (i = 0; i < SIZE; i++) {
  A[i] = A[i] * B[i] * s;
}
```

6.0 evolves execution model significantly

```
// assume in main with initialization omitted
// assume no OpenMP directives omitted

TS = 4096;
#pragma omp taskloop grainsize(TS) threadset(omp_pool)
for (i = 0; i < SIZE; i++) {
  A[i] = A[i] * B[i] * s;
}
```

- OpenMP 6.0 defines OpenMP threads as members of logical thread pool
  - → Pool size can be specified by `OMP_THREAD_LIMIT` environment variable
- OpenMP 6.0 also adds the concept of free-agent threads
  - → Do not need `parallel masked` directive
  - → Instead `threadset` clause can specify that unassigned threads may execute tasks

Task dependences constrain modularity

```
// assume library must ensure fine-grain dependences are honored
int my_func(double *M, double *v) {
  int i, j, k;

  for (i = 0; i < N_ROWS; i += ROWS_PER_TASK) {
    #pragma omp task depend(inout:M[i*N_COLS])
    for (j = 0; j < ROWS_PER_TASK; j++) {
      for (k = 0; k < N_COLS; k++) {
        M[(i+j)*N_COLS + k] = M[(i+j)*N_COLS + k] * v[k]; } } }
  return 0;
}
```

- Successive calls to `my_func` with the same `M` are ordered correctly in OpenMP 5.2 and earlier if they are issued in the same task
  - → Ensures all uses of `task` construct will not deadlock
  - → Other synchronization can alleviate constraint by eliminating concurrency of tasks from different calls so this solution does not provide the desired result

Transparency supports rich dependence graphs

```
// assume my_func as in previous example
double M[N_ROWS*NCOLS], v[NUM_VS][N_COLS];
int i;

// code to initialize M and v omitted for brevity

for (i = 0; i < NUM_VS; i++) {
  #pragma omp task depend(inout:i) transparent(omp_impex)
  my_func(M, &v[i*N_COLS]);
}
```

- The calls to `my_func` are ordered because of the dependence shown
- These tasks are transparent importing and exporting ("`omp_impex`") tasks
  - → Dependences expressed in the calls are now imported and exported
  - → Deadlock freedom is still guaranteed

Extended `parallel` directive to support complete user control of number of threads

- The `parallel` directive will accept a new modifier and two "new" clauses

```
#pragma omp parallel [num_threads(prescriptiveness: nthreads)] \
                     [severity(fatal|warning)][message(msg-string)]
  structured-block
```

- Using `strict` *prescriptiveness* requires `nthreads` to be provided
- Clauses, previously available on `error` directive, effective with `strict` if cannot provide `nthreads`
  - → Display `msg-string` as part of implementation-defined message
  - → If `severity` is `fatal` execution is terminated
  - → If `severity` is `warning` message is displayed but execution continues
- Also now allowed to provide a list for `nthreads` to support nested parallelism