Programming OpenMP

Christian Terboven

Michael Klemm

Agenda (in total 4 days)

- Day 1: OpenMP Introduction
- **Day 2: Tasking & Optimizations for NUMA**
  - → Welcome
  - → Tasking Motivation
  - → Tasking Model
  - → Taskloop
  - → Task Dependencies
  - → Cut-off
  - → NUMA
  - → Task Affinity
  - → **Hands-On**
- Day 3: Introduction to Offloading with OpenMP
- Day 4: Advanced Offloading Topics

Material

https://github.com/cterboven/OpenMP-tutorial-CSC

Programming OpenMP

Tasking Introduction

Christian Terboven

Michael Klemm

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

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

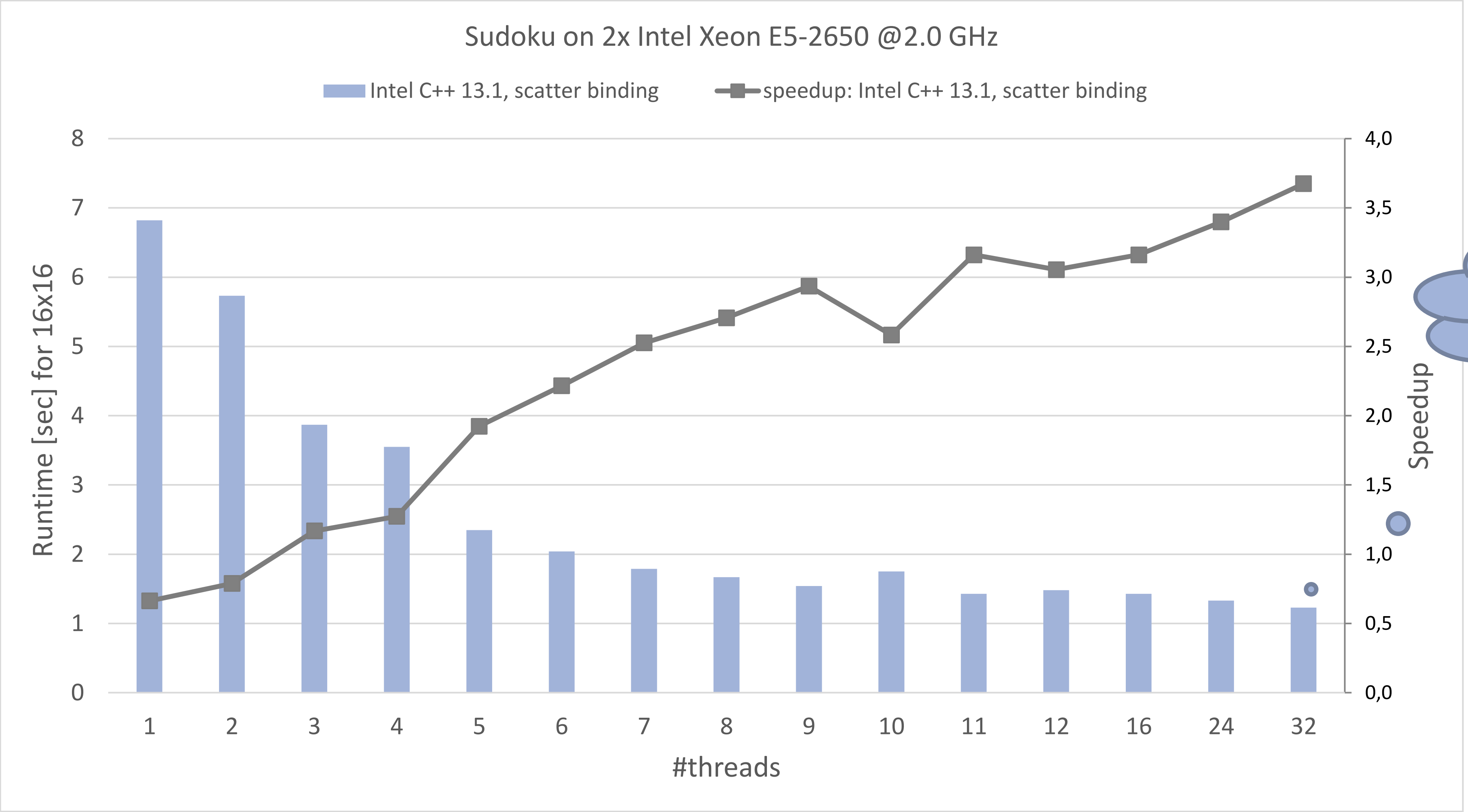
- Wait for completion

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

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

`#pragma omp taskwait`
wait for all child tasks

Performance Evaluation



Is this the best we can 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> ) {
    ...
}
```

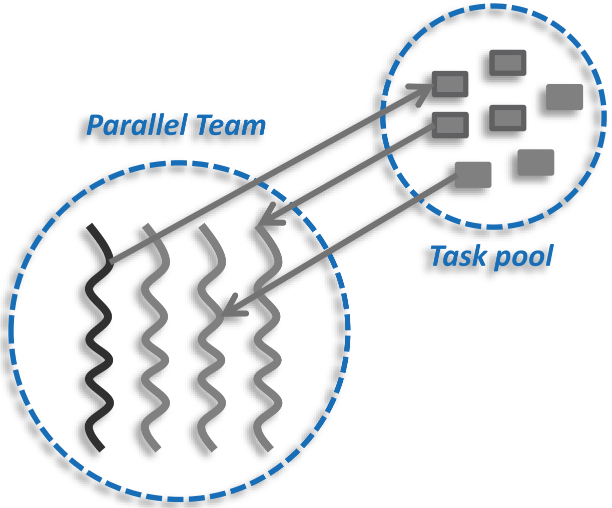
  - → recursive functions

```
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:

Data Environment

- → private(list)
- → firstprivate(list)
- → shared(list)
- → default(shared | none)
- → in_reduction(r-id: list)

Miscellaneous

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

Cutoff Strategies

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

Synchronization

- → depend(dep-type: list)

Task Scheduling

- → untied
- → priority(priority-value)
- → affinity(list)

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 ☹

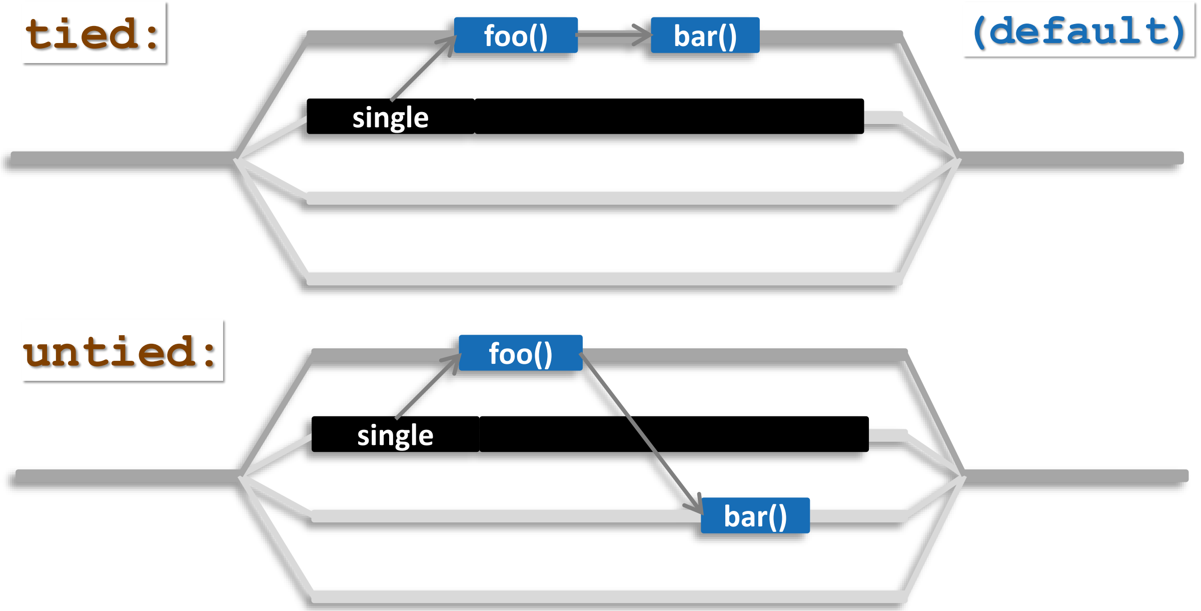
Task scheduling: taskyield directive

- 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()
   }
}
```



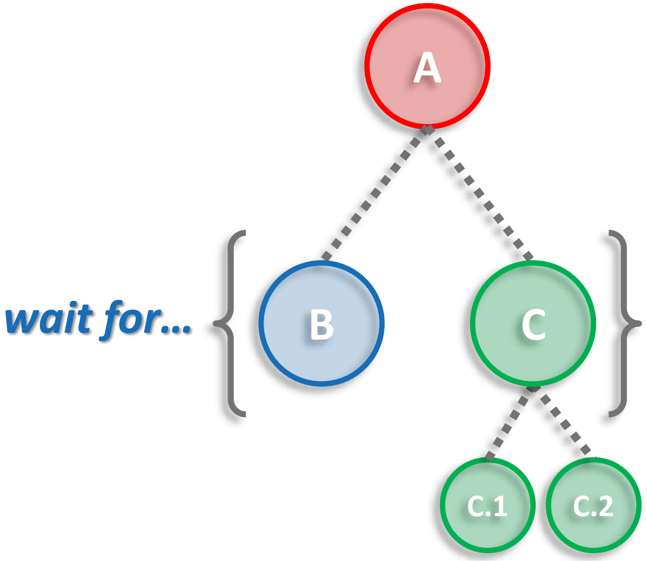
Task synchronization: taskwait directive

- 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…

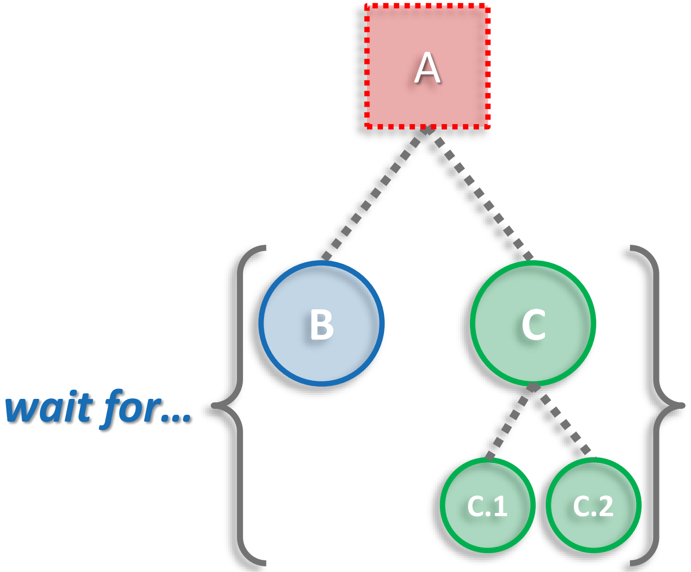
Task synchronization: taskgroup construct

- 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)…

5

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

4

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

1

```
int A[SIZE];
#pragma omp threadprivate(A)

// ...
#pragma omp task
{
  // A: threadprivate
}
```

2

```
int *p;

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

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

3

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

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

Implicit data-sharing attributes (in-practice)

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

```
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 (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

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

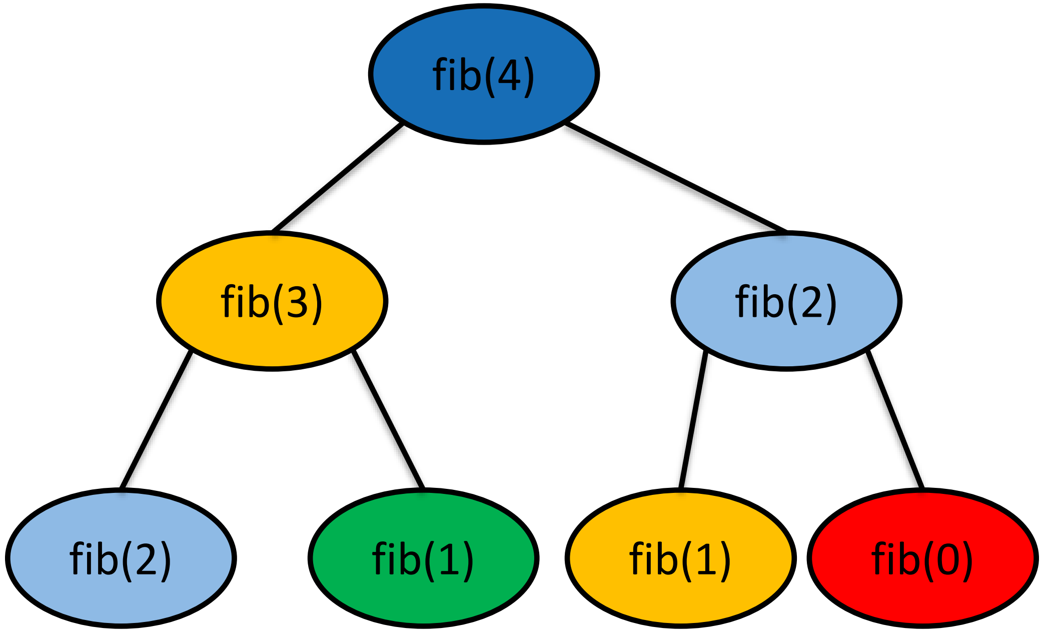
Fibonacci illustrated

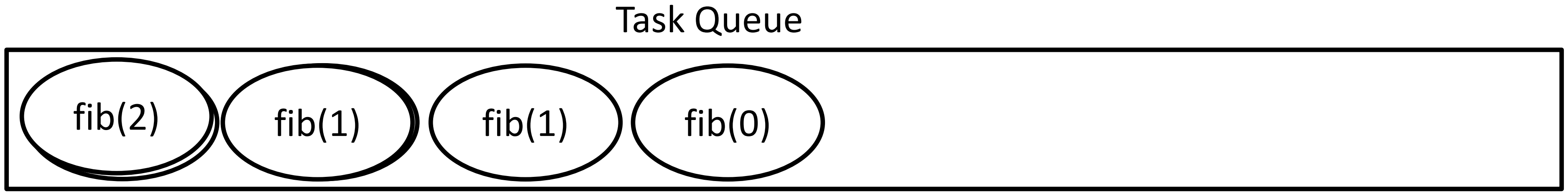
```c
int main(int argc,
         char* argv[])
{
    [...]
    #pragma omp parallel
    {
        #pragma omp single
       {
           fib(input);
       }
    }
    [...]
}
```

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

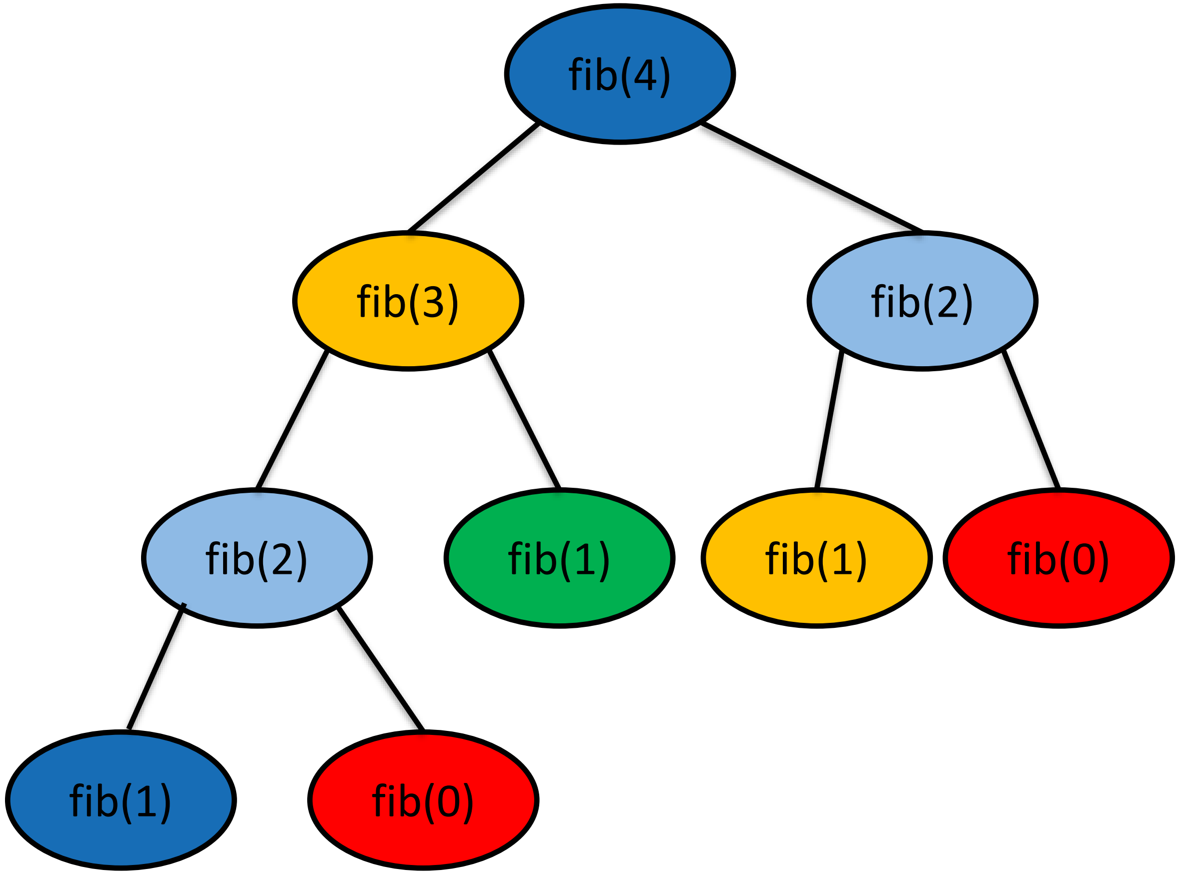
- Only one Task / Thread enters fib() from main(), it is responsible for creating the two initial work tasks
- Taskwait is required, as otherwise x and y would get lost

- 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
- …



Programming OpenMP

Tasking: taskloop and dependences

Christian Terboven

Michael Klemm

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

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

Improving Tasking Performance:

Task dependences

Motivation

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

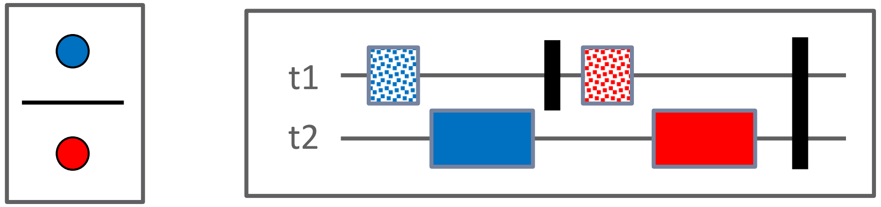
OpenMP 3.1

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

  #pragma omp taskwait

● #pragma omp task
  x++;
}
```

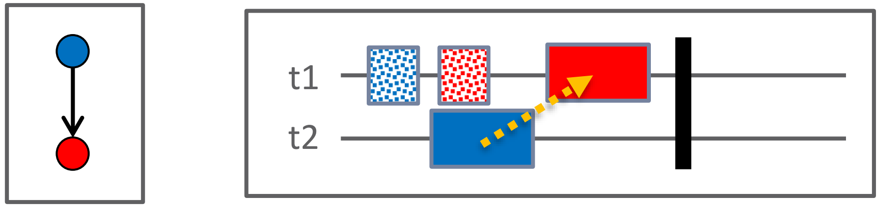
OpenMP 4.0

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



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

OpenMP 3.1



OpenMP 4.0



▨ Task's creation time

■ Task's execution time

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++;
}
```

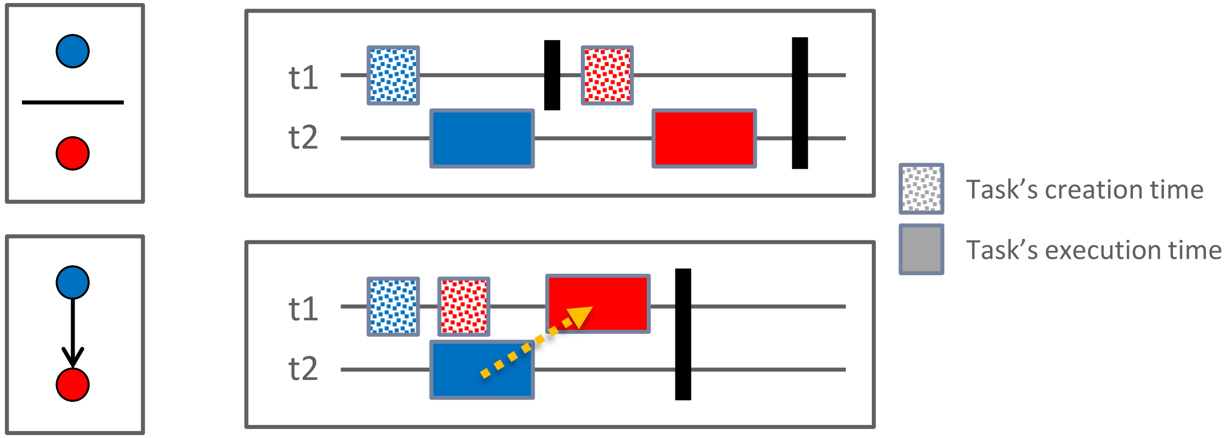
OpenMP 4.0

```
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 parallelism!!!!

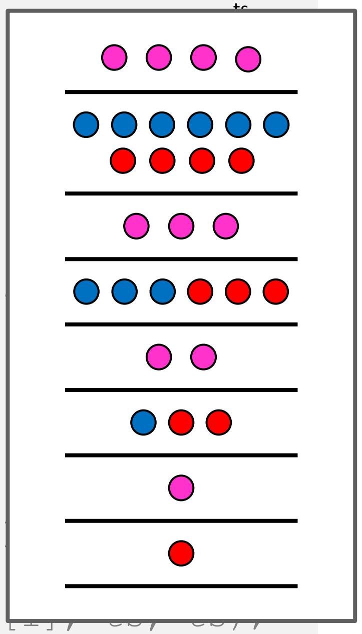
OpenMP 3.1

OpenMP 4.0



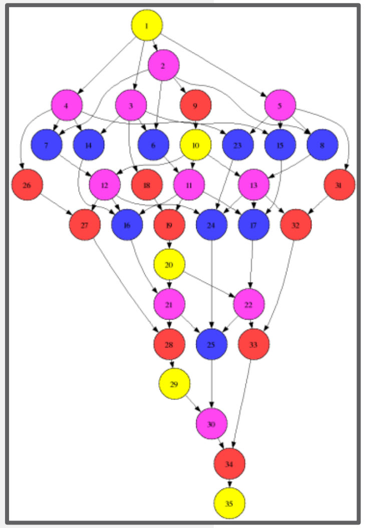
Motivation: Cholesky factorization

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

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

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



OpenMP 3.1

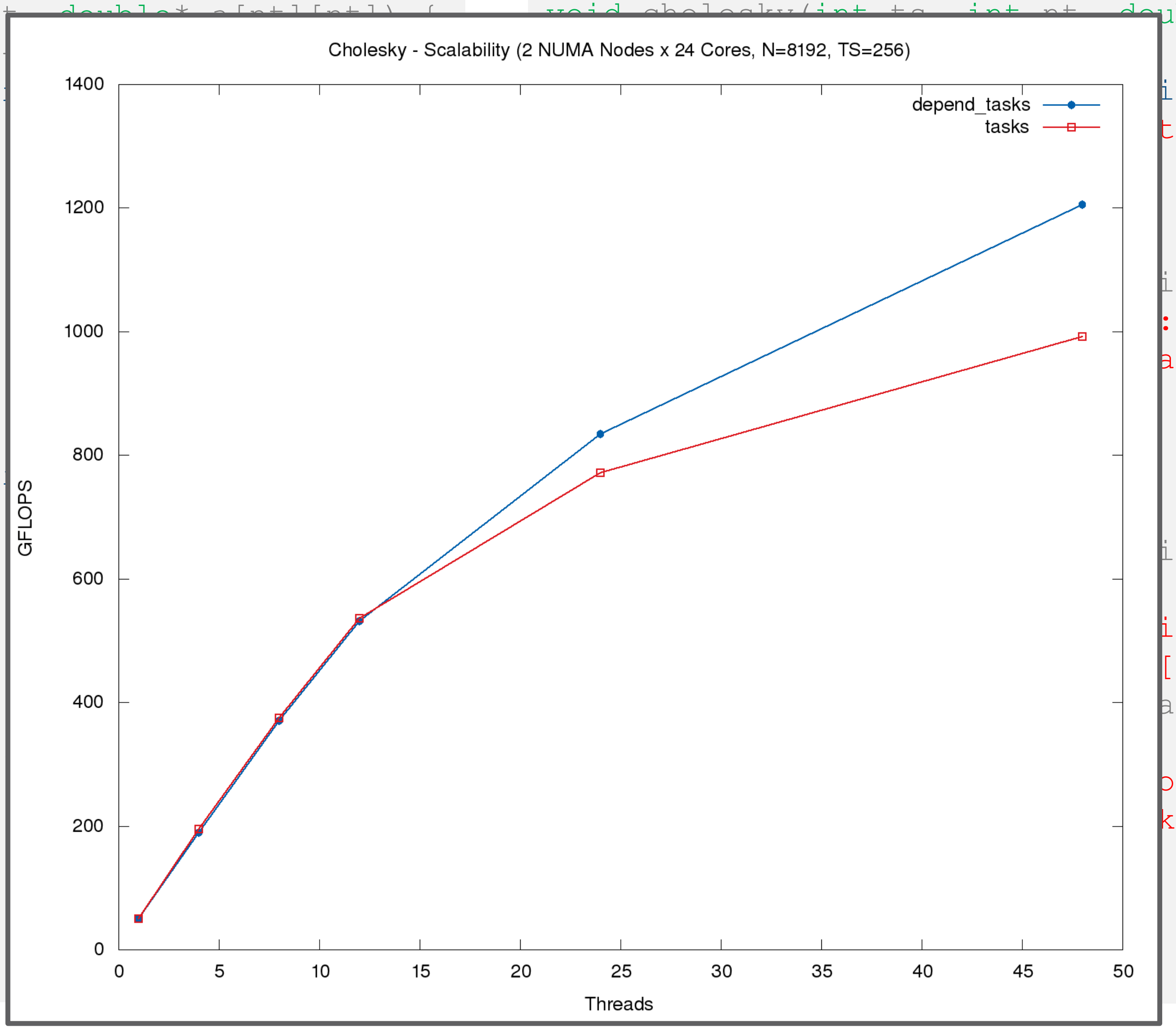
```
void cholesky(int ts, int nt, double* a[nt][nt]) {
  for (int k = 0; k < nt; k++) {
    // Diagonal Block factorization
    #pragma omp task depend(inout: a[k][k])
 ●  potrf(a[k][k], ts, ts);

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



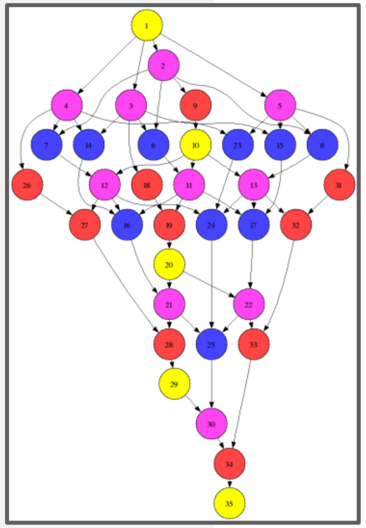
OpenMP 4.0

Motivation: Cholesky factorization

```
void cholesky(int ts, int nt
  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` 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

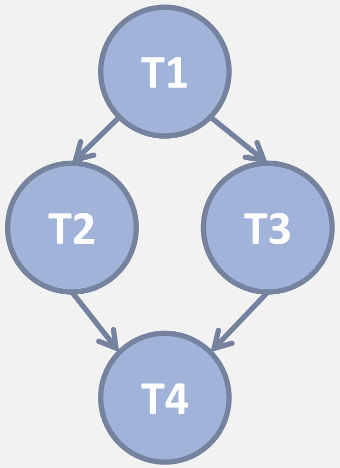
What’s in the spec: depend clause (1)

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

- If a task defir
  - → the task will c
    ne of the list items in
    an out or in

- If a task defir
  - → the task will c
    ne of the list items in
    an in, out c

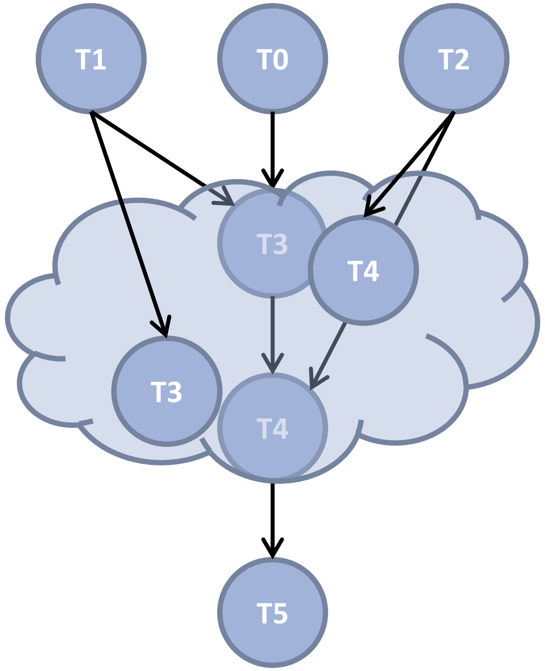
```
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(inout: x) //T1
  { ... }

  #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: depend clause (2)

- New dependency type: `mutexinoutset`

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

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

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

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

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

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



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

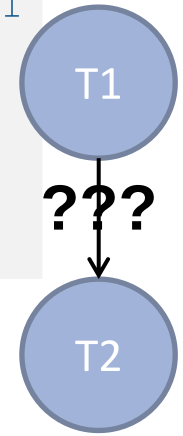
What’s in the spec: depend clause (4)

- 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);
}
```



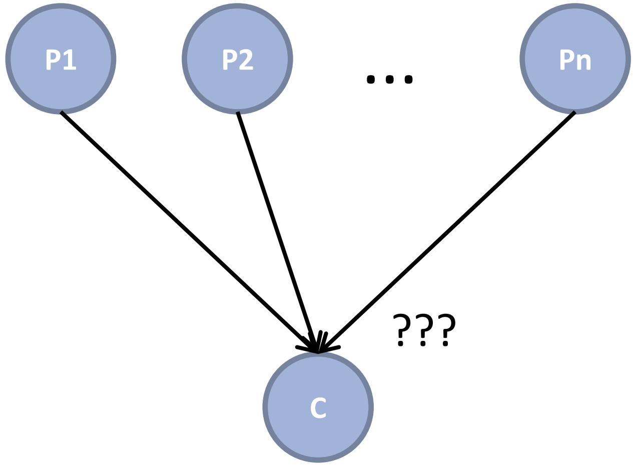
What’s in the spec: depend clause (5)

- 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])`



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
i
#
#
{
}
```

```
//test1_v3.cc
i
#
#
{
}
```

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

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

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

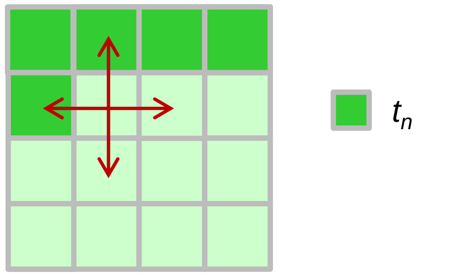
Use case

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:

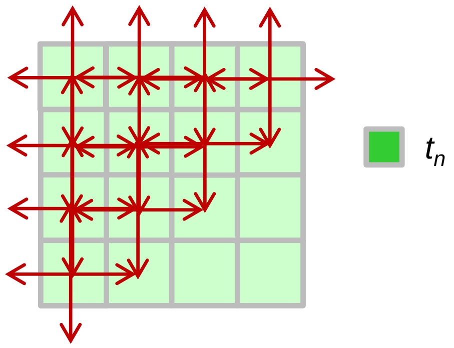
- two cells (north & west) that are computed in the current time step, and
- two cells (south & east) that were computed in the previous time step

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

For an specific t



We can exploit the wavefront to obtain parallelism!!

Use case : Gauss-seidel (3)

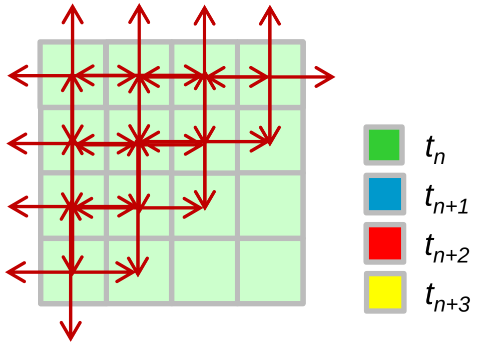
```
void gauss_seidel(int tsteps, int size, int TS, int (*p)[size]) {
  int NB = size / TS;
  #pragma omp parallel
  for (int t = 0; t < tsteps; ++t) {
    // First NB diagonals
    for (int diag = 0; diag < NB; ++diag) {
      #pragma omp for
      for (int d = 0; d <= diag; ++d) {
        int ii = 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
      }
    }
  }
}
```

2nd parallelization strategy

multiple time iterations

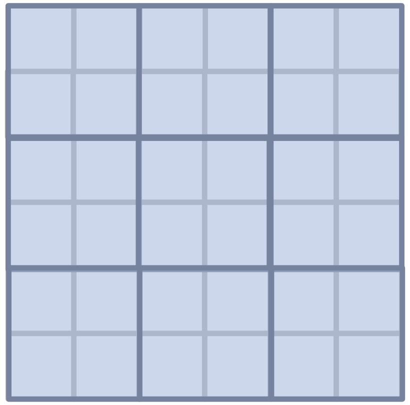


We can exploit the wavefront of multiple time steps to obtain MORE parallelism!!

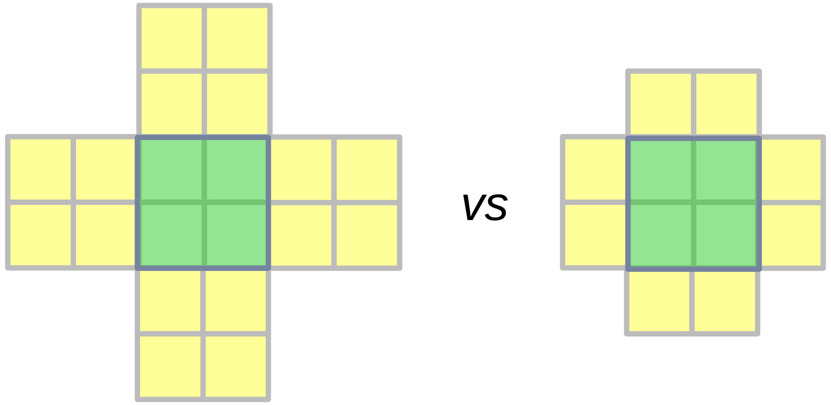
Use case : Gauss-seidel (5)

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

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

inner matrix region

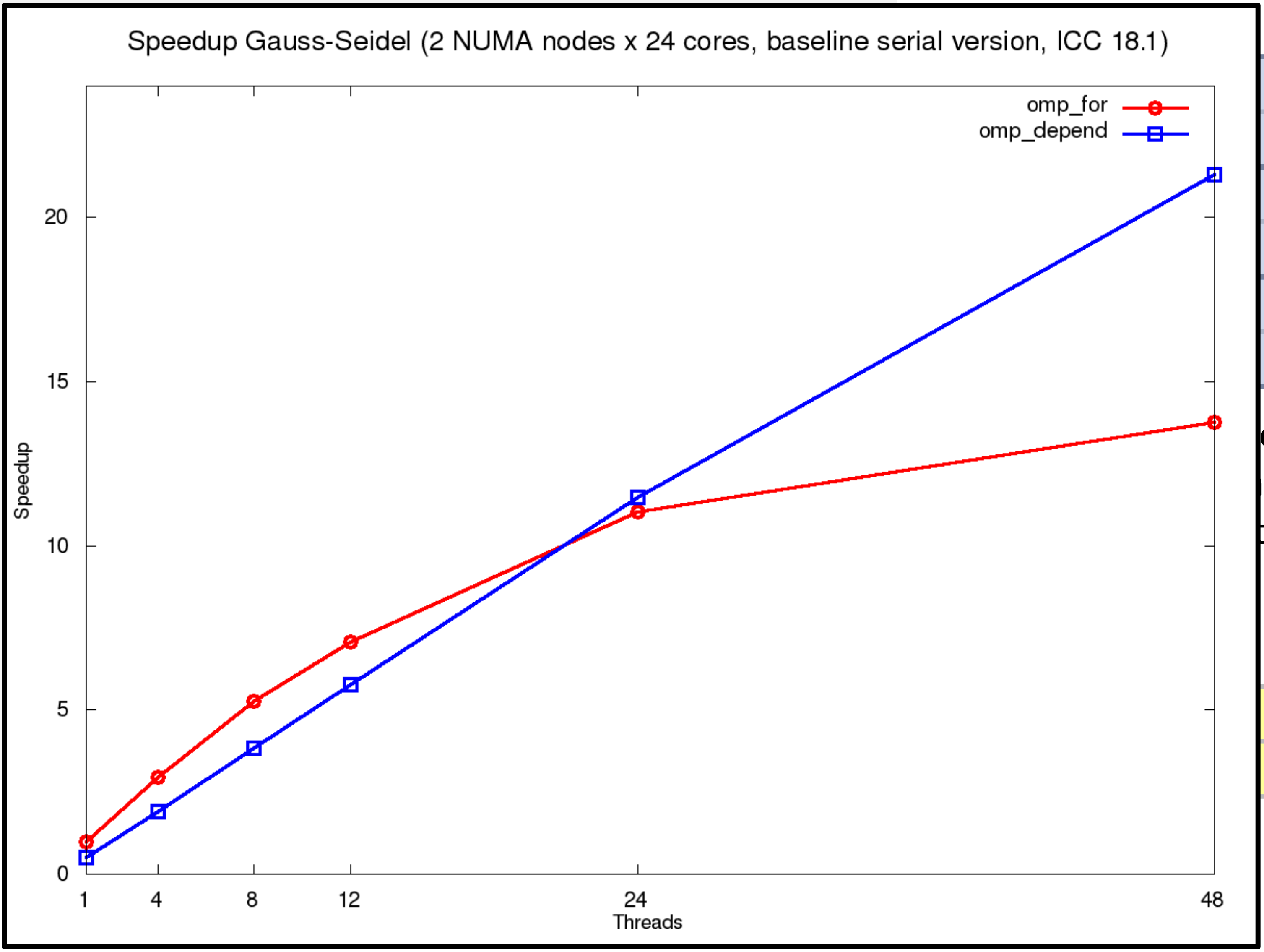


Q: Why do the input dependences depend on the whole block rather than just a column/row?

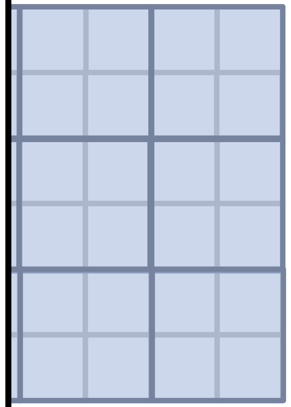


Use case : Gauss-seidel (5)

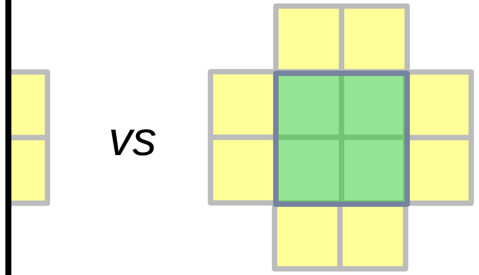
```
void gauss_seidel(i
  int NB = size / T

  #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



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



OpenMP 5.0: (even) more advanced features

Advanced features: deps on `taskwait`

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

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

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

  #pragma omp taskwait depend(in: x)

  std::cout << x << std::endl;
}
```

Programming OpenMP

Cut-off strategies

Christian Terboven

Michael Klemm

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

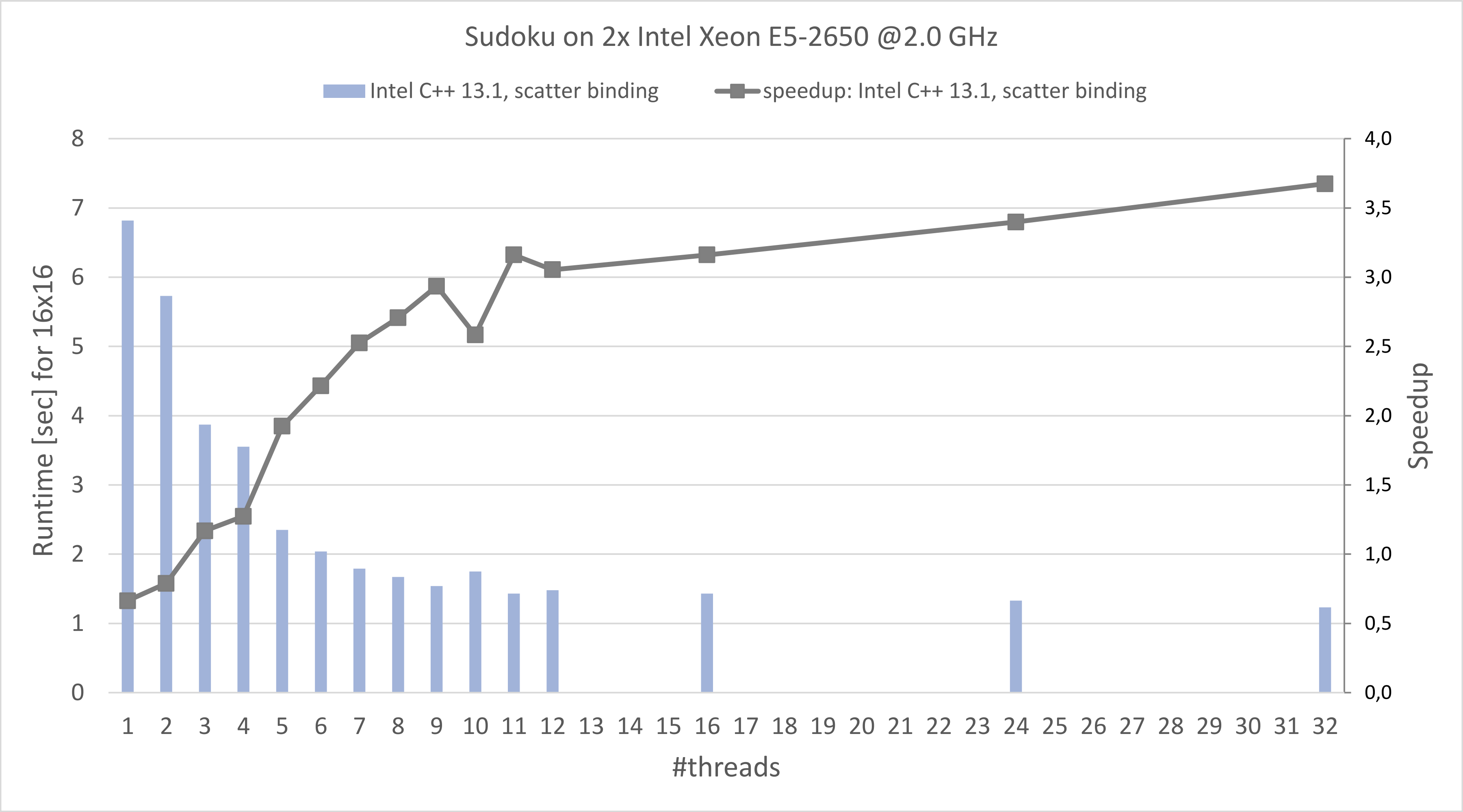
- (2) Try all numbers:
  - (2 a) Check Sudoku
    - If invalid: skip
    - If valid: Go to next field

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

- Wait for completion

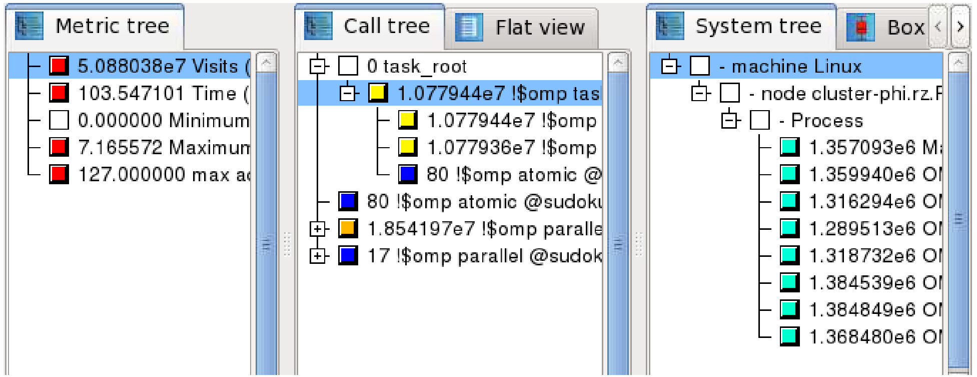
`#pragma omp taskwait`
wait for all child tasks

Performance Evaluation



Performance Analysis

Event-based profiling provides a good overview :



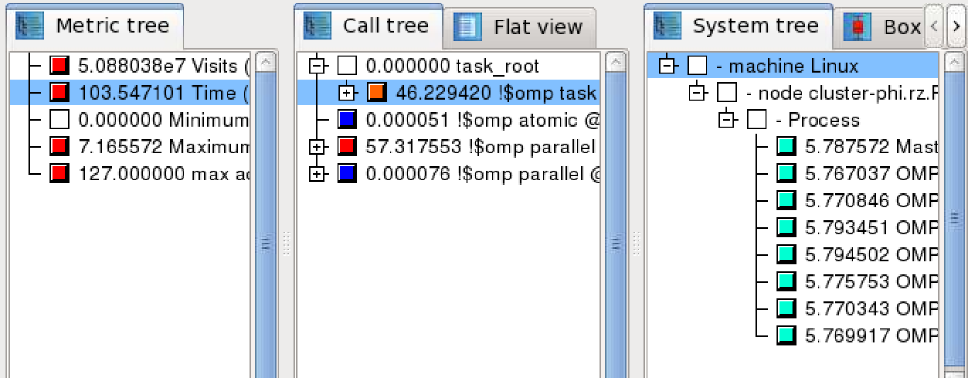
Metric tree | Call tree | Flat view | System tree | Box

- 🟥 5.088038e7 Visits (
- 🟥 103.547101 Time (
- □ 0.000000 Minimum
- 🟥 7.165572 Maximum
- 🟥 127.000000 max ac

- □ 0 task_root
  - 🟨 1.077944e7 !$omp tas
    - 🟨 1.077944e7 !$omp
    - 🟨 1.077936e7 !$omp
    - 🟦 80 !$omp atomic @
  - 🟦 80 !$omp atomic @sudoku
  - 🟧 1.854197e7 !$omp paralle
  - 🟦 17 !$omp parallel @sudok

- □ - machine Linux
  - □ - node cluster-phi.rz.F
    - □ - Process
      - 🟩 1.357093e6 Ma
      - 🟩 1.359940e6 OI
      - 🟩 1.316294e6 OI
      - 🟩 1.289513e6 OI
      - 🟩 1.318732e6 OI
      - 🟩 1.384539e6 OI
      - 🟩 1.384849e6 OI
      - 🟩 1.368480e6 OI

Every thread is executing ~1.3m tasks…



Metric tree | Call tree | Flat view | System tree | Box

- 🟥 5.088038e7 Visits
- 🟥 103.547101 Time (
- □ 0.000000 Minimum
- 🟥 7.165572 Maximum
- 🟥 127.000000 max ac

- □ 0.000000 task_root
  - 🟧 46.229420 !$omp task
  - 🟦 0.000051 !$omp atomic @
  - 🟥 57.317553 !$omp parallel
  - 🟦 0.000076 !$omp parallel @

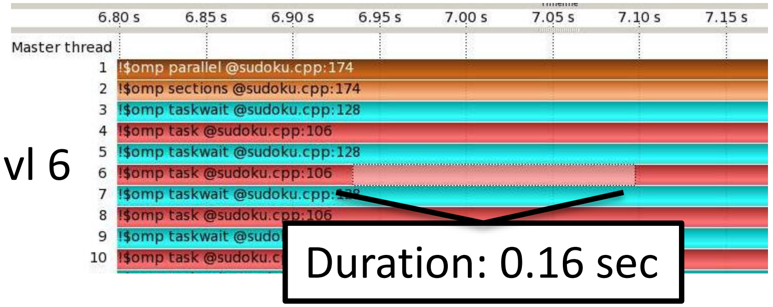
- □ - machine Linux
  - □ - node cluster-phi.rz.F
    - □ - Process
      - 🟩 5.787572 Mast
      - 🟩 5.767037 OMP
      - 🟩 5.770846 OMP
      - 🟩 5.793451 OMP
      - 🟩 5.794502 OMP
      - 🟩 5.775753 OMP
      - 🟩 5.770343 OMP
      - 🟩 5.769917 OMP

… in ~5.7 seconds.

=> average duration of a task is ~4.4 µs

Tracing provides more details:

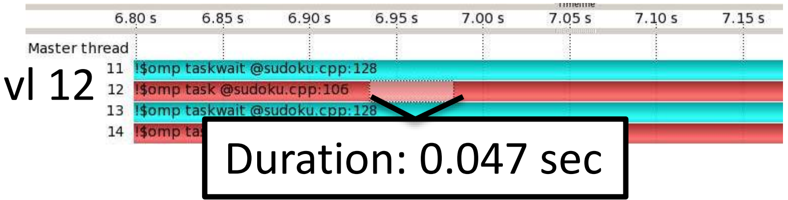
lvl 6



1 !$omp parallel @sudoku.cpp:174
2 !$omp sections @sudoku.cpp:174
3 !$omp taskwait @sudoku.cpp:128
4 !$omp task @sudoku.cpp:106
5 !$omp taskwait @sudoku.cpp:128
6 !$omp task @sudoku.cpp:106
7 !$omp taskwait @sudoku.cpp:128
8 !$omp task @sudoku.cpp:106
9 !$omp taskwait @sudo
10 !$omp task @sudoku.c

Duration: 0.16 sec

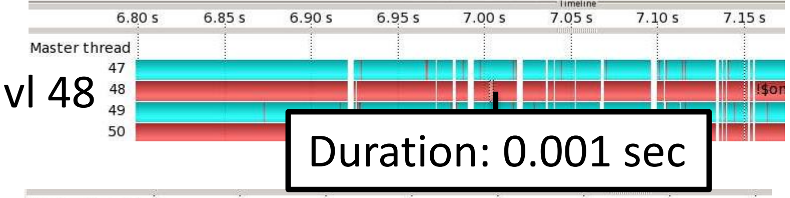
lvl 12



11 !$omp taskwait @sudoku.cpp:128
12 !$omp task @sudoku.cpp:106
13 !$omp taskwait @sudoku.cpp:128
14 !$omp ta

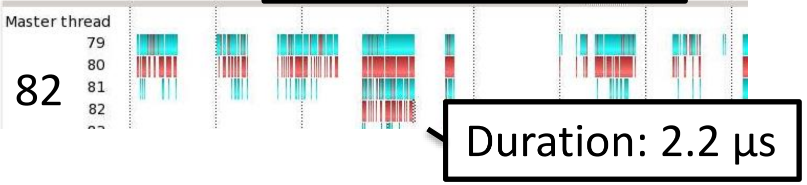
Duration: 0.047 sec

lvl 48



Duration: 0.001 sec

lvl 82

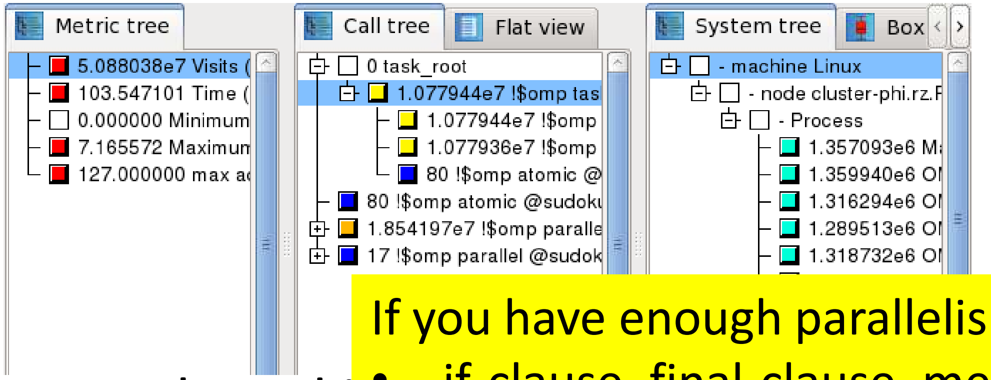


Duration: 2.2 µs

Tasks get much smaller down the call-stack.

Performance Analysis

Event-based profiling provides a good overview :



Metric tree

- 5.088038e7 Visits (
- 103.547101 Time (
- 0.000000 Minimum
- 7.165572 Maximum
- 127.000000 max ac

Call tree | Flat view

- 0 task_root
  - 1.077944e7 !$omp tas
    - 1.077944e7 !$omp
    - 1.077936e7 !$omp
    - 80 !$omp atomic @
  - 80 !$omp atomic @sudoku
  - 1.854197e7 !$omp paralle
  - 17 !$omp parallel @sudok

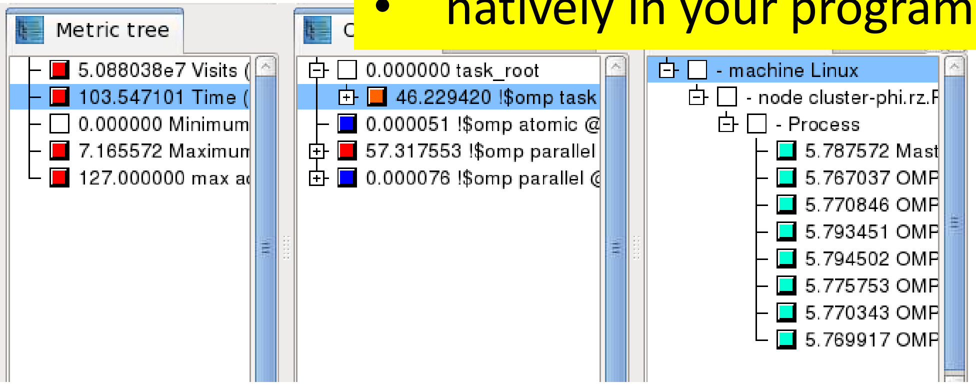
System tree | Box

- machine Linux
  - node cluster-phi.rz.F
    - Process
      - 1.357093e6 Ma
      - 1.359940e6 OI
      - 1.316294e6 OI
      - 1.289513e6 OI
      - 1.318732e6 OI

Every thread i

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

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



Metric tree

- 5.088038e7 Visits (
- 103.547101 Time (
- 0.000000 Minimum
- 7.165572 Maximum
- 127.000000 max ac

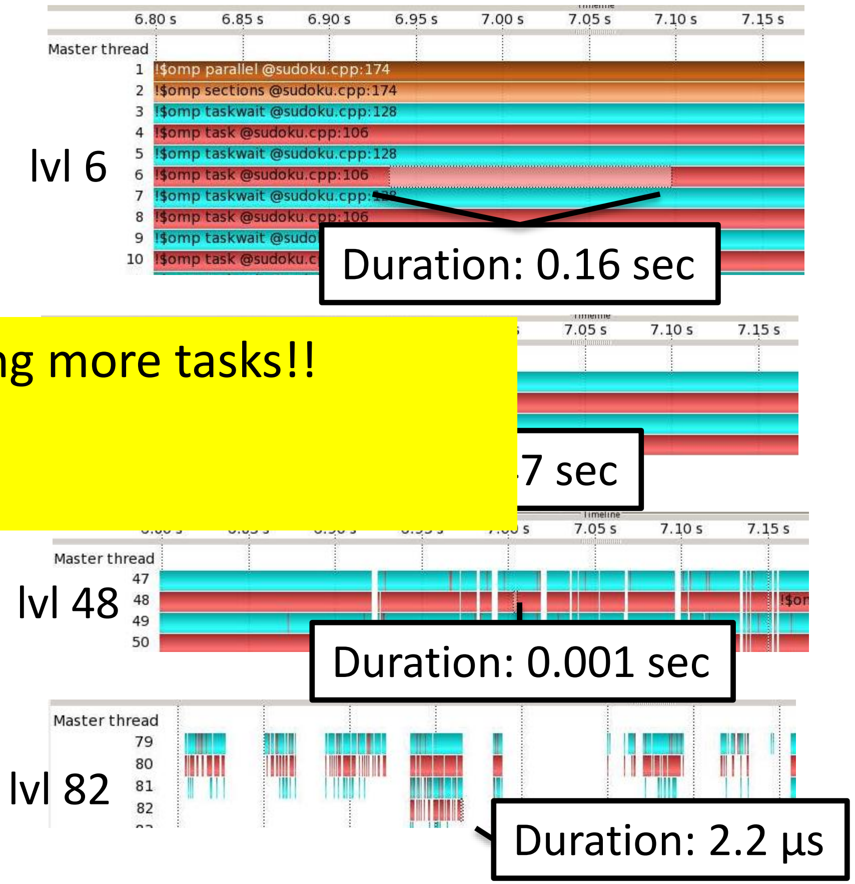
C

- 0.000000 task_root
  - 46.229420 !$omp task
  - 0.000051 !$omp atomic @
  - 57.317553 !$omp parallel
  - 0.000076 !$omp parallel @

- machine Linux
  - node cluster-phi.rz.F
    - Process
      - 5.787572 Mast
      - 5.767037 OMP
      - 5.770846 OMP
      - 5.793451 OMP
      - 5.794502 OMP
      - 5.775753 OMP
      - 5.770343 OMP
      - 5.769917 OMP

... in ~5.7 seconds.
=> average duration of a task is ~4.4 µs

Tracing provides more details:



Duration: 0.16 sec

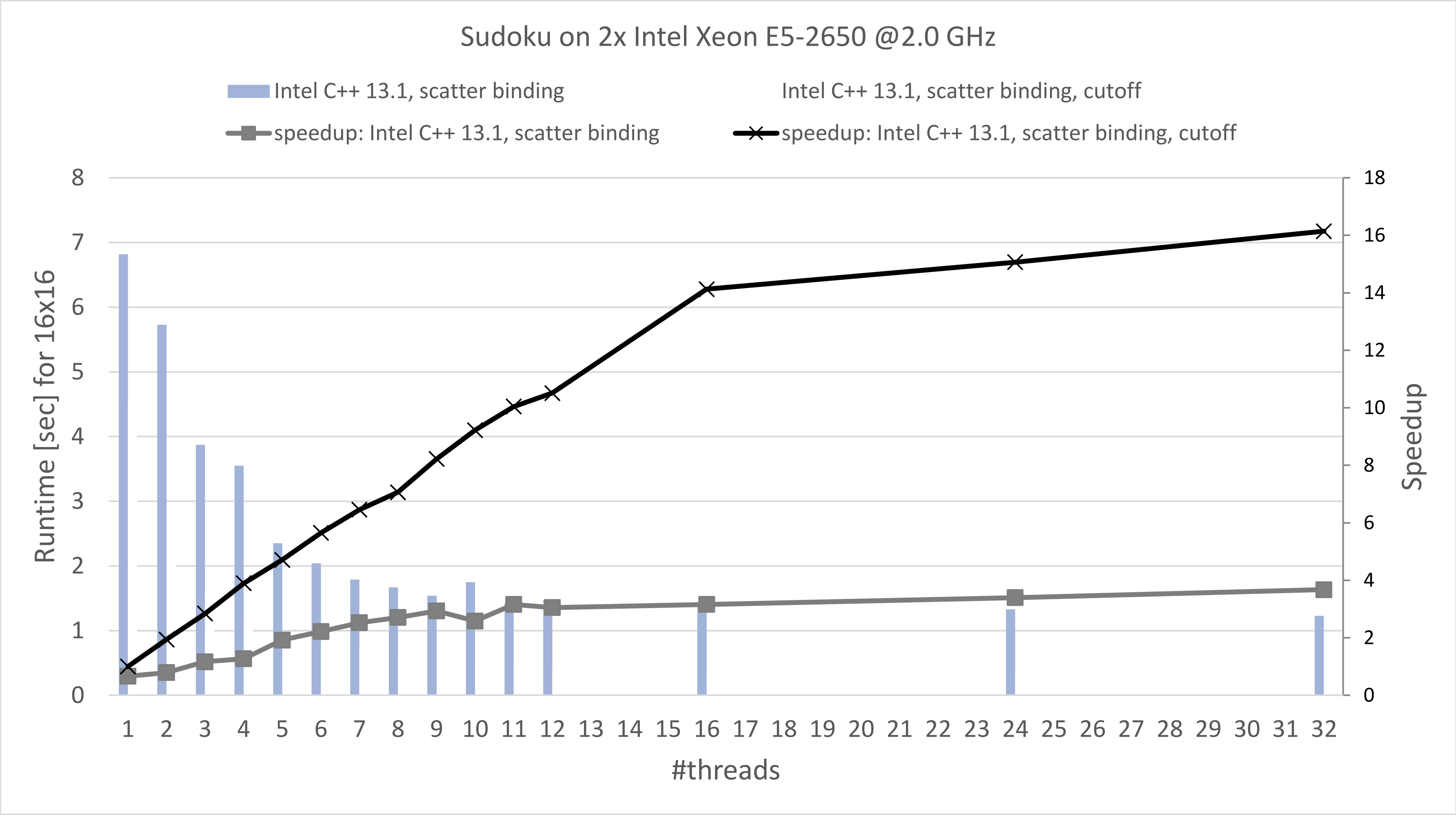
7 sec

Duration: 0.001 sec

Duration: 2.2 µs

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”);
}
```

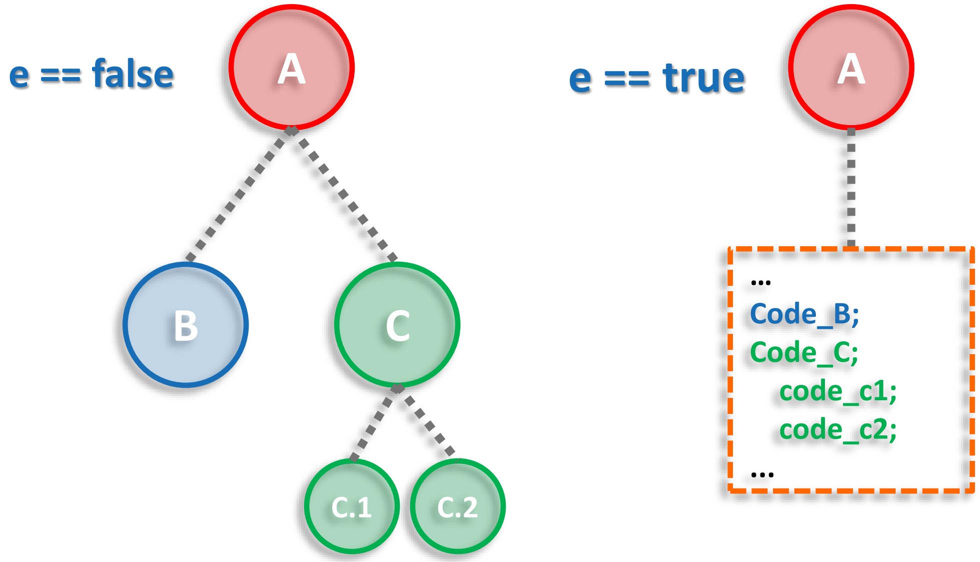
Really useful to debug tasking applications!

- 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 advantage of `final` neither `mergeable` =(

Programming OpenMP

NUMA

Christian Terboven

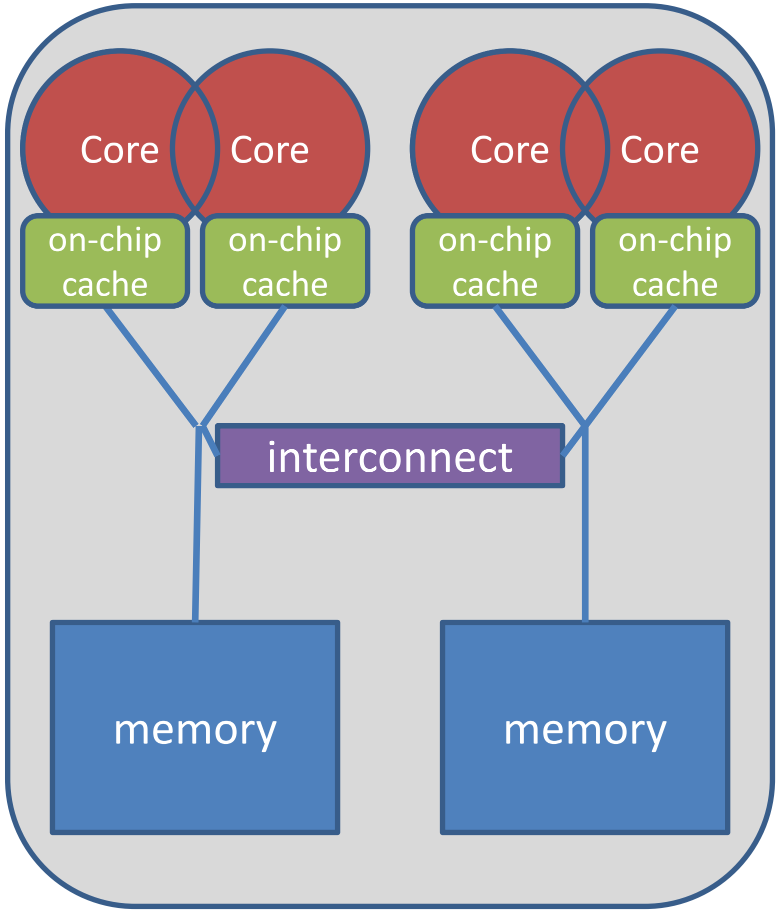
Michael Klemm

OpenMP: Memory Access

Non-uniform Memory

How To Distribute The Data ?

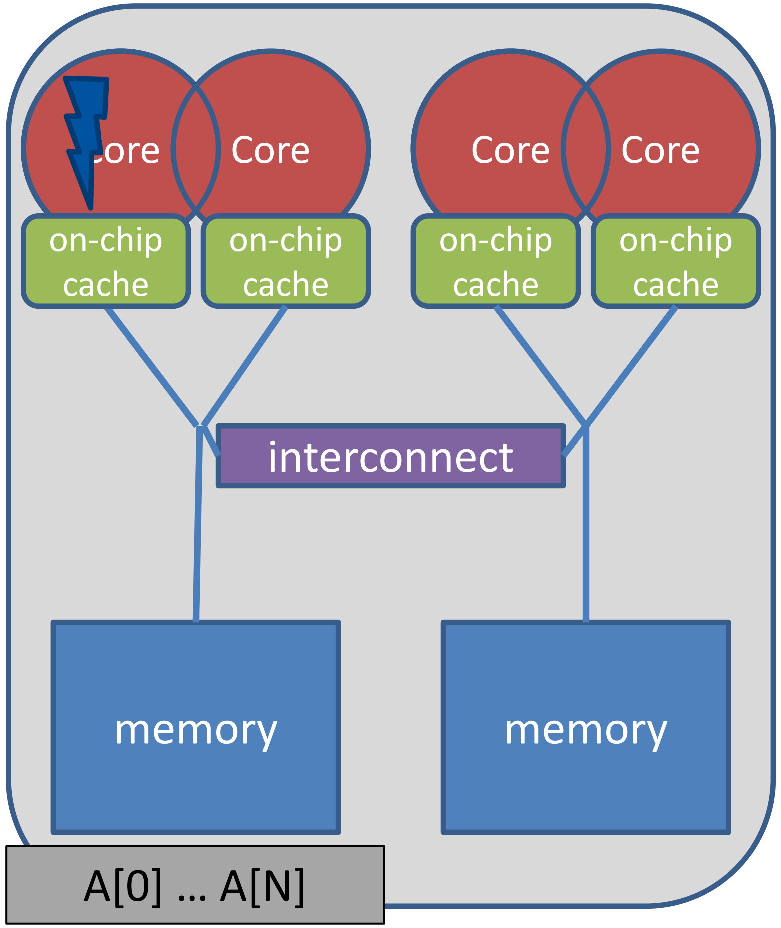
```
double* A;
A = (double*)
    malloc(N * sizeof(double));



for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```



Non-uniform Memory

- **Serial code: all array elements are allocated in the memory of the NUMA node closest to the core executing the initializer thread (first touch)**

```
double* A;
A = (double*)
    malloc(N * sizeof(double));



for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```



About Data Distribution

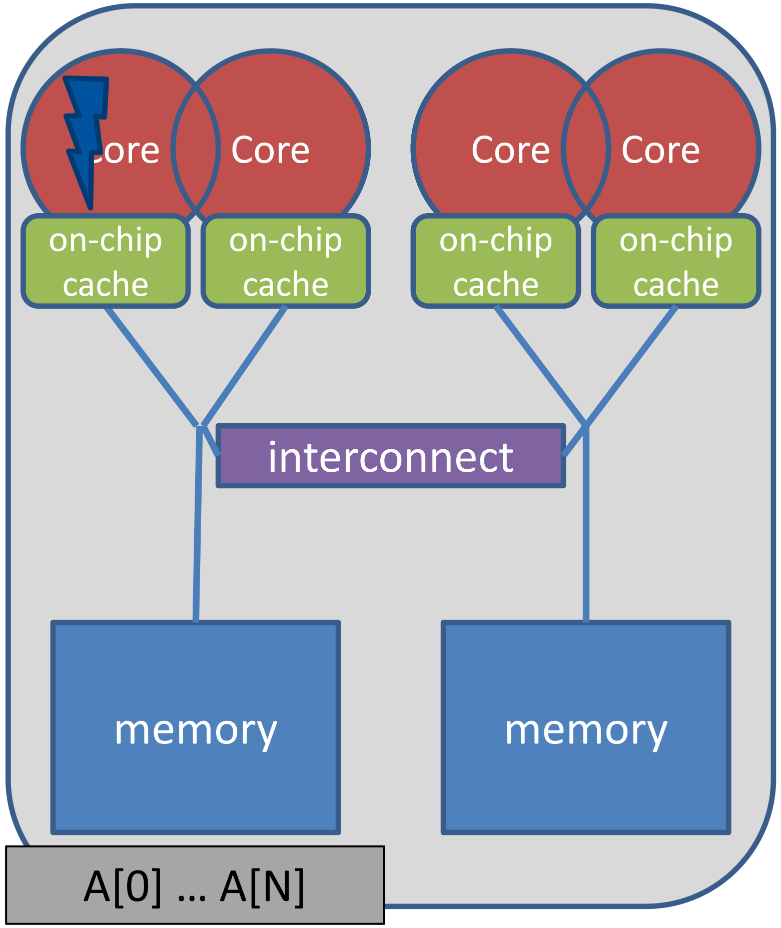
- Important aspect on cc-NUMA systems
  - →If not optimal, longer memory access times and hotspots
- Placement comes from the Operating System
  - →This is therefore Operating System dependent
- Windows, Linux and Solaris all use the “First Touch” placement policy by default
  - →May be possible to override default (check the docs)

Non-uniform Memory

- **Serial code: all array elements are allocated in the memory of the NUMA node closest to the core executing the initializer thread (first touch)**

```
double* A;
A = (double*)
    malloc(N * sizeof(double));
```

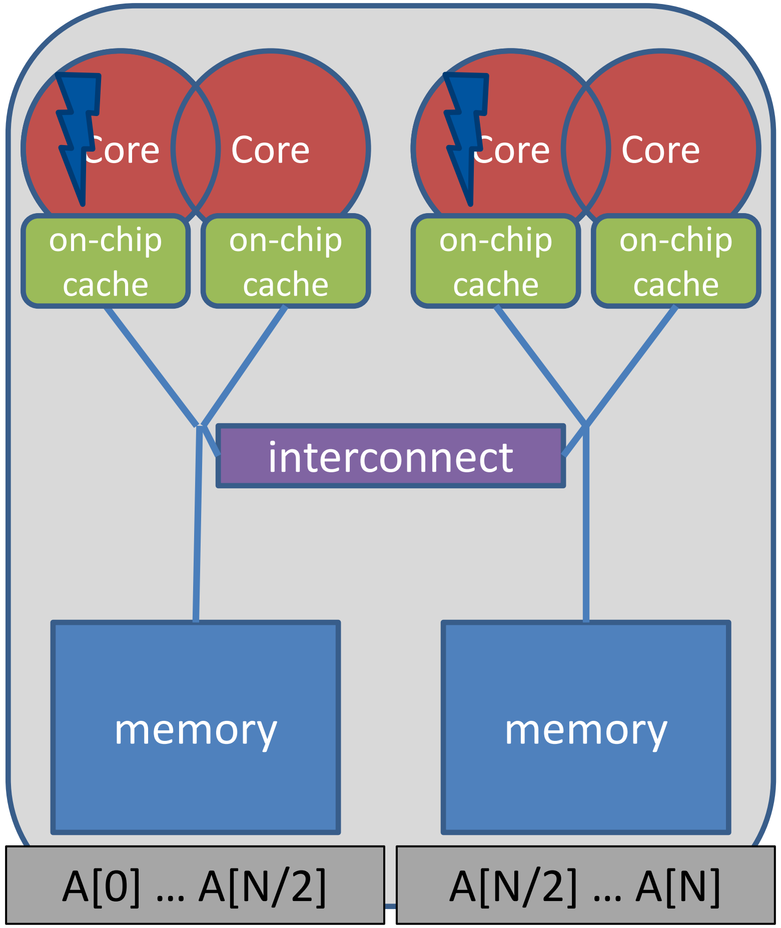
```
for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```



First Touch Memory Placement

- First Touch w/ parallel code: all array elements are allocated in the memory of the NUMA node that contains the core that executes the thread that initializes the partition

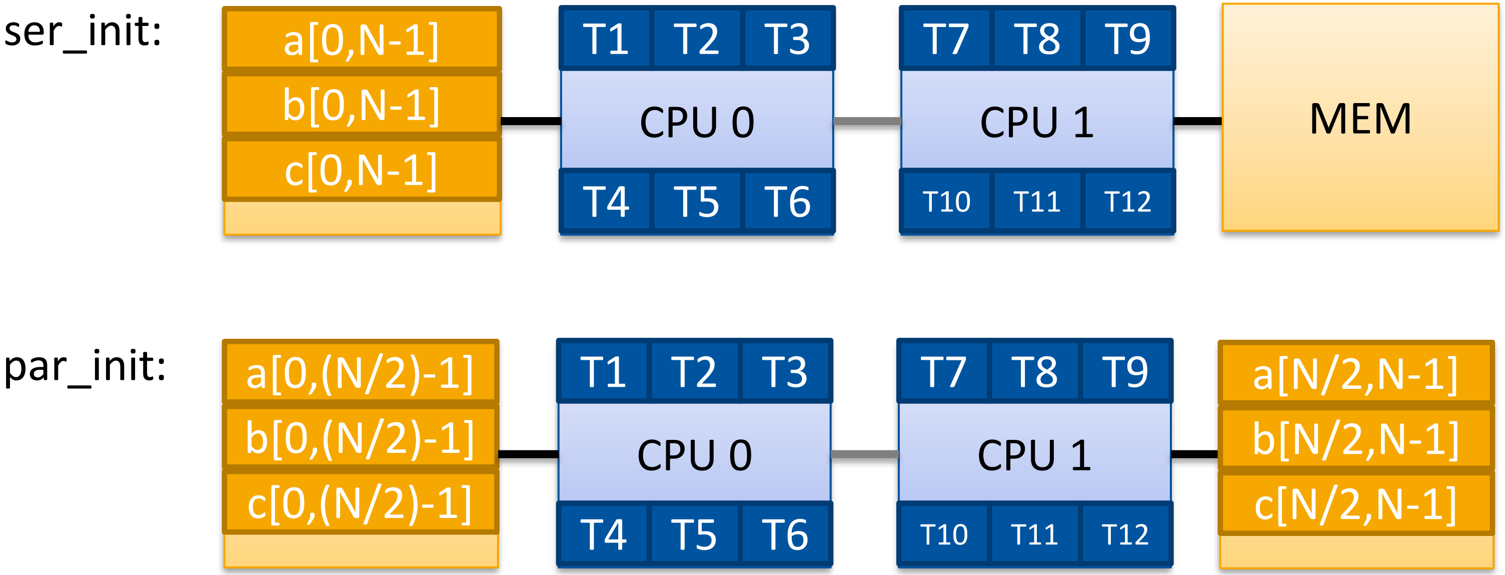
```
double* A;
A = (double*)
    malloc(N * sizeof(double));

omp_set_num_threads(2);

#pragma omp parallel for
for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```



Serial vs. Parallel Initialization

- Stream example on 2 socket sytem with Xeon X5675 processors, 12 OpenMP threads:

	copy	scale	add	triad
ser_init	18.8 GB/s	18.5 GB/s	18.1 GB/s	18.2 GB/s
par_init	41.3 GB/s	39.3 GB/s	40.3 GB/s	40.4 GB/s



Get Info on the System Topology

- Before you design a strategy for thread binding, you should have a basic understanding of the system topology. Please use one of the following options on a target machine:
  - → Intel MPI‘s `cpuinfo` tool
    - → `cpuinfo`
    - → Delivers information about the number of sockets (= packages) and the mapping of processor ids to cpu cores that the OS uses.
  - → hwlocs‘ `hwloc-ls` tool
    - → `hwloc-ls`
    - → Displays a graphical representation of the system topology, separated into NUMA nodes, along with the mapping of processor ids to cpu cores that the OS uses and additional info on caches.

Decide for Binding Strategy

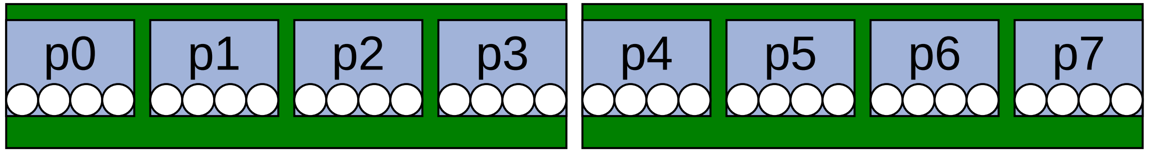
- Selecting the „right“ binding strategy depends not only on the topology, but also on application characteristics.
  - →Putting threads far apart, i.e., on different sockets
    - →May improve aggregated memory bandwidth available to application
    - →May improve the combined cache size available to your application
    - →May decrease performance of synchronization constructs
  - →Putting threads close together, i.e., on two adjacent cores that possibly share some caches
    - →May improve performance of synchronization constructs
    - →May decrease the available memory bandwidth and cache size

Places + Binding Policies (1/2)

- Define OpenMP Places
  - → set of OpenMP threads running on one or more processors
  - → can be defined by the user, i.e. `OMP_PLACES=cores`
- Define a set of OpenMP Thread Affinity Policies
  - → SPREAD: spread OpenMP threads evenly among the places, partition the place list
  - → CLOSE: pack OpenMP threads near master thread
  - → MASTER: collocate OpenMP thread with master thread
- Goals
  - → user has a way to specify where to execute OpenMP threads
  - → locality between OpenMP threads / less false sharing / memory bandwidth

Places

- Assume the following machine:



  → 2 sockets, 4 cores per socket, 4 hyper-threads per core

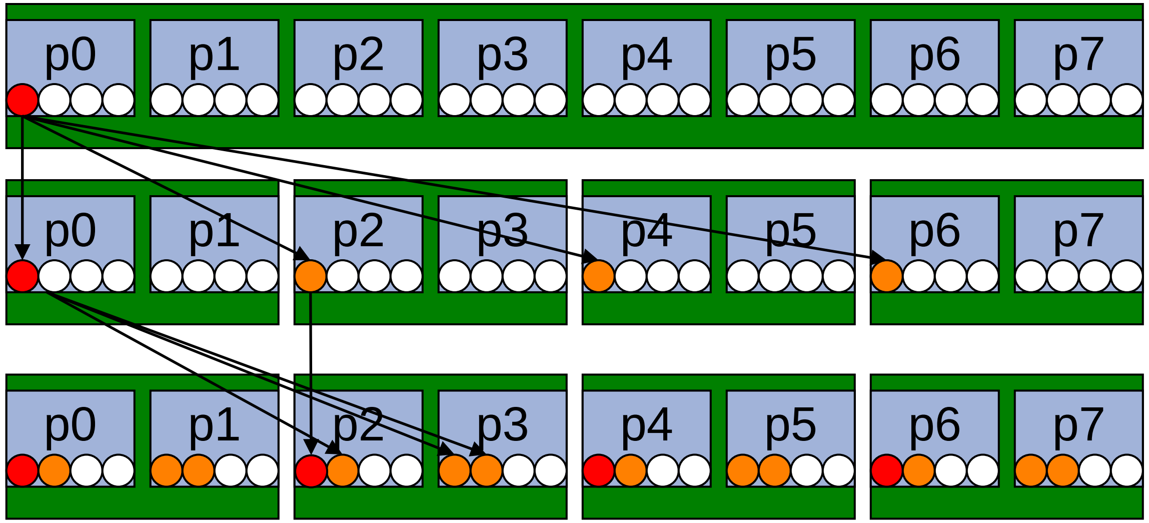
- Abstract names for OMP_PLACES:
  - → threads: Each place corresponds to a single hardware thread on the target machine.
  - → cores: Each place corresponds to a single core (having one or more hardware threads) on the target machine.
  - → sockets: Each place corresponds to a single socket (consisting of one or more cores) on the target machine.
  - → ll_caches: Each place corresponds to a set of cores that share the last level cache.
  - → numa_domains: Each place corresponds to a set of cores for which their closest memory is: the same memory; and at a similar distance from the cores.

Places + Binding Policies (2/2)

- Example‘s Objective:
  - → separate cores for outer loop and near cores for inner loop
- Outer Parallel Region: proc_bind(spread) num_threads(4)
  Inner Parallel Region: proc_bind(close) num_threads(4)
  - → spread creates partition, compact binds threads within respective partition

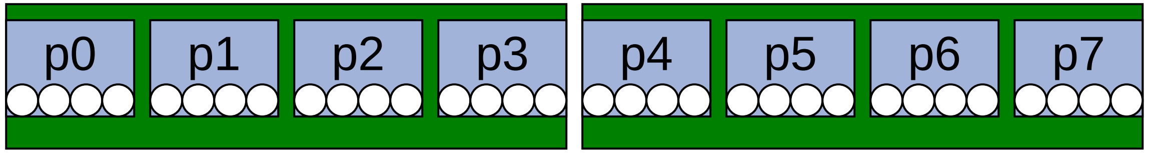
```
OMP_PLACES=(0,1,2,3), (4,5,6,7), ... = (0-3):8:4   = cores
#pragma omp parallel proc_bind(spread) num_threads(4)
#pragma omp parallel proc_bind(close) num_threads(4)
```

- Example
  - → initial
  - → spread 4
  - → close 4



More Examples (1/3)

- Assume the following machine:



→2 sockets, 4 cores per socket, 4 hyper-threads per core

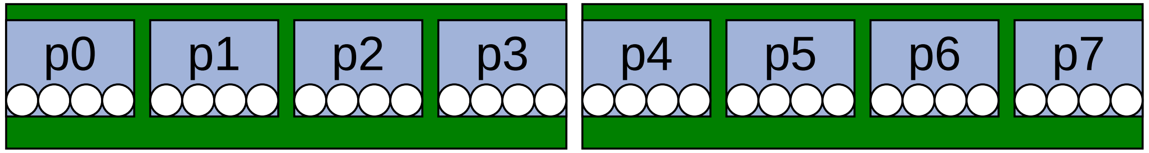
- Parallel Region with two threads, one per socket

→`OMP_PLACES=sockets`

→`#pragma omp parallel num_threads(2) proc_bind(spread)`

More Examples (2/3)

- Assume the following machine:



- Parallel Region with four threads, one per core, but only on the first socket

  → `OMP_PLACES=cores`

  → `#pragma omp parallel num_threads(4) proc_bind(close)`

More Examples (3/3)

- Spread a nested loop first across two sockets, then among the cores within each socket, only one thread per core

→`OMP_PLACES=cores`

→`#pragma omp parallel num_threads(2) proc_bind(spread)`

→`#pragma omp parallel num_threads(4) proc_bind(close)`

Working with OpenMP Places

Places API (1/2) (just for reference)

- 1: Query information about binding and a single place of all places with ids 0 … `omp_get_num_places()`:

- `omp_proc_bind_t omp_get_proc_bind()`: returns the thread affinity policy (omp_proc_bind_false, true, master, …)

- `int omp_get_num_places()`: returns the number of places

- `int omp_get_place_num_procs(int place_num)`: returns the number of processors in the given place

- `void omp_get_place_proc_ids(int place_num, int* ids)`: returns the ids of the processors in the given place

Places API (2/2) (just for reference)

- 2: Query information about the place partition:
- `int omp_get_place_num()`: returns the place number of the place to which the current thread is bound
- `int omp_get_partition_num_places()`: returns the number of places in the current partition
- `void omp_get_partition_place_nums(int* pns)`: returns the list of place numbers corresponding to the places in the current partition

Places API: Example (just for reference)

- Simple routine printing the processor ids of the place the calling thread is bound to:

```
void print_binding_info() {
     int my_place = omp_get_place_num();
     int place_num_procs = omp_get_place_num_procs(my_place);

     printf("Place consists of %d processors: ", place_num_procs);

     int *place_processors = malloc(sizeof(int) * place_num_procs);
     omp_get_place_proc_ids(my_place, place_processors)

     for (int i = 0; i < place_num_procs - 1; i++) {
             printf("%d ", place_processors[i]);
     }
     printf("\n");

     free(place_processors);
}
```

OpenMP 5.0 way to do this

- Set OMP_DISPLAY_AFFINITY=TRUE
  - →Instructs the runtime to display formatted affinity information
  - →Example output for two threads on two physical cores:

```
nesting_level=   1,   thread_num=   0,   thread_affinity=   0,1
nesting_level=   1,   thread_num=   1,   thread_affinity=   2,3
```

  - →Output can be formatted with OMP_AFFINITY_FORMAT env var or corresponding routine
  - →Formatted affinity information can be printed with `omp_display_affinity(const char* format)`

Affinity format specification

t	omp_get_team_num()
T	omp_get_num_teams()
L	omp_get_level()
n	omp_get_thread_num()
N	omp_get_num_threads()

a	omp_get_ancestor_thread_num() at level-1
H	hostname
P	process identifier
i	native thread identifier
A	thread affinity: list of processors (cores)

- Example:

```
OMP_AFFINITY_FORMAT=“Affinity: %0.3L %.8n %.15{A} %.12H‟
```

→Possible output:

```
Affinity: 001        0      0-1,16-17      host003
Affinity: 001        1      2-3,18-19      host003
```

A first summary

A first summary

- Everything under control?
- In principle Yes, but only if
  - →threads can be bound explicitly,
  - →data can be placed well by first-touch, or can be migrated,
  - →you focus on a specific platform (= OS + arch) → no portability

- What if the data access pattern changes over time?

- What if you use more than one level of parallelism?

NUMA Strategies: Overview

- First Touch: Modern operating systems (i.e., Linux >= 2.4) decide for a physical location of a memory page during the first page fault, when the page is first „touched“, and put it close to the CPU causing the page fault.

- Explicit Migration: Selected regions of memory (pages) are moved from one NUMA node to another via explicit OS syscall.

- Automatic Migration: Limited support in current Linux systems.

  →Not made for HPC and disabled on most HPC systems.

User Control of Memory Affinity

- Explicit NUMA-aware memory allocation:
  - → By carefully touching data by the thread which later uses it
  - → By changing the default memory allocation strategy
    - → Linux: `numactl` command
    - → Windows: `VirtualAllocExNuma()` (limited functionality)
  - → By explicit migration of memory pages
    - → Linux: `move_pages()`
    - → Windows: no option

- Example: using numactl to distribute pages round-robin:
  - → `numactl –interleave=all ./a.out`

Managing Memory Spaces

Memory Management

- Allocator := an OpenMP object that fulfills requests to allocate and deallocate storage for program variables
- OpenMP allocators are of type `omp_allocator_handle_t`
- Default allocator for Host
  - →via `OMP_ALLOCATOR` env. var. or corresponding API
- OpenMP 5.0 supports a set of memory allocators

OpenMP Allocators

- Selection of a certain kind of memory

Allocator name	Storage selection intent
omp_default_mem_alloc	use default storage
omp_large_cap_mem_alloc	use storage with large capacity
omp_const_mem_alloc	use storage optimized for read-only variables
omp_high_bw_mem_alloc	use storage with high bandwidth
omp_low_lat_mem_alloc	use storage with low latency
omp_cgroup_mem_alloc	use storage close to all threads in the contention group of the thread requesting the allocation
omp_pteam_mem_alloc	use storage that is close to all threads in the same parallel region of the thread requesting the allocation
omp_thread_local_mem_alloc	use storage that is close to the thread requesting the allocation

Using OpenMP Allocators

- New clause on all constructs with data sharing clauses:
  - → `allocate( [allocator:] list )`
- Allocation:
  - → `omp_alloc(size_t size, omp_allocator_handle_t allocator)`
- Deallocation:
  - → `omp_free(void *ptr, const omp_allocator_handle_t allocator)`
  - → `allocator` argument is optional
- `allocate` directive: standalone directive for allocation, or declaration of allocation stmt.

OpenMP Allocator Traits / 1

- Allocator traits control the behavior of the allocator

sync_hint	contended, uncontended, serialized, private default: contended
alignment	positive integer value that is a power of two default: 1 byte
access	all, cgroup, pteam, thread default: all
pool_size	positive integer value
fallback	default_mem_fb, null_fb, abort_fb, allocator_fb default: default_mem_fb
fb_data	an allocator handle
pinned	true, false default: false
partition	environment, nearest, blocked, interleaved default: environment

OpenMP Allocator Traits / 2

- `fallback`: describes the behavior if the allocation cannot be fulfilled
  - →`default_mem_fb`: return system's default memory
  - →Other options: null, abort, or use different allocator
- `pinned`: request pinned memory, i.e. for GPUs

OpenMP Allocator Traits / 3

- `partition`: partitioning of allocated memory of physical storage resources (think of NUMA)
  - →`environment`: use system's default behavior
  - →`nearest`: most closest memory
  - →`blocked`: partitioning into approx. same size with at most one block per storage resource
  - →`interleaved`: partitioning in a round-robin fashion across the storage resources

OpenMP Allocator Traits / 4

- Construction of allocators with traits via

  →
  ```
  omp_allocator_handle_t   omp_init_allocator(
  omp_memspace_handle_t memspace,
  int ntraits, const omp_alloctrait_t traits[]);
  ```

  →Selection of memory space mandatory

  →Empty traits set: use defaults

- Allocators have to be destroyed with `*_destroy_*`
- Custom allocator can be made default with

  ```
  omp_set_default_allocator(omp_allocator_handle_t allocator)
  ```

OpenMP Memory Spaces

- Storage resources with explicit support in OpenMP:

omp_default_mem_space	System’s default memory resource
omp_large_cap_mem_space	Storage with larg(er) capacity
omp_const_mem_space	Storage optimized for variables with constant value
omp_high_bw_mem_space	Storage with high bandwidth
omp_low_lat_mem_space	Storage with low latency

→Exact selection of memory space is implementation-def.

→Pre-defined allocators available to work with these

Programming OpenMP

NUMA

Christian Terboven

Michael Klemm

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

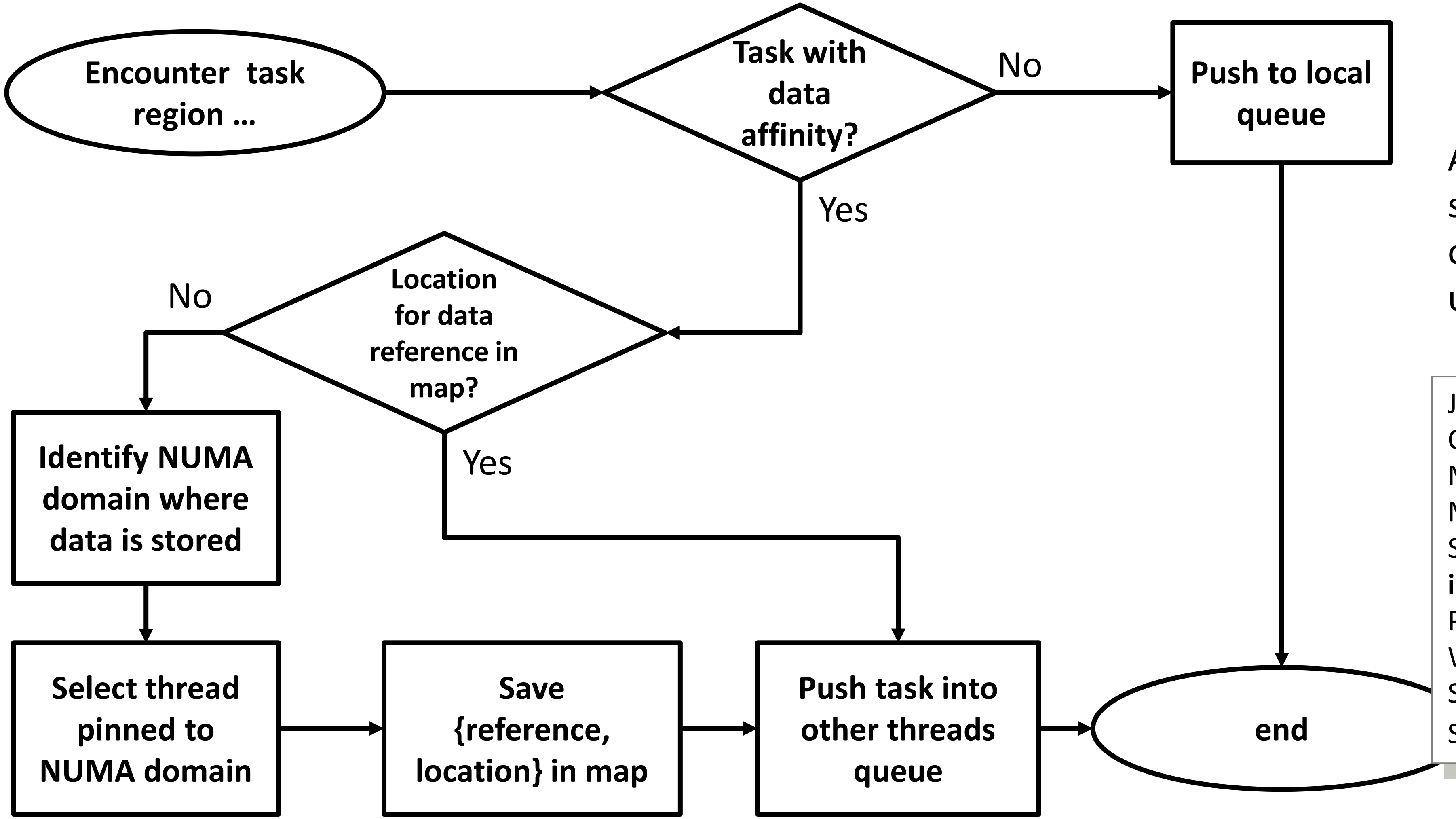
- Excerpt from task-parallel STREAM

```
#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

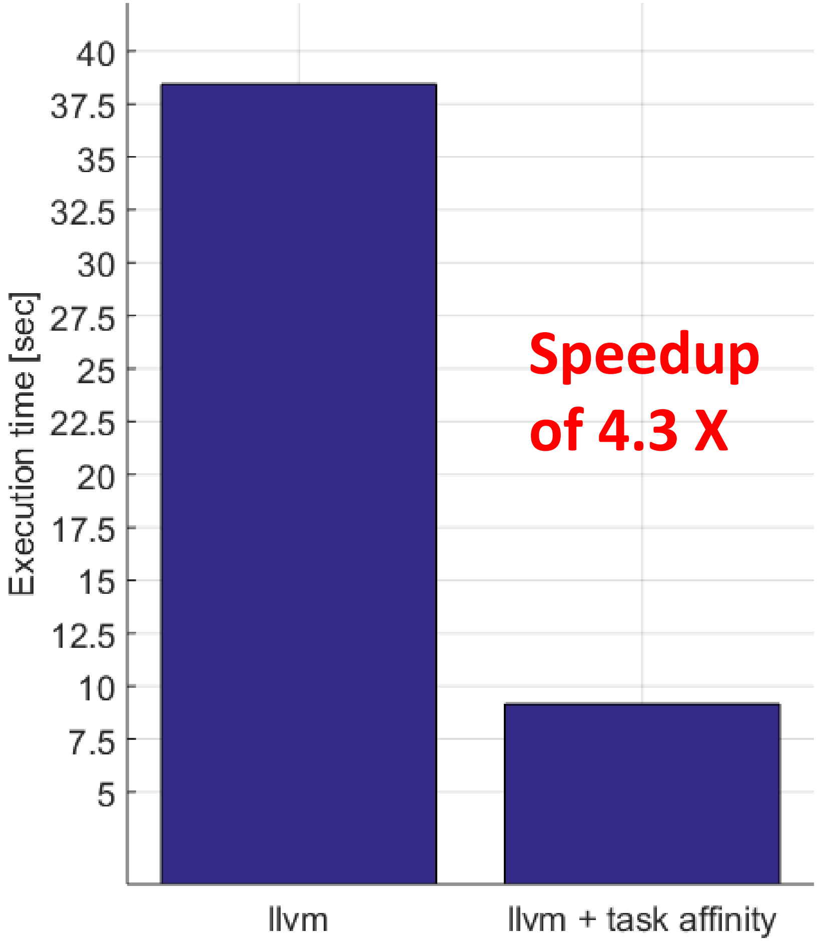

Encounter task region …
Task with data affinity?
No
Push to local queue
Yes
Location for data reference in map?
No
Identify NUMA domain where data is stored
Yes
Select thread pinned to NUMA domain
Save {reference, location} in map
Push task into other threads queue
end

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.

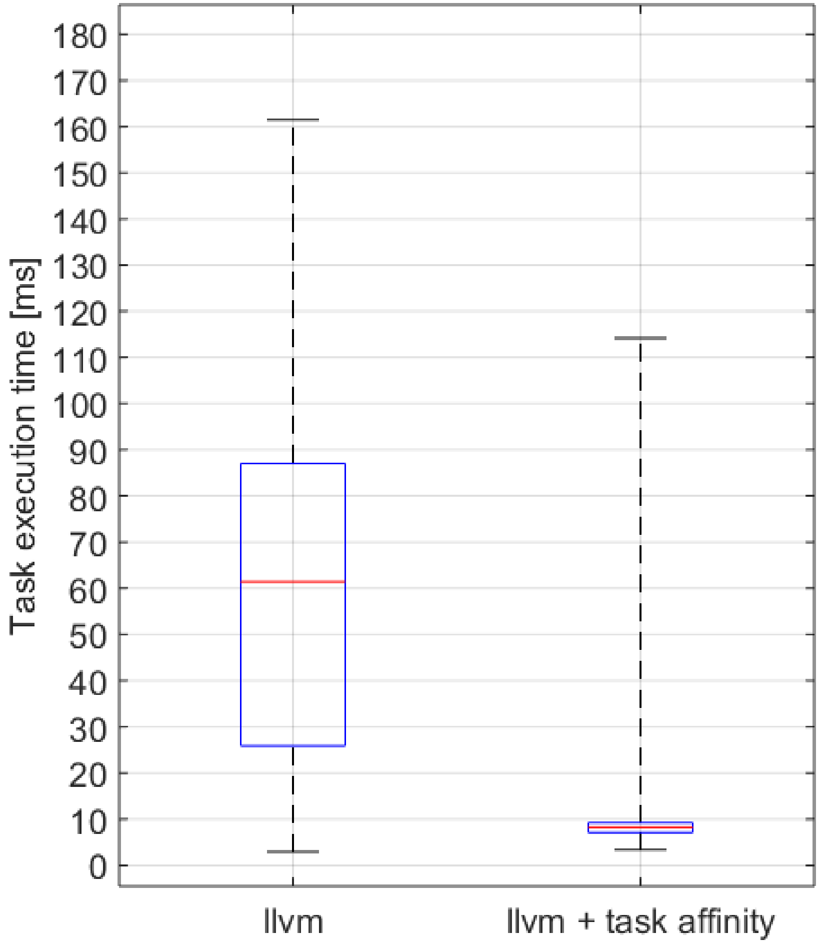
Evaluation

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