

Advanced OpenMP

**Tasking, Vectorization, Memory Access
Accelerators, Tools for Performance and Correctness**

Christian Terboven



Michael Klemm



Updated slides

- Slides are never perfect ...
- ... but we offer a free update service :-)



<https://tinyurl.com/prace-omp>



Agenda

■ Day I:

- Tasking
- Vectorization: OpenMP SIMD
- Memory Access: NUMA

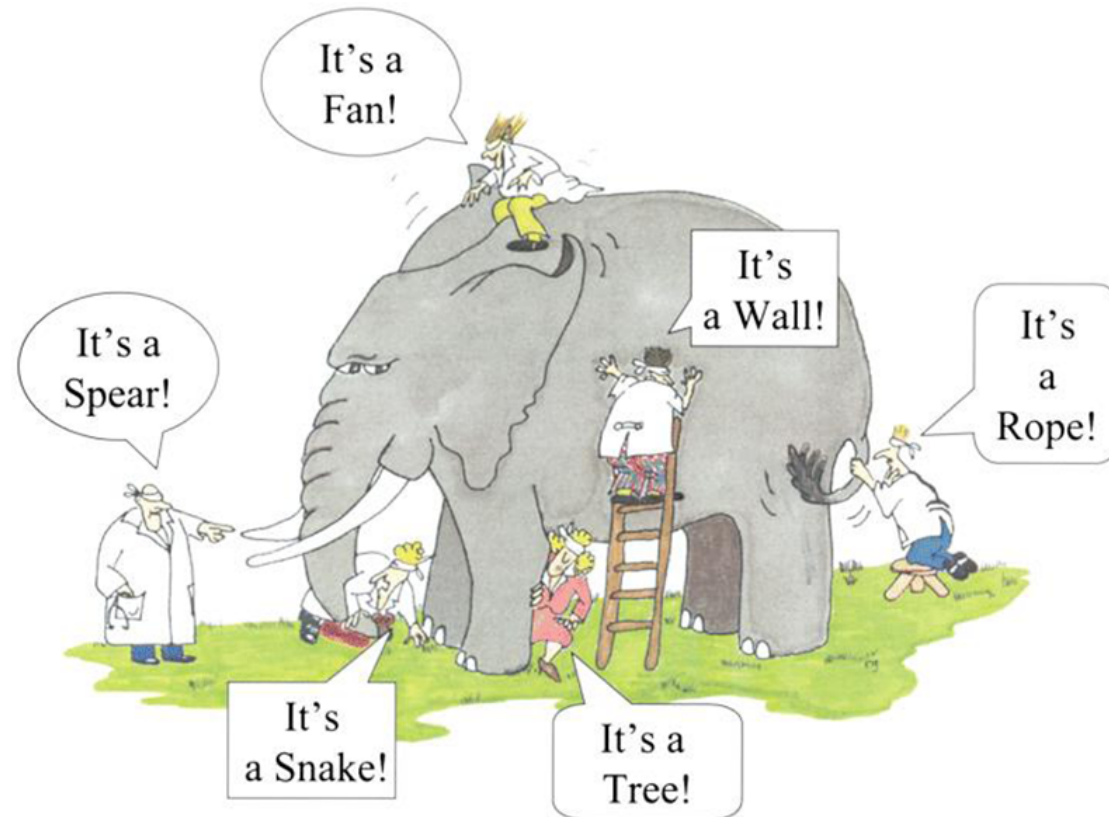
■ Day II:

- Accelerators
- Tools for Performance and Correctness
- Misc. OpenMP 5.0 Features & Outlook

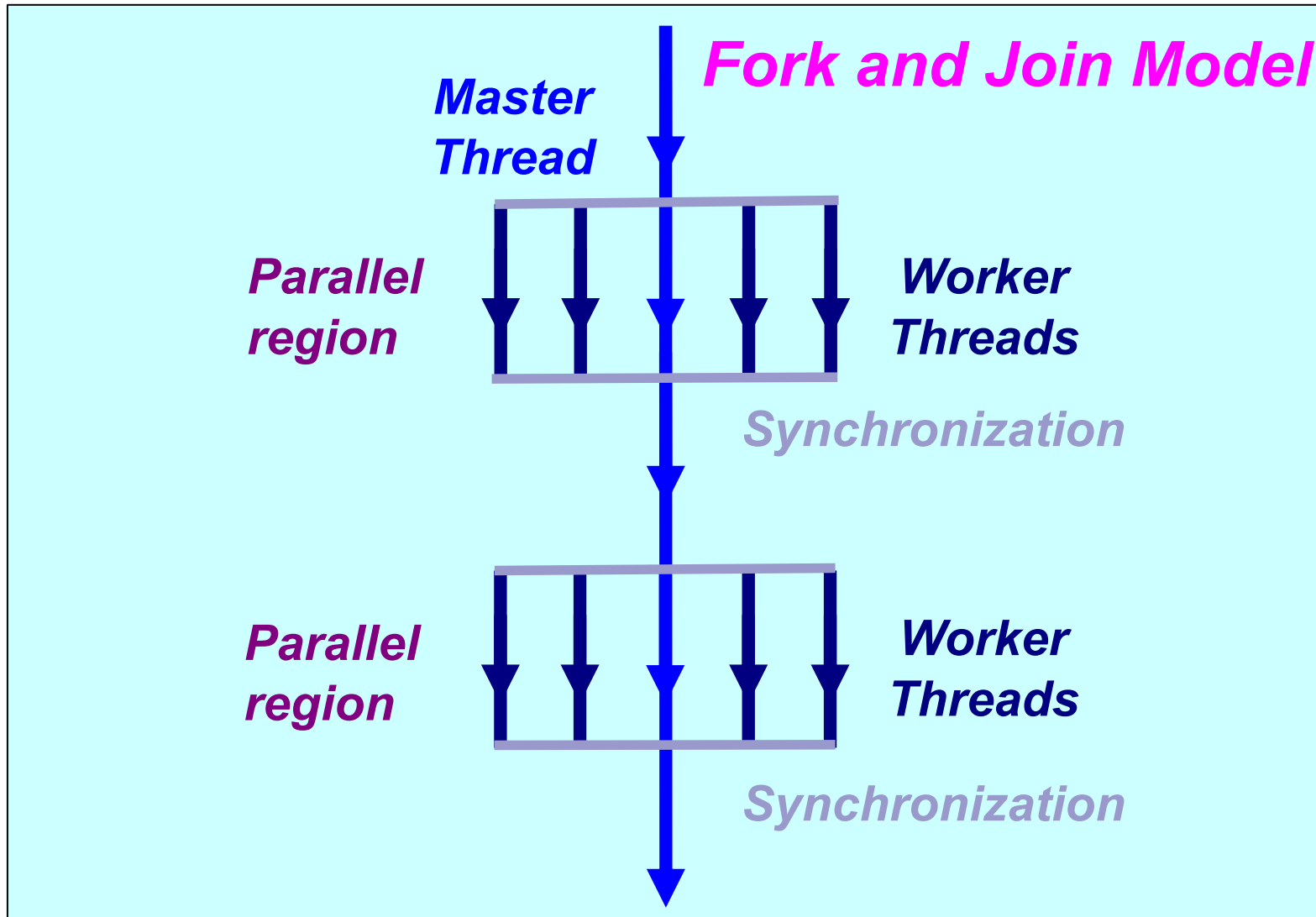
OpenMP Overview

What is OpenMP?

- De-facto standard Application Programming Interface (API) to write shared memory parallel applications in C, C++, and Fortran
- Consists of Compiler Directives, Runtime routines and Environment variables
- **Version 4.5 has been released in July 2015**
- **Version 5.0 has been released during last SC**



The OpenMP Execution Model



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

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



The Worksharing Constructs

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

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

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

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

The Single and Master Directives

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

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

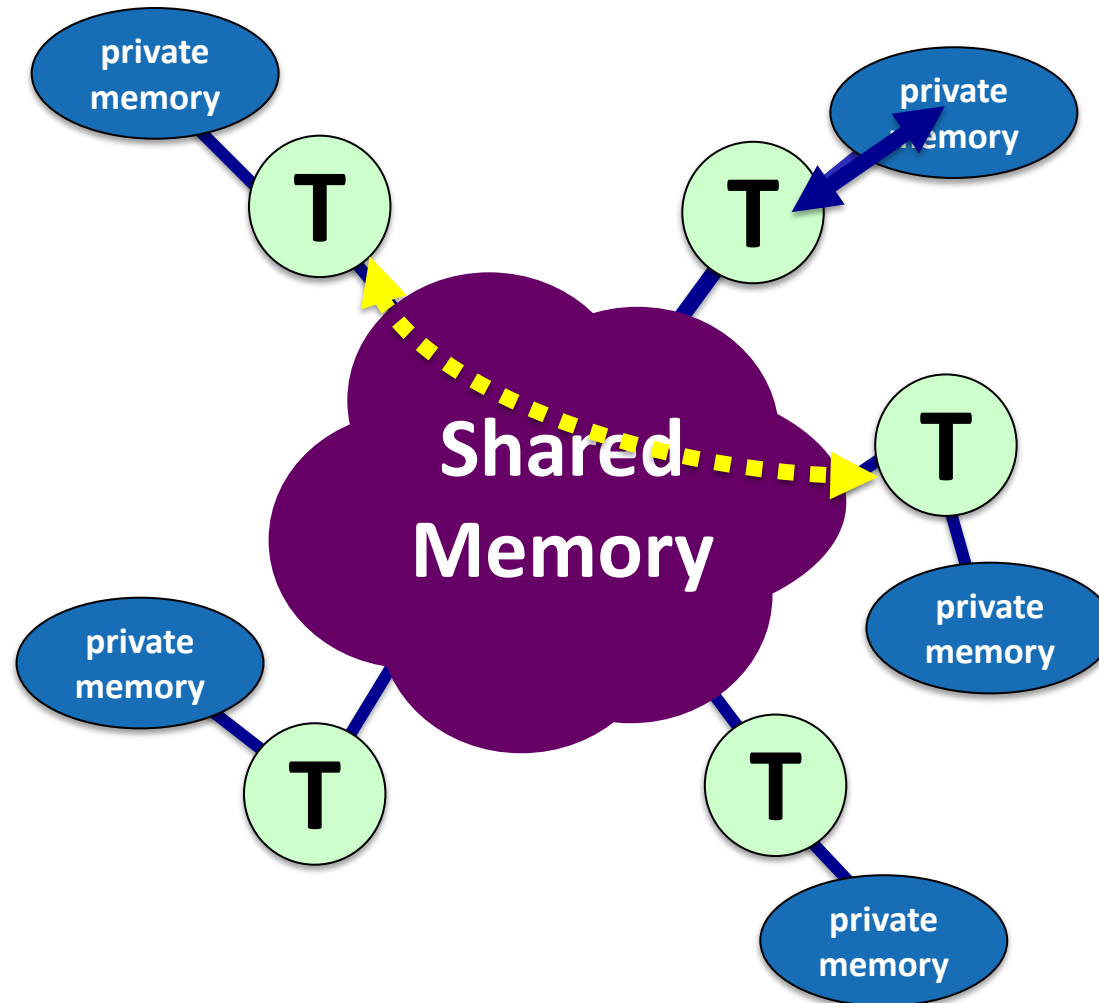
- Master: the master thread executes the code enclosed

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

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

The OpenMP Memory Model

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



The OpenMP Barrier

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

```
#pragma omp barrier
```

Tasking Motivation

Sudoku for Lazy Computer Scientists

- Lets solve Sudoku puzzles with brute multi-core force

	6					8	11			15	14			16	
15	11				16	14			12			6			
13		9	12					3	16	14		15	11	10	
2		16		11		15	10	1							
	15	11	10			16	2	13	8	9	12				
12	13			4	1	5	6	2	3				11	10	
5		6	1	12		9		15	11	10	7	16		3	
	2				10		11	6		5		13		9	
10	7	15	11	16				12	13					6	
9						1			2	16	10			11	
1		4	6	9	13			7		11		3	16		
16	14			7		10	15	4	6	1			13	8	
11	10		15				16	9	12	13			1	5	4
		12		1	4	6		16				11	10		
		5		8	12	13		10			11	2			14
3	16			10			7			6				12	

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

Parallel Brute-force Sudoku

- This parallel algorithm finds all valid solutions

	6					8	11			15	14			16	
15	11				16	14				12			6		
13		9	12					3	16	14		15	11	10	
2		16		11		15	10	1							
	15	11	10			16	2	13	8	9	12				
12	13			4	1	5	6	2	3				11	10	
5		6	1	12		9		15	11	10	7	16		3	
	2				10		11	6		5			13	9	
10	7	15	11	16				12	13					6	
9						1			2	16	10			11	
1		4	6	9	13			7		11		3	16		
16	14			7		10	15	4	6	1				13	8
11	10		15				16	9	12	13			1	5	4
		12		1	4	6		16				11	10		
		5		8	12	13		10			11	2			14
3	16			10			7			6					12

- (1) Search an empty field

```
#pragma omp parallel
#pragma omp single
such that one task starts the
execution of the algorithm
```

- (2) Try all numbers:

- (2 a) Check Sudoku

- If invalid: skip

- If valid: Go to next number

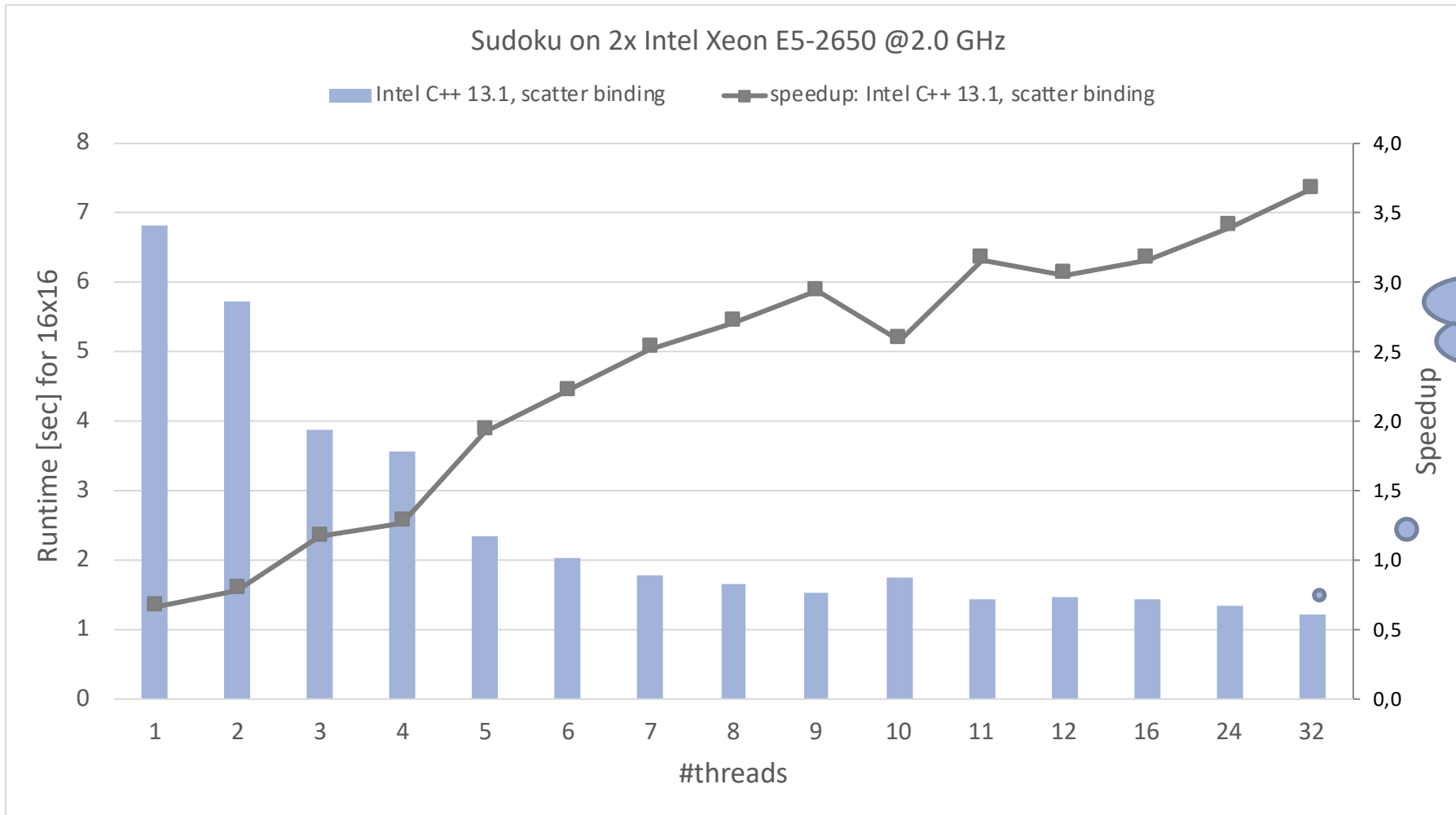
```
#pragma omp task
needs to work on a new copy
of the Sudoku board
```

- Wait for completion

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



Performance Evaluation



Is this the best we can do?

Tasking Overview

What is a task in OpenMP?

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

Tasking execution model

- Supports unstructured parallelism

→ unbounded loops

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

→ recursive functions

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

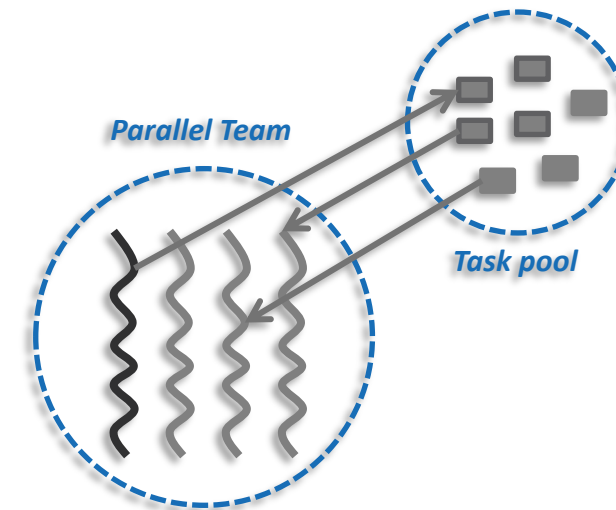
- Several scenarios are possible:

→ single creator, multiple creators, nested tasks (tasks & WS)

- All threads in the team are candidates to execute tasks

- Example (unstructured parallelism)

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



The task construct

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

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

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

- Where clause is one of:

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

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

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

→ depend(dep-type: list)	Synchronization
--------------------------	-----------------

→ untied	Task Scheduling
→ 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

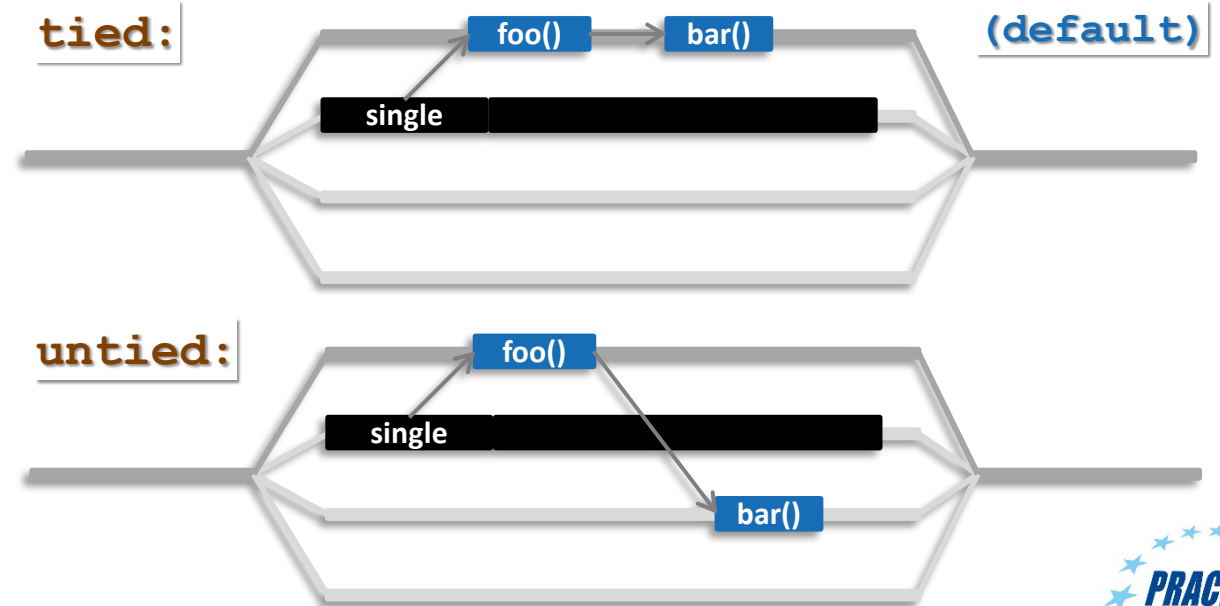
- Task scheduling points (and the taskyield directive)

- 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 scheduling: programmer's hints

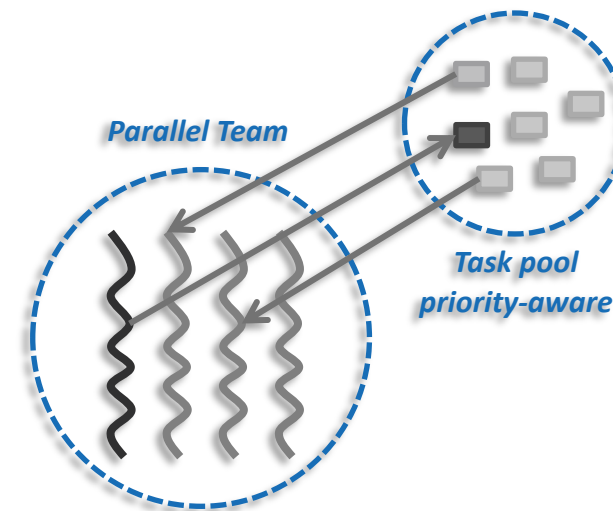
- Programmers may specify a priority value when creating a task

```
#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)

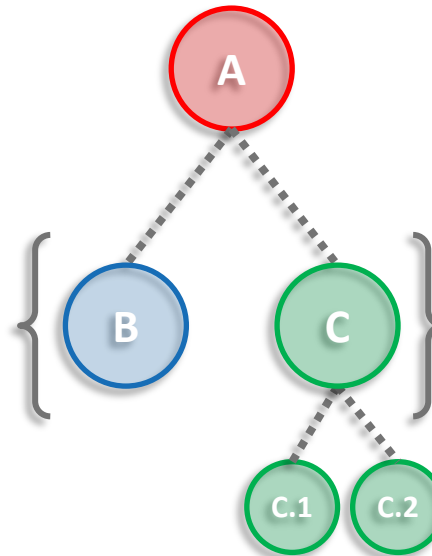
→ It is a stand-alone directive

```
#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
```

wait for...



Task synchronization: barrier semantics

- OpenMP barrier (implicit or explicit)

- All tasks created by any thread of the current team are guaranteed to be completed at barrier exit

```
#pragma omp barrier
```

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

Task synchronization: taskgroup construct

- The taskgroup construct (deep task synchronization)

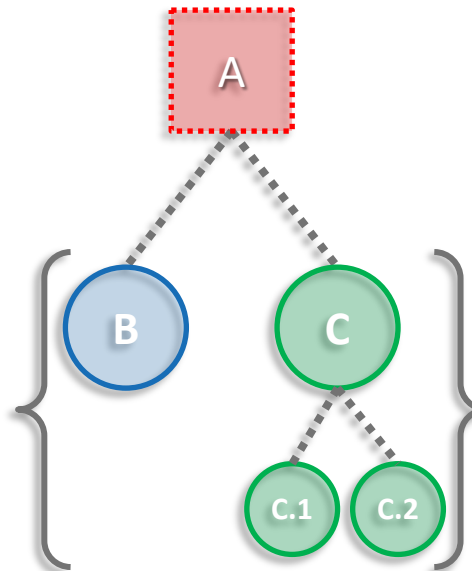
→ attached to a structured block; completion of all descendants of the current task; TSP at the end

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

→ where clause (could only be): reduction(reduction-identifier: list-items)

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

wait for...



Data Environment

Explicit data-sharing clauses

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

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

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

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

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

→ shared: data which is not explicitly included in any other data sharing clause will be **shared**

→ none: compiler will issue an error if the attribute is not explicitly set by the programmer (very useful!!!)

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

```
#pragma omp task default(none)
{
  // Compiler will force to specify the scope for
  // every single variable referenced in the context
}
```

Hint: Use default(none) to be forced to think about every variable if you do not see clearly.

Pre-determined data-sharing attributes

- threadprivate variables are threadprivate (1)
- dynamic storage duration objects are shared (malloc, new,...) (2)
- static data members are shared (3)
- variables declared inside the construct
 - static storage duration variables are shared (4)
 - automatic storage duration variables are private (5)
- the loop iteration variable(s)...

```

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

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

1

```

int *p;

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

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

2

```

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

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

3

```

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

5

```

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

4

Implicit data-sharing attributes (in-practice)

■ Implicit data-sharing rules for the task region

- the **shared** attribute is lexically inherited
- in any other case the variable is **firstprivate**

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

■ (in-practice) variable values within the task:

- value of a: 1
- value of b: x // undefined (undefined in parallel)
- value of c: 3
- value of d: 4
- value of e: 5

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

Task reductions (using taskgroup)

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

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

- Register a new reduction at [1]
 - Computes the final result after [3]
- The (task) in_reduction clause [participating]

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

- Task participates in a reduction operation [2]

```
int res = 0;
node_t* node = NULL;
...
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp taskgroup task_reduction(+: res)
    { // [1]
      while (node) {
        #pragma omp task in_reduction(+: res) \
          firstprivate(node)

        { // [2]
          res += node->value;
        }
        node = node->next;
      }
    } // [3]
  }
}
```

Task reductions (+ modifiers)

■ Reduction modifiers

- Former reductions clauses have been extended
- task modifier allows to express task reductions
- Registering a new task reduction [1]
- Implicit tasks participate in the reduction [2]
- Compute final result after [4]

■ The (task) in_reduction clause [participating]

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

- Task participates in a reduction operation [3]

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

Tasking illustrated

Fibonacci illustrated

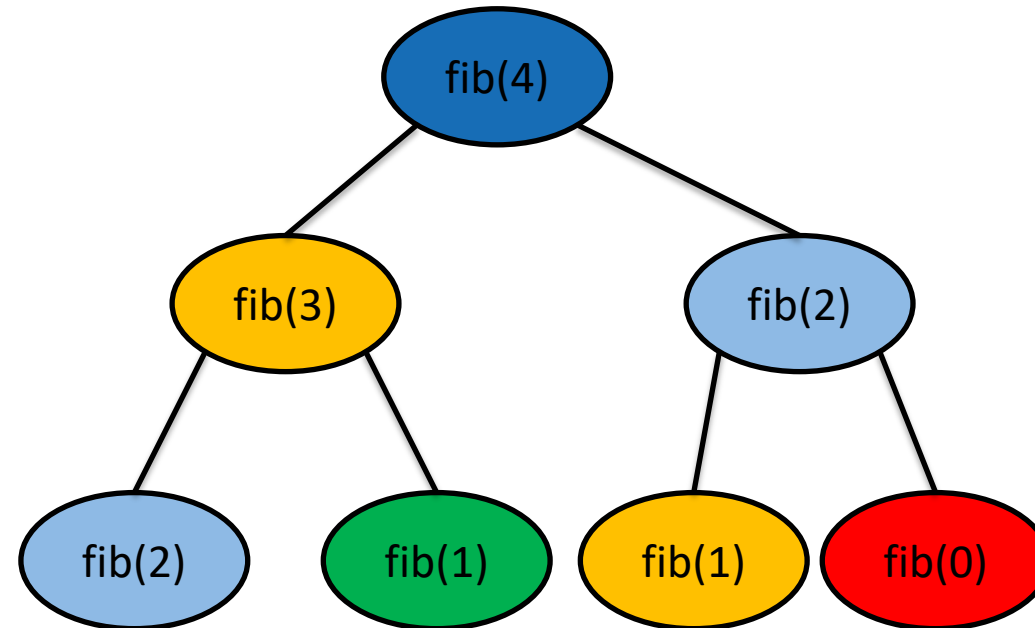
```
1  int main(int argc,  
2      char* argv[])  
3  {  
4      [...]  
5      #pragma omp parallel  
6      {  
7          #pragma omp single  
8          {  
9              fib(input);  
10         }  
11     }  
12     [...]  
13 }
```

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

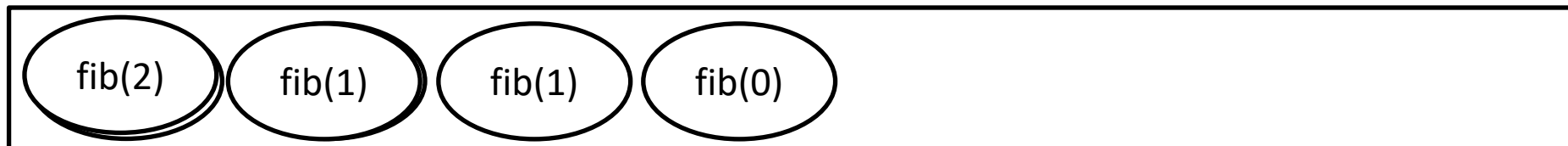
■ Only one Task / Thread enters fib() from main(), it is responsible for creating the two initial work tasks

■ Taskwait is required, as otherwise x and y would get lost

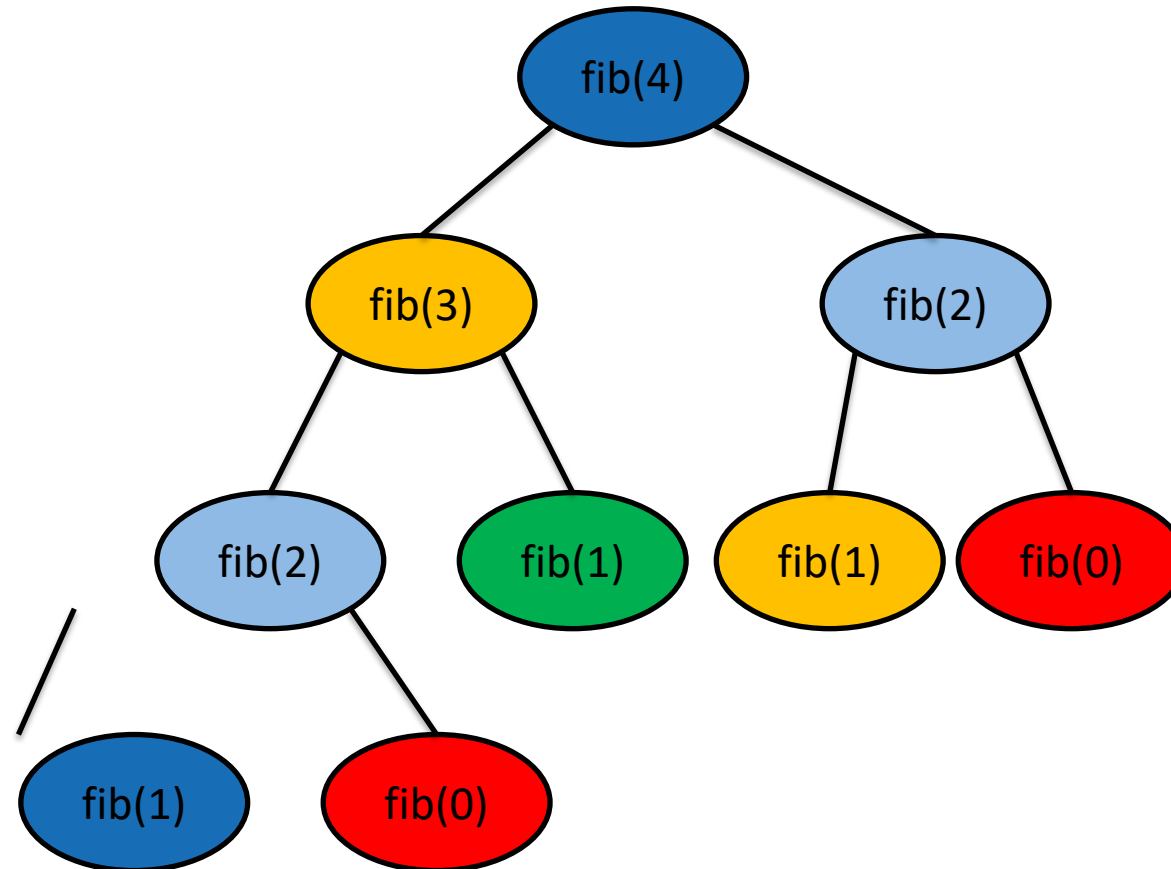
- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks



Task Queue



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



The `taskloop` Construct

Tasking use case: saxpy (taskloop)

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

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

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

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

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

- 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

- Clause: `in_reduction(r-id: list)`
 - Reuse an already defined reduction scope
 - All created tasks participate in the reduction
 - It can be used with the `nogroup*` clause, but it is user responsibility to guarantee result

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

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

Composite construct: taskloop simd

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

→ C/C++ syntax:

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

→ 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



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

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

  #pragma omp taskwait

  ● #pragma omp task
  x++;
}
```

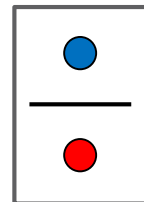
OpenMP 3.1

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

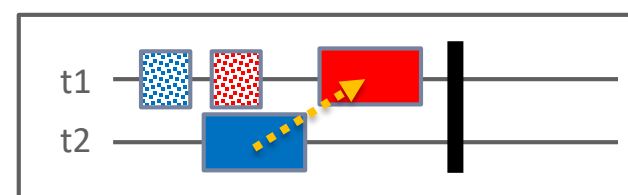
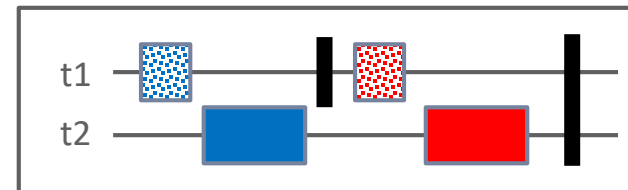
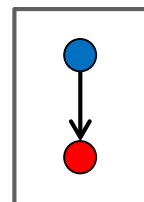
  ● #pragma omp task depend(inout: x)
  x++;
}
```

OpenMP 4.0

OpenMP 3.1



OpenMP 4.0



Task's creation time
 Task's execution time

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

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

  #pragma omp taskwait

  ● #pragma omp task
  x++;
}
```

OpenMP 3.1

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

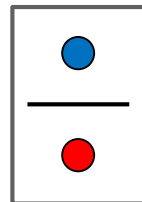
  #pragma omp taskwait

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

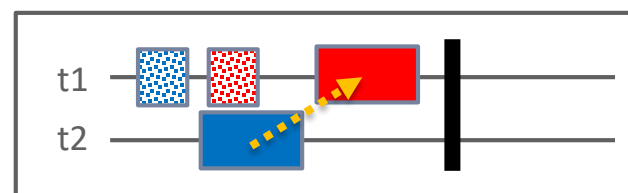
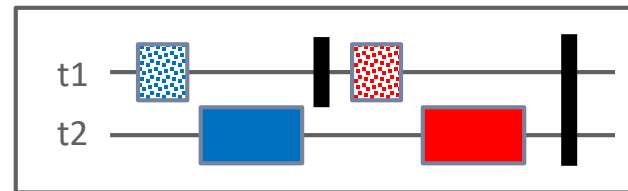
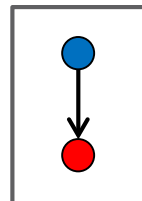
OpenMP 4.0

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

OpenMP 3.1



OpenMP 4.0



Task's creation time
Task's execution time

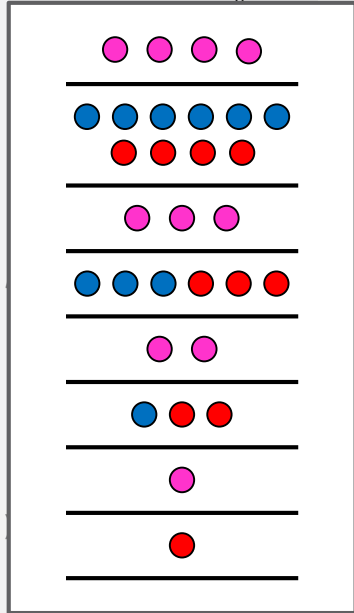


Motivation: Cholesky factorization

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

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

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

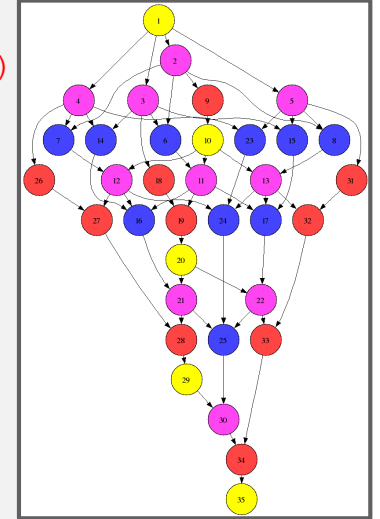


OpenMP 3.1

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

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

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



OpenMP 4.0

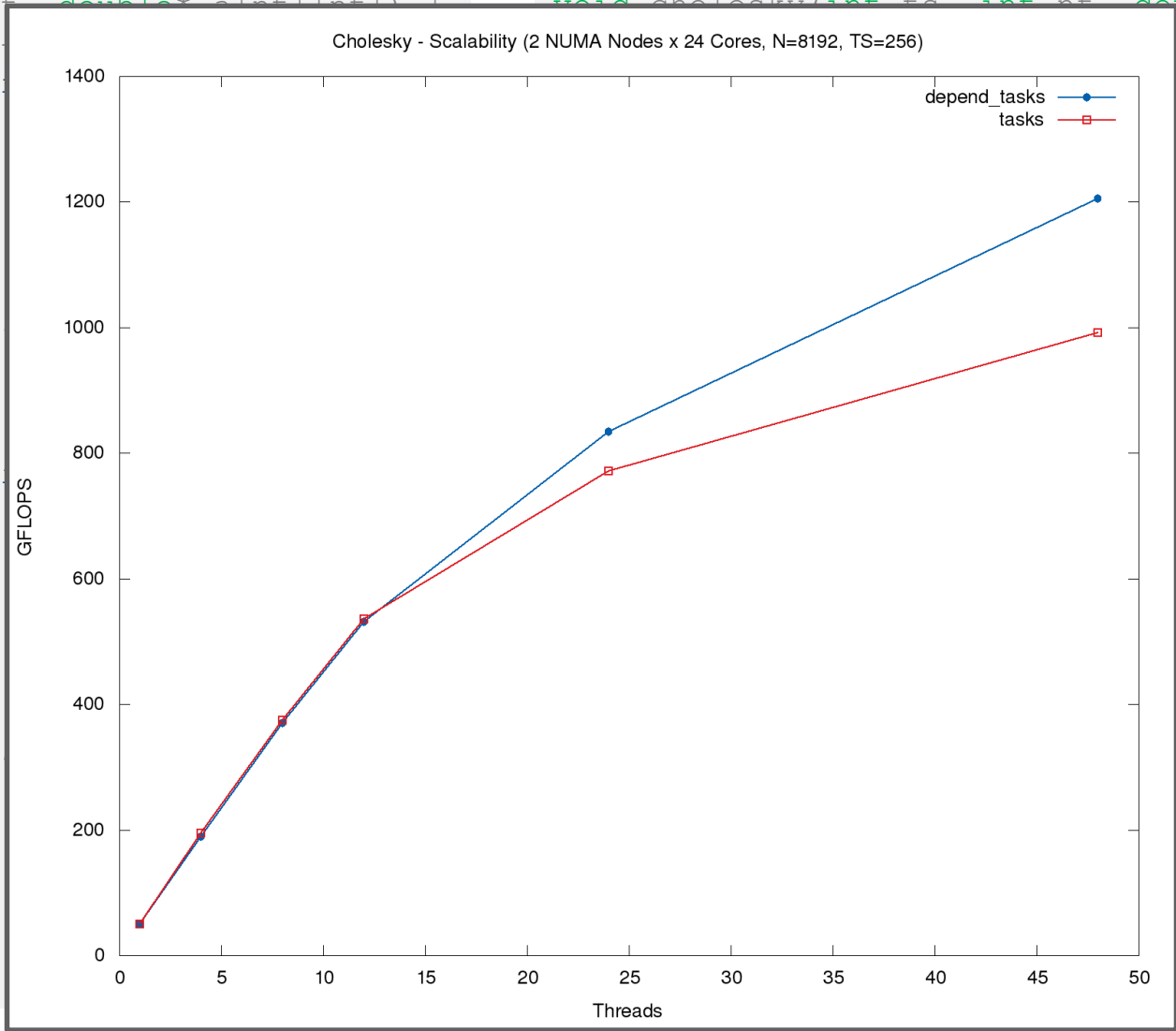
Motivation: Cholesky factorization

```

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

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

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

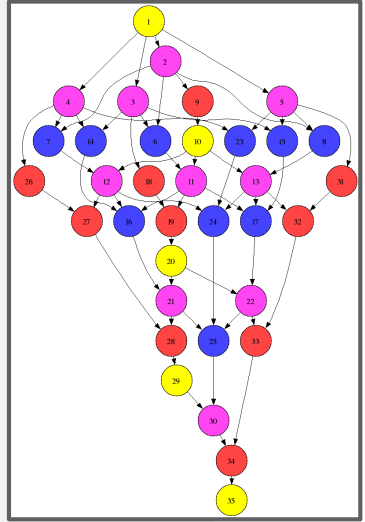


```

// ... (partial code from previous block) ...
    }
}

// ... (partial code from previous block) ...
    }
}

// ... (partial code from previous block) ...
    }
}
    
```



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 defines

→ the task will create
an out or inout

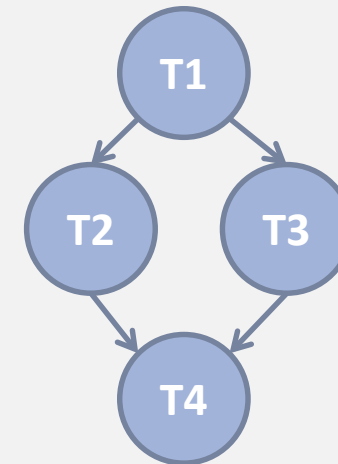
```

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
    { ... }
}
    
```



one of the list items in

- If a task defines

→ the task will create
an in, inout

one of the list items in

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

■ New dependency type: mutexinoutset

```

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

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

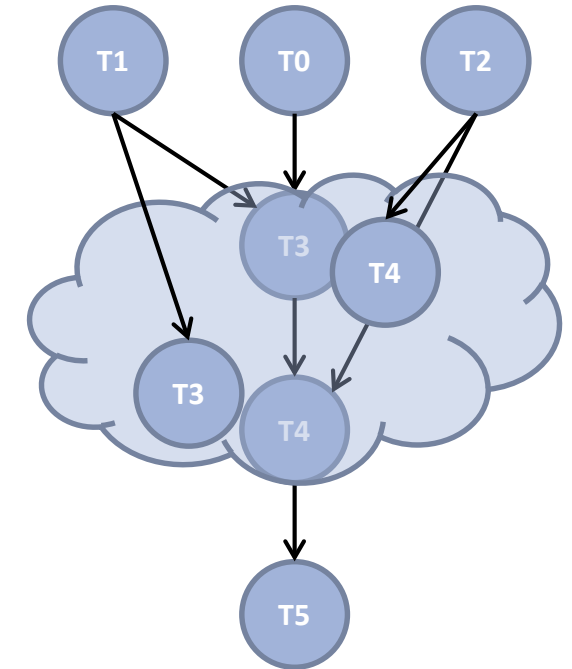
    #pragma omp task depend(out: y) //T2
    short_computation(y);

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

    #pragma omp task depend(in: y) depend(mutexinoutset //T4 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**

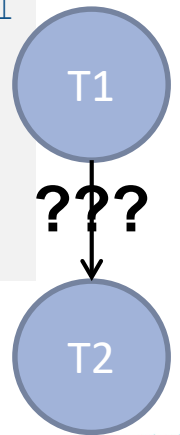
```
//test1.cc
int x = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(inout: x) //T1
    {
        #pragma omp task depend(inout: x) //T1.1
        x++;

        #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: depend clause (5)

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

```

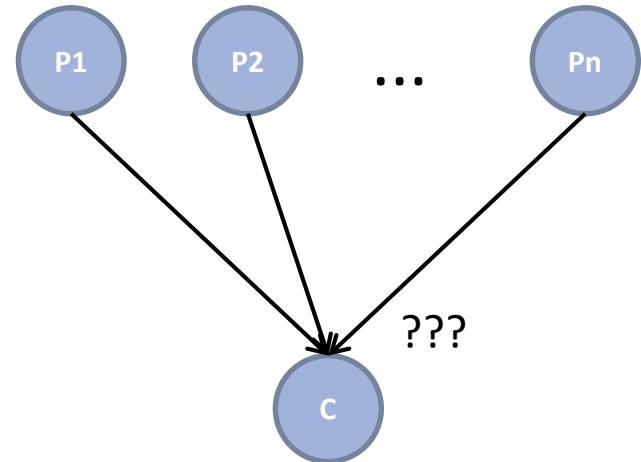
std::list<int> list = ...;
int n = list.size();

#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(iter@0)(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;
}
```

OK, but it always prints '0' :(

```
// 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;
}
```

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

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

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

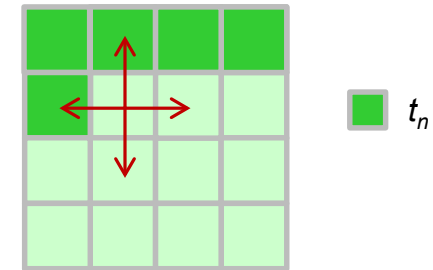
Use case

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



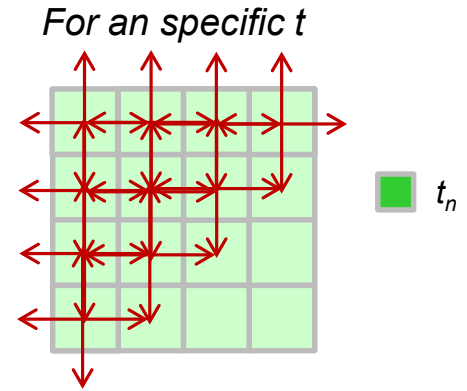
Each cell depends on:

- 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)

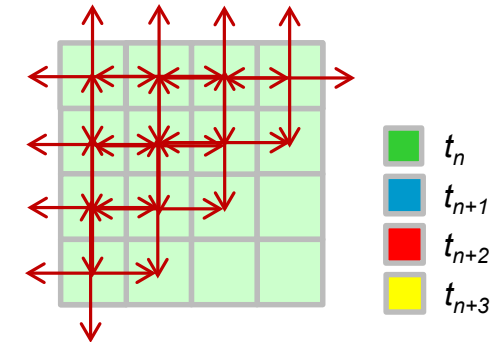
```
void gauss_seidel(int tsteps, int size, int TS, int (*p)[size]) {
    int NB = size / TS;
    #pragma omp parallel
    for (int t = 0; t < tsteps; ++t) {
        // First NB diagonals
        for (int diag = 0; diag < NB; ++diag) {
            #pragma omp for
            for (int d = 0; d <= diag;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

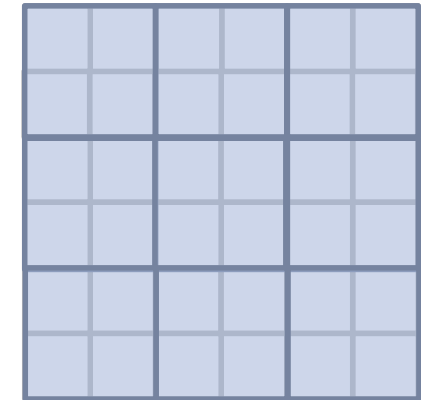
Use case : Gauss-seidel (5)

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

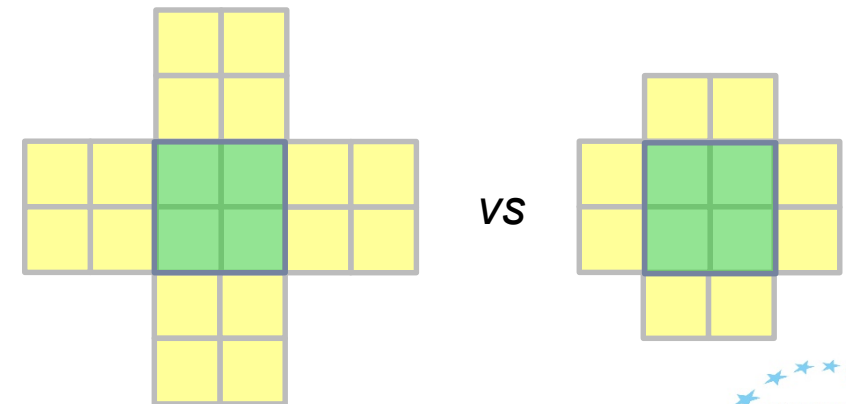
    #pragma omp parallel
    #pragma omp single
    for (int t = 0; t < tsteps; ++t)
        for (int ii=1; ii < size-1; ii+=TS)
            for (int jj=1; jj < size-1; jj+=TS) {
                #pragma omp task depend(inout: p[ii:TS][jj:TS])
                depend(in: p[ii-TS:TS][jj:TS], p[ii+TS:TS][jj:TS],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

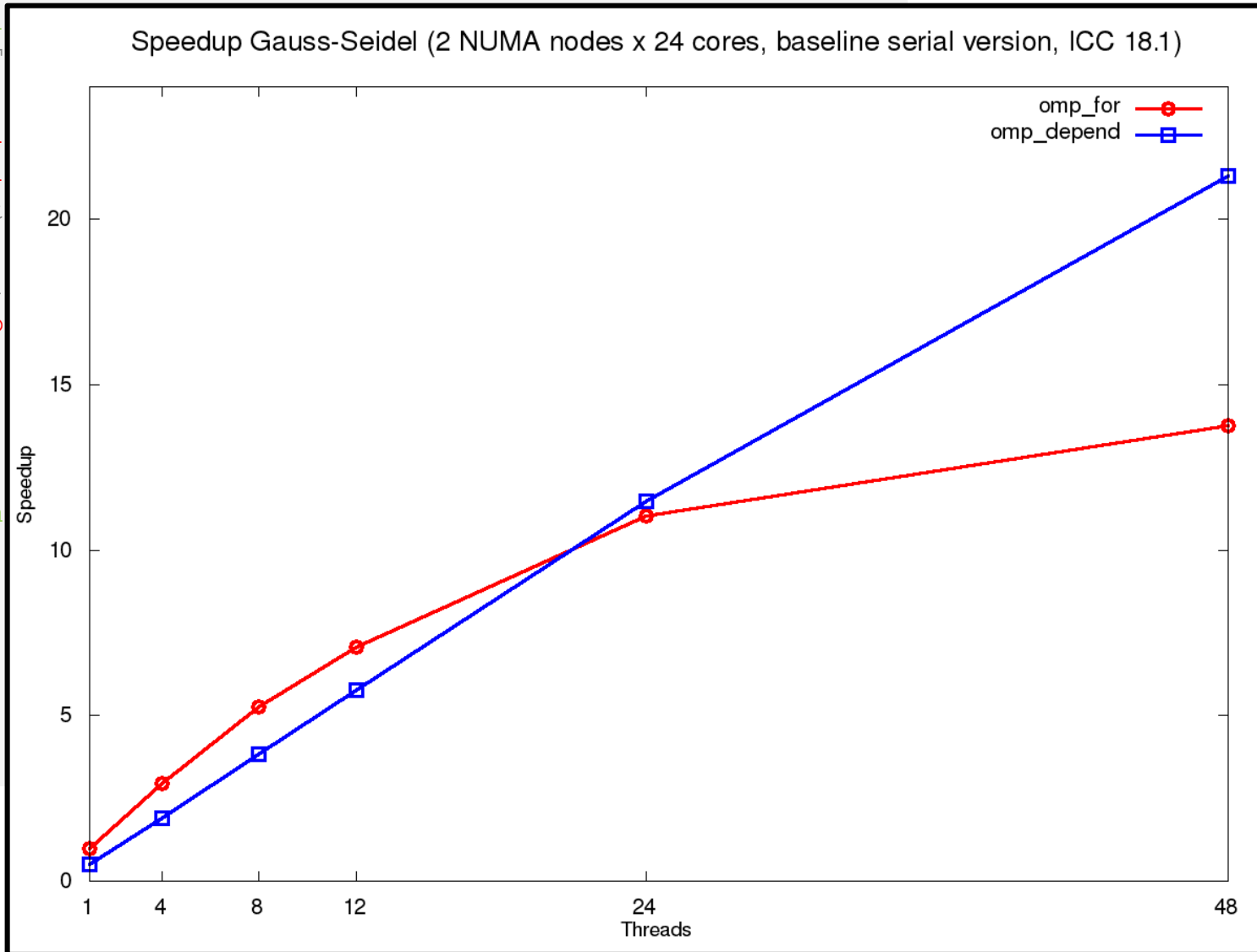


Use case : Gauss-seidel (5)

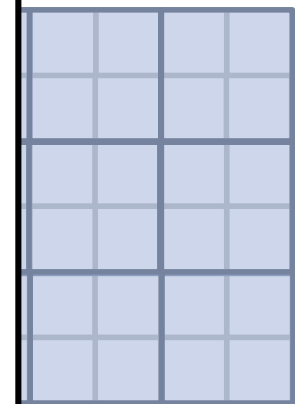
```

void gauss_seidel(int size)
{
    int NB = size / T;

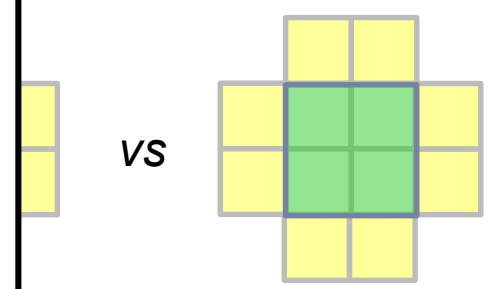
    #pragma omp parallel
    #pragma omp single
    for (int t = 0; t < T; t++)
        for (int ii=1; ii <= NB; ii++)
            for (int jj=1; jj <= NB; jj++)
                #pragma omp
                depend(
                    {
                        for (int
                            for (in
                                p[i]
                            }
                        }
                    }
                }
}
    
```



matrix region



the input dependences
the whole block rather
than 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;
}
```

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)`

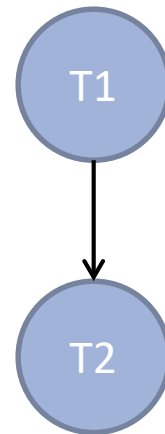
- `#pragma omp depobj (obj) destroy`

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
}
```

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

```
#pragma omp parallel
#pragma omp single
such that one task starts the
execution of the algorithm
```

- (2) Try all numbers:

- (2 a) Check Sudoku

- If invalid: skip

- If valid: Go to next field

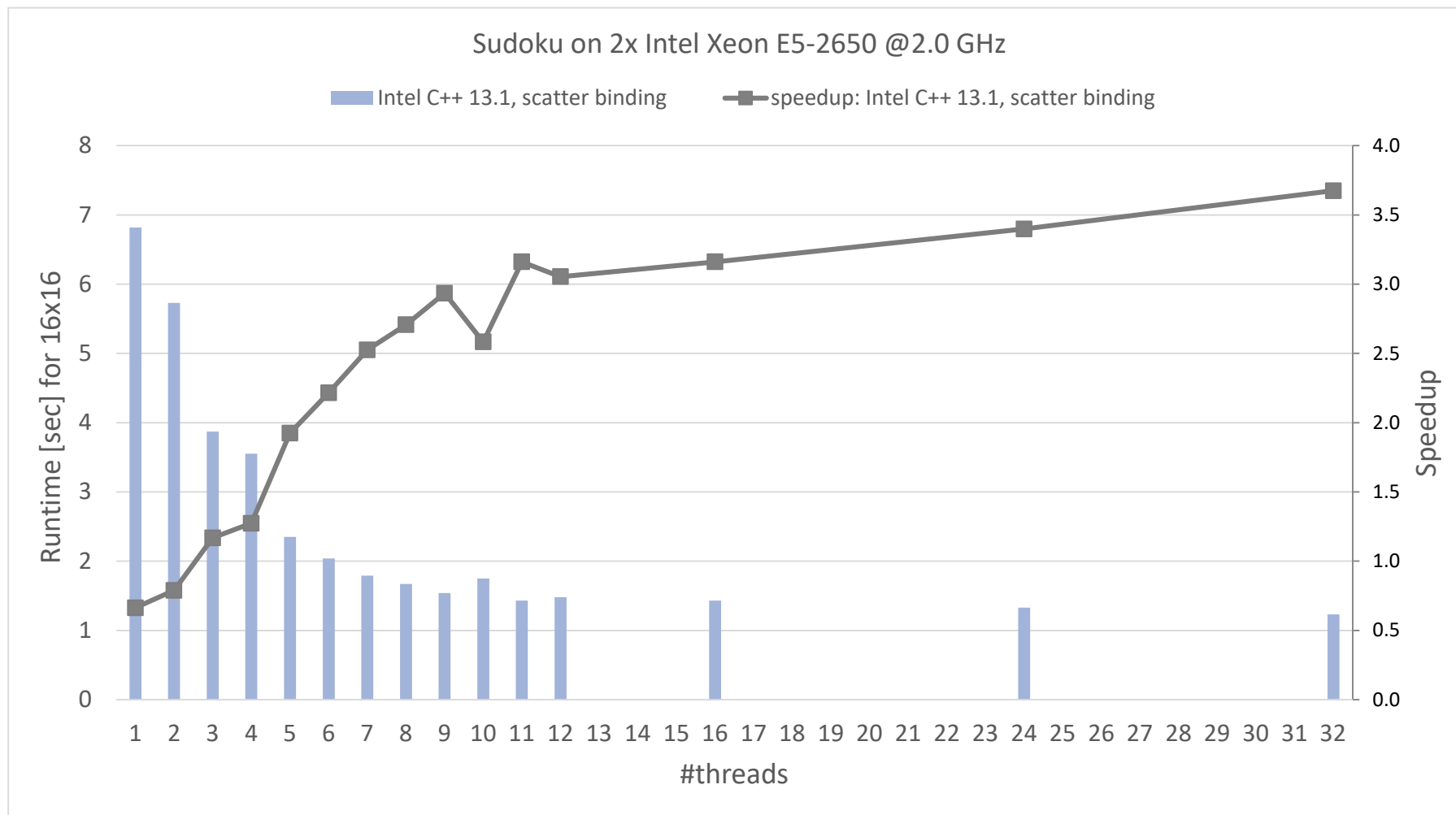
```
#pragma omp task
needs to work on a new copy
of the Sudoku board
```

- Wait for completion

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

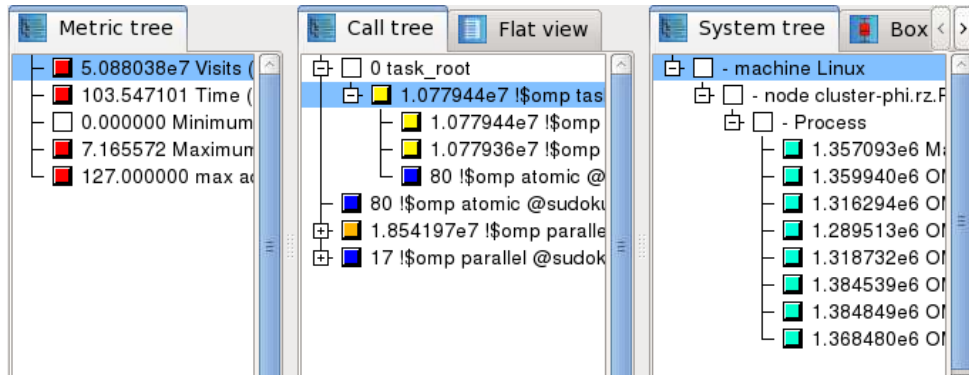


Performance Evaluation

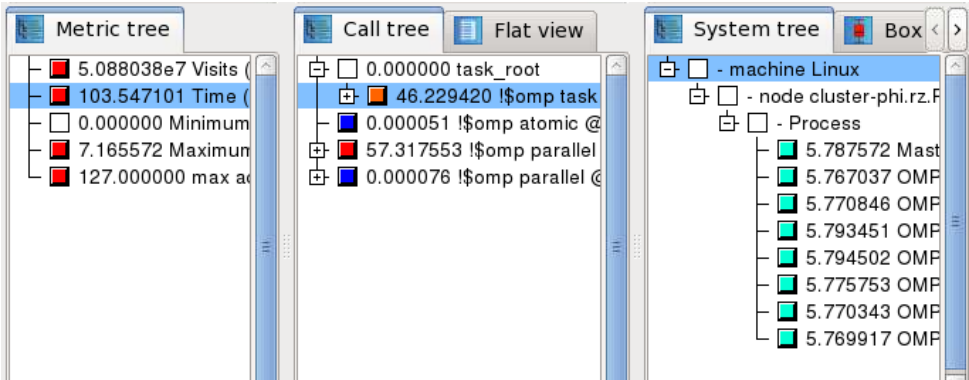


Performance Analysis

Event-based profiling provides a good overview :



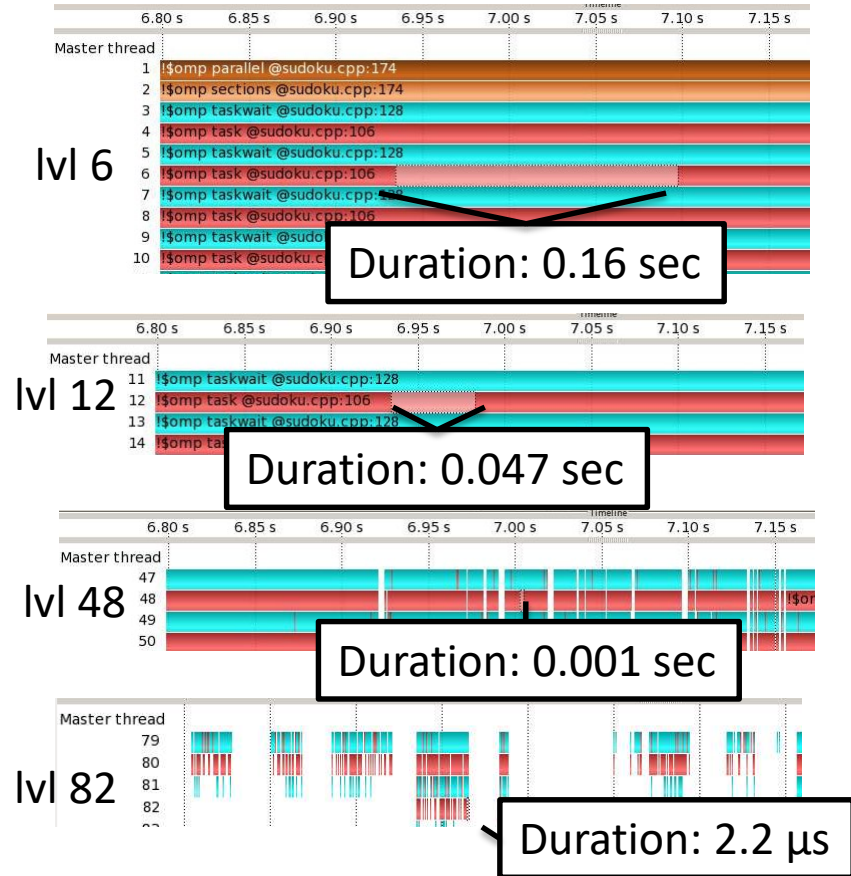
Every thread is executing ~1.3m tasks...



... in ~5.7 seconds.

=> average duration of a task is ~4.4 μ s

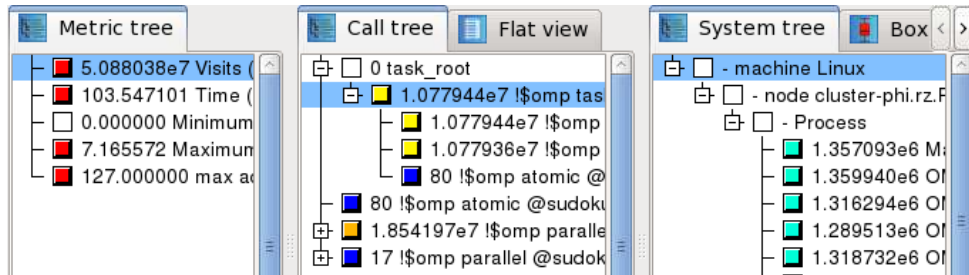
Tracing provides more details:



Tasks get much smaller down the call-stack.

Performance Analysis

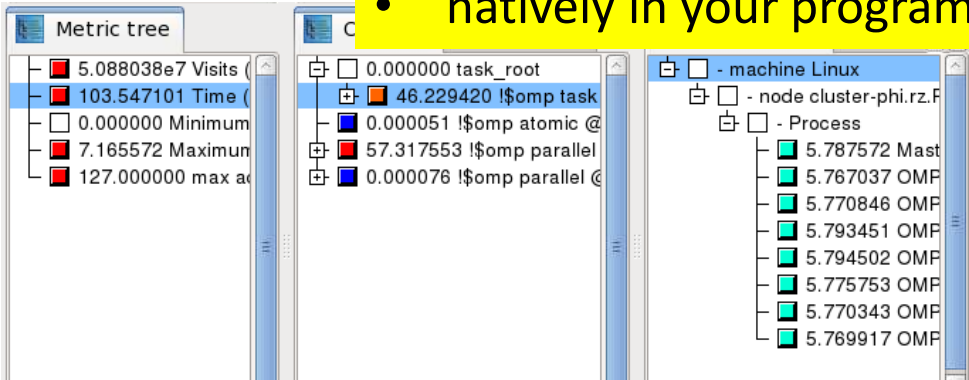
Event-based profiling provides a good overview :



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

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

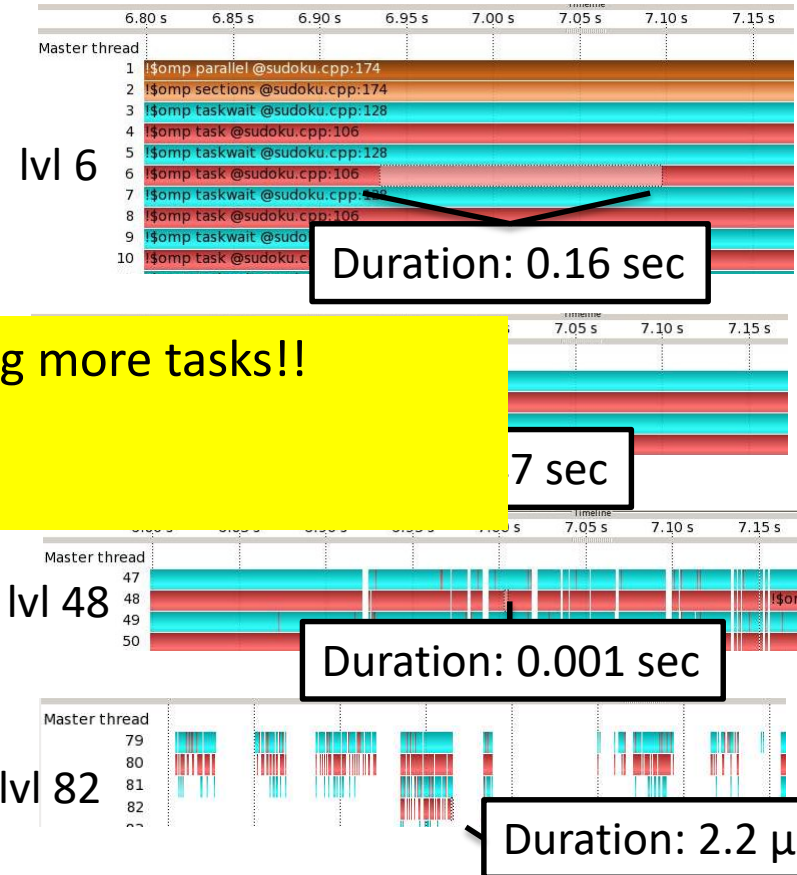
Every thread i



... in ~5.7 seconds.

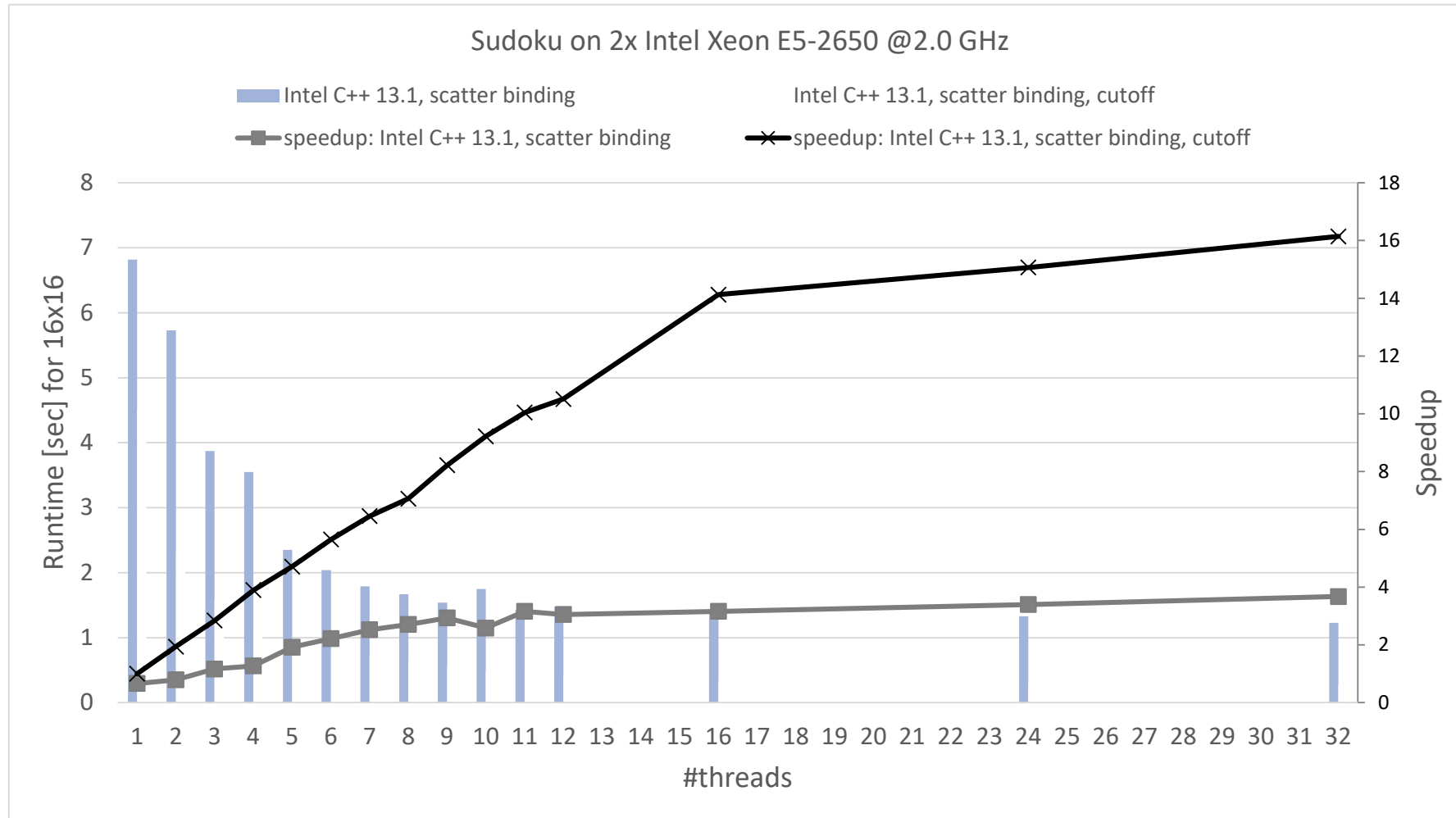
=> average duration of a task is ~4.4 μ s

Tracing provides more details:



Tasks get much smaller down the call-stack.

Performance Evaluation (with cutoff)



The `if` clause

- Rule of thumb: the `if (expression)` clause as a “switch off” mechanism
 - Allows lightweight implementations of task creation and execution but it reduces the parallelism

- If the `expression` of the `if` clause evaluates to `false`

- the encountering task is suspended
- the new task is executed immediately (task dependences are respected!!)
- the encountering task resumes its execution once the new task is completed
- This is known as *undeferred task*

```
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

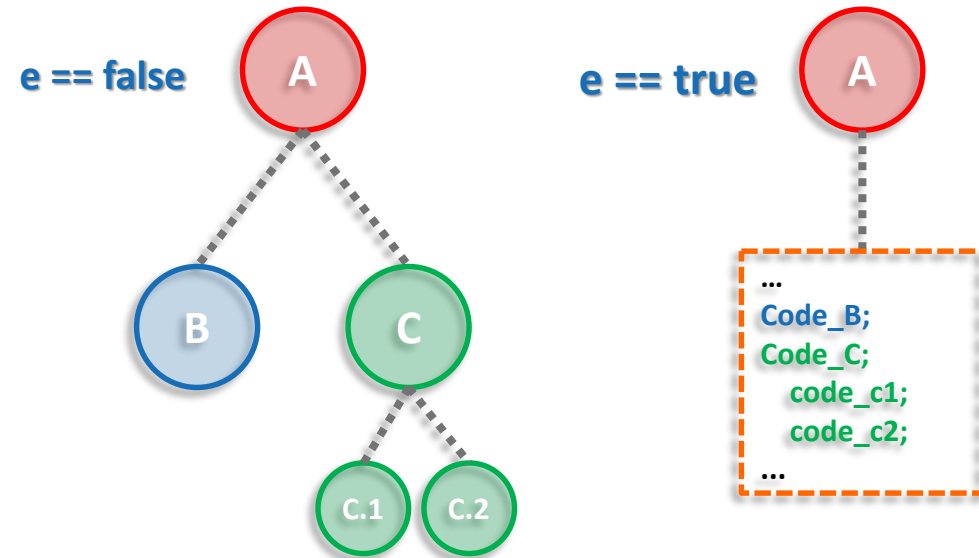
- The `final (expression)` clause

- 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` = (



Vectorization w/ OpenMP SIMD

Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/software/products.

All rights reserved. Intel, the Intel logo, Xeon, Xeon Phi, VTune, and Cilk are trademarks of Intel Corporation in the U.S. and other countries.

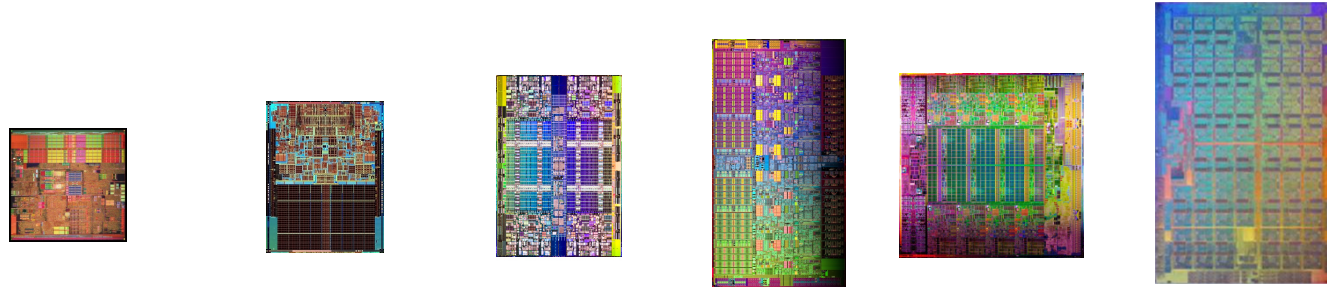
*Other names and brands may be claimed as the property of others.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Evolution of Intel Hardware



Images not intended to reflect actual die sizes

	64-bit Intel® Xeon® processor	Intel® Xeon® processor 5100 series	Intel® Xeon® processor 5500 series	Intel® Xeon® processor 5600 series	Intel® Xeon® processor E5-2600v3 series	Intel® Xeon® Scalable Processor
Frequency	3.6 GHz	3.0 GHz	3.2 GHz	3.3 GHz	2.3 GHz	2.5 GHz
Core(s)	1	2	4	6	18	28
Thread(s)	2	2	8	12	36	56
SIMD width	128 (2 clock)	128 (1 clock)	128 (1 clock)	128 (1 clock)	256 (1 clock)	512 (1 clock)

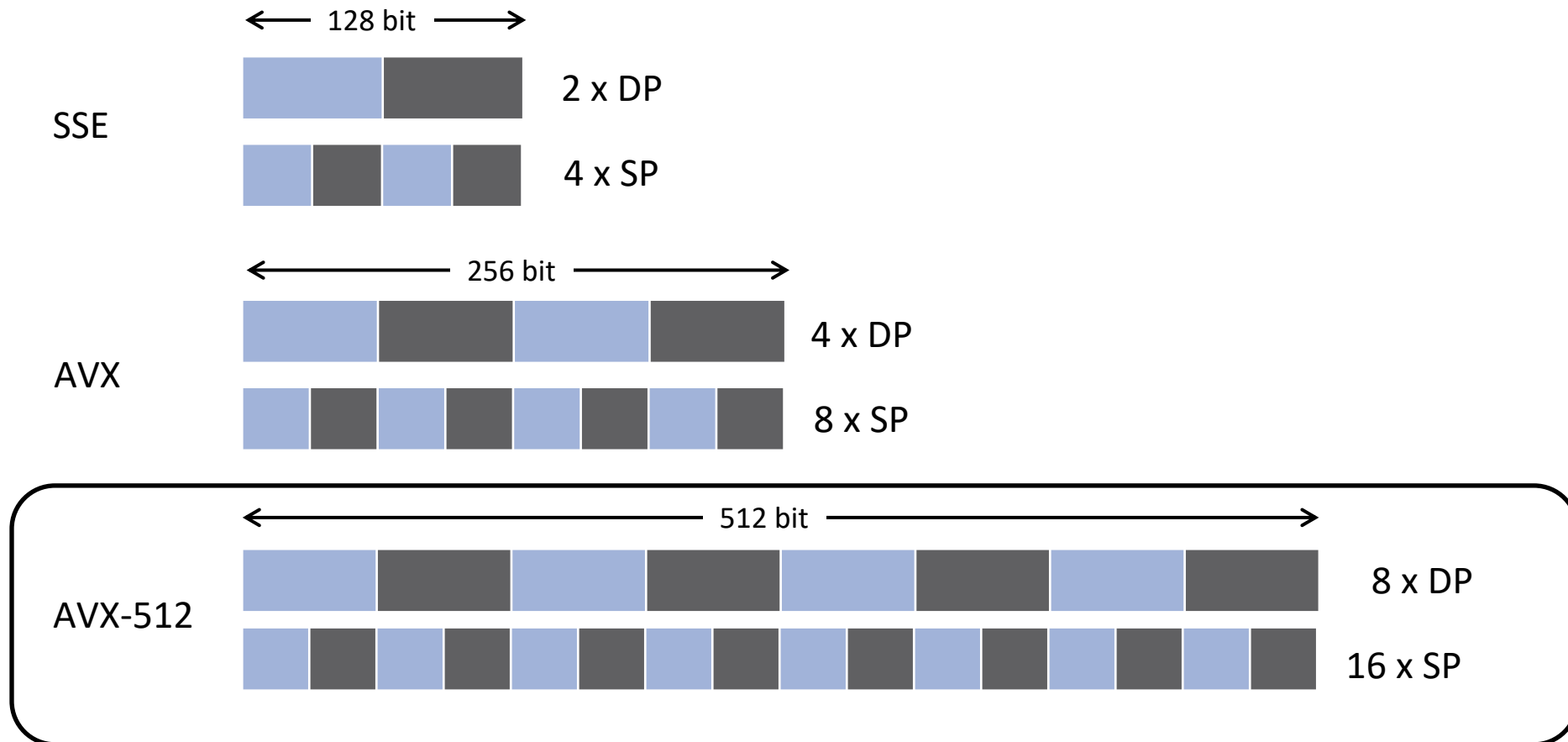
Levels of Parallelism

- OpenMP already supports several levels of parallelism in today's hardware

Cluster	Group of computers communicating through fast interconnect
Coprocessors/Accelerators	Special compute devices attached to the local node through special interconnect
Node	Group of processors communicating through shared memory
Socket	Group of cores communicating through shared cache
Core	Group of functional units communicating through registers
Hyper-Threads	Group of thread contexts sharing functional units
Superscalar	Group of instructions sharing functional units
Pipeline	Sequence of instructions sharing functional units
Vector	Single instruction using multiple functional units

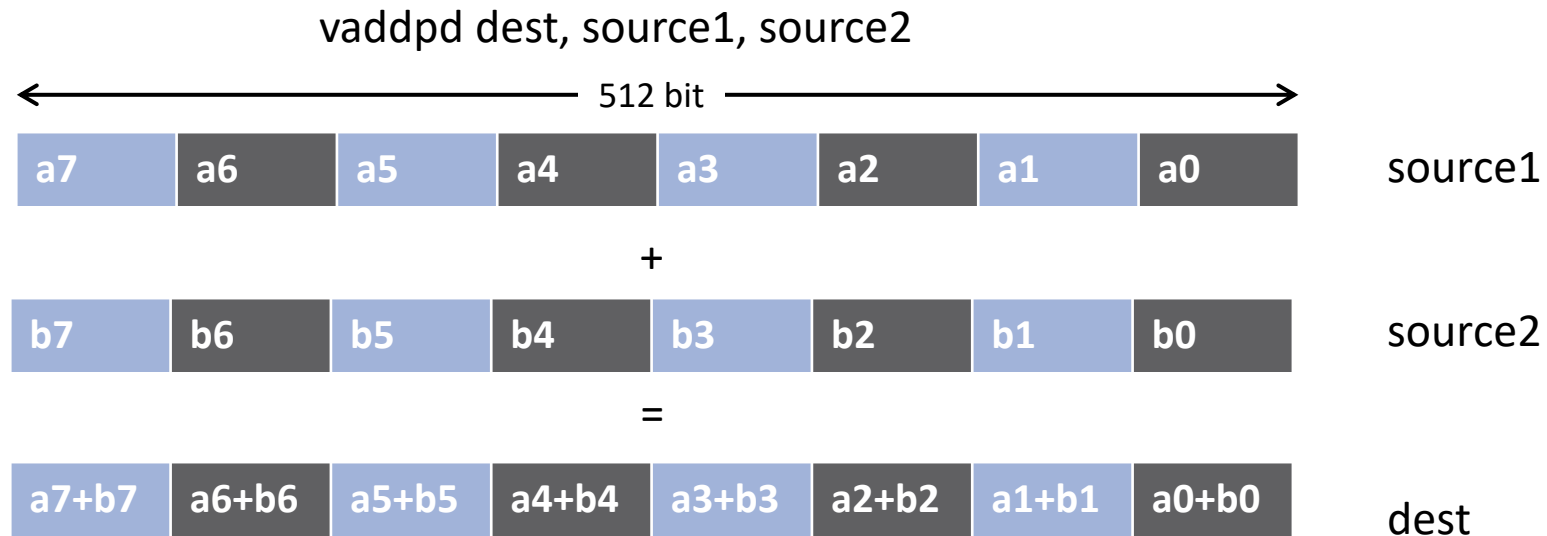
SIMD on Intel® Architecture

- Width of SIMD registers has been growing in the past:



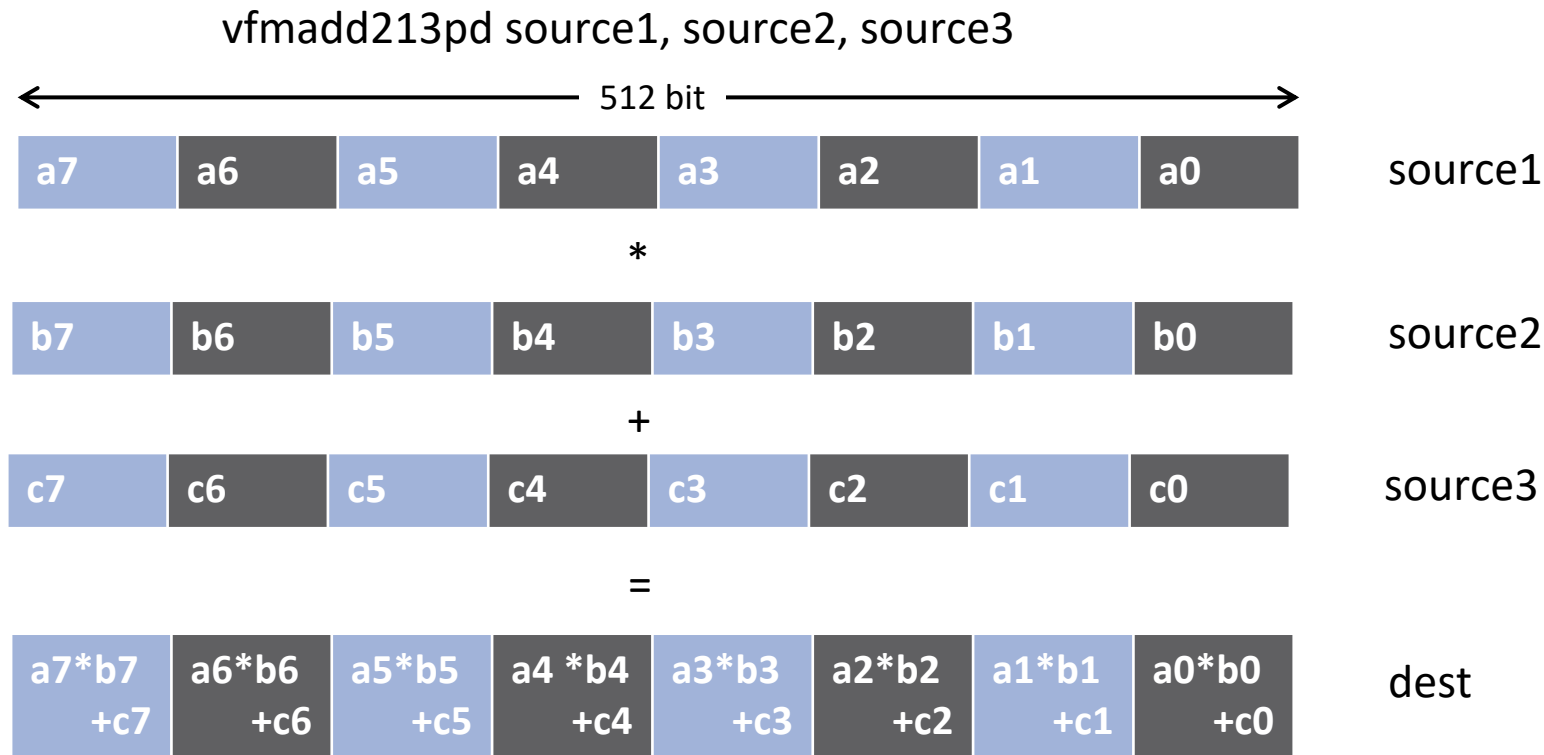
More Powerful SIMD Units

- SIMD instructions become more powerful
- One example is the Intel® Xeon Phi™ Coprocessor



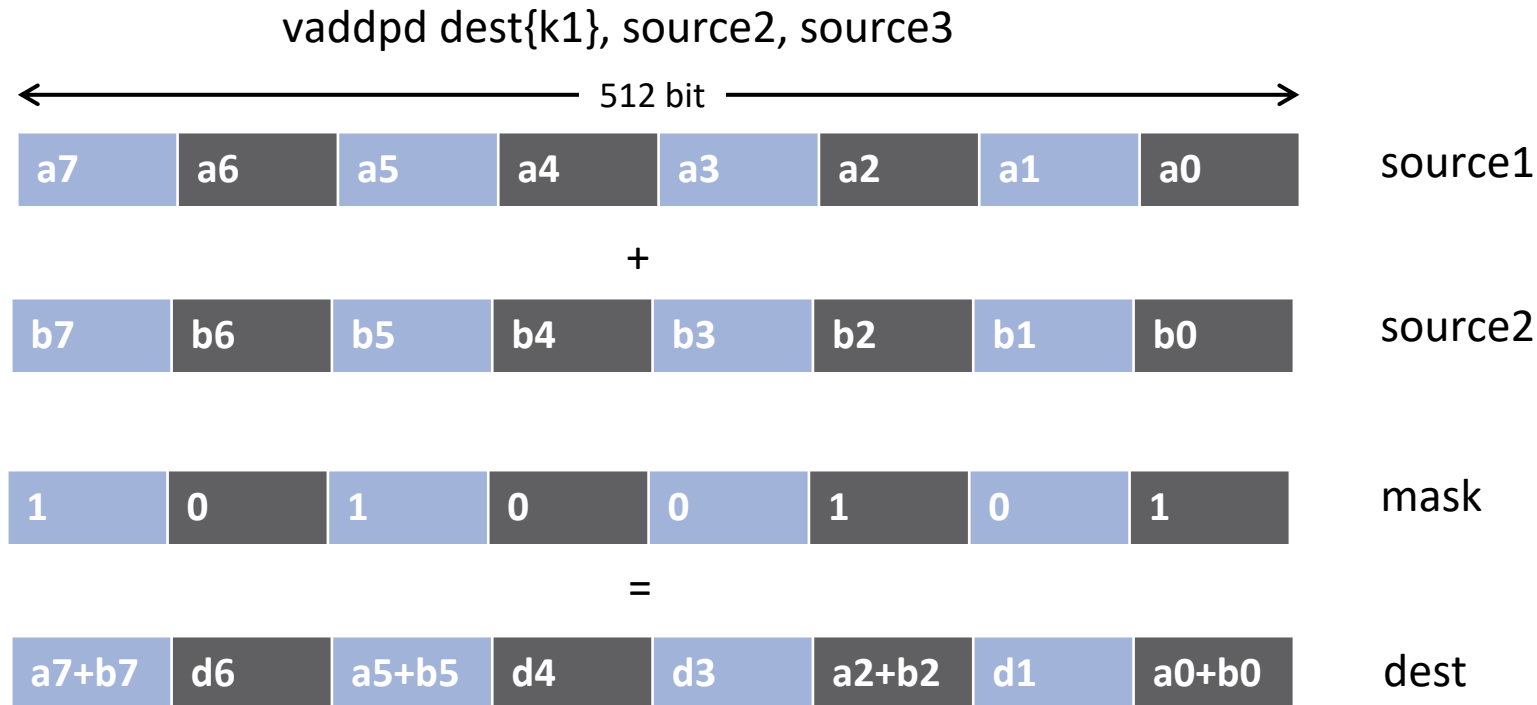
More Powerful SIMD Units

- SIMD instructions become more powerful
- One example is the Intel® Xeon Phi™ Coprocessor



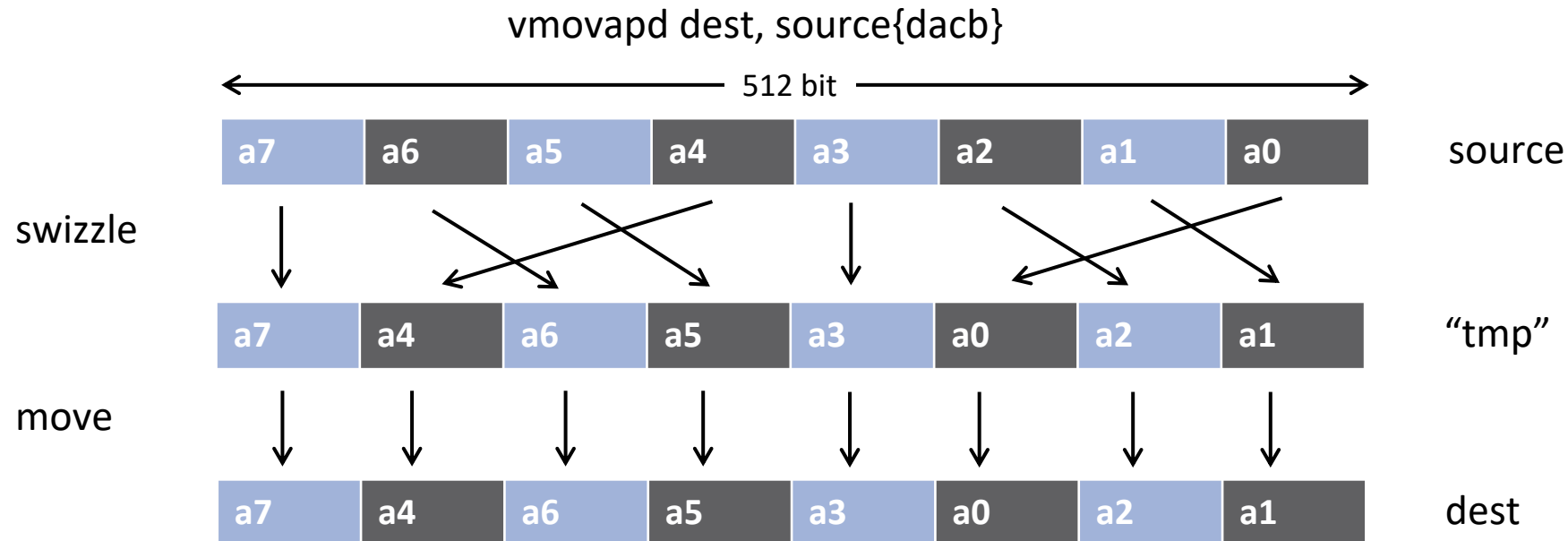
More Powerful SIMD Units

- SIMD instructions become more powerful
- One example is the Intel® Xeon Phi™ Coprocessor



More Powerful SIMD Units

- SIMD instructions become more powerful
- One example is the Intel® Xeon Phi™ Coprocessor



Auto-vectorization

- Compilers offer auto-vectorization as an optimization pass

- Usually part of the general loop optimization passes

- Code analysis detects code properties that inhibit SIMD vectorization



- Heuristics determine if SIMD execution might be beneficial

- If all goes well, the compiler will generate SIMD instructions

- Example: Intel® Composer XE

- -vec (automatically enabled with -O2)

- -qopt-report

Why Auto-vectorizers Fail

- **Data dependencies**
- **Other potential reasons**
 - Alignment
 - Function calls in loop block
 - Complex control flow / conditional branches
 - Loop not “countable”
 - e.g., upper bound not a runtime constant
 - Mixed data types
 - Non-unit stride between elements
 - Loop body too complex (register pressure)
 - Vectorization seems inefficient
- **Many more ... but less likely to occur**


Data Dependencies

- Suppose two statements S1 and S2
- S2 depends on S1, iff S1 must execute before S2
 - Control-flow dependence
 - Data dependence
 - Dependencies can be carried over between loop iterations
- Important flavors of data dependencies

FLOW

```


s1: a = 40
    b = 21
s2: c = a + 2
  
```



ANTI

```

    b = 40
s1: a = b + 1
s2: b = 21
  
```



Loop-Carried Dependencies

- Dependencies may occur across loop iterations

→ Loop-carried dependency

- The following code contains such a dependency:

```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2)
{
    size_t i;
    for (i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
    }
}
```

Loop-carried dependency for a[i] and a[i+17]; distance is 17.

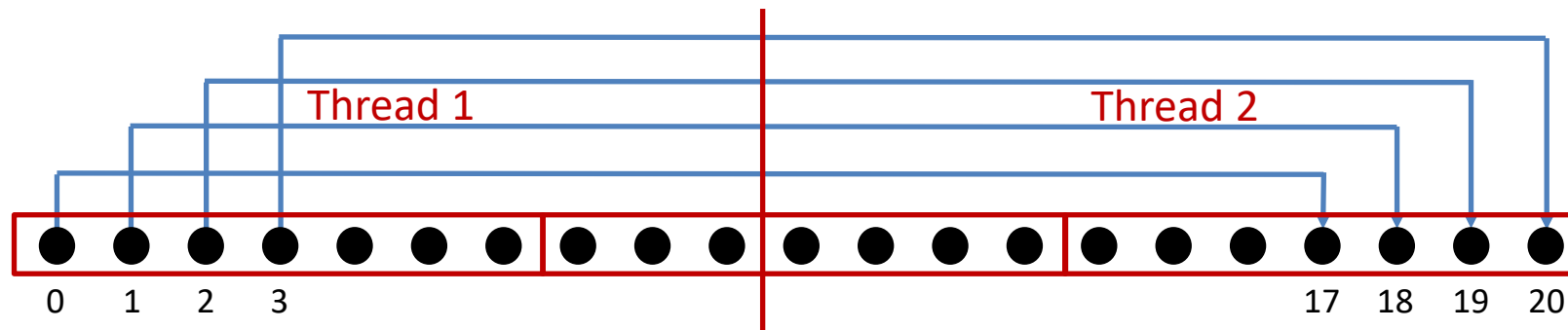
- Some iterations of the loop have to complete before the next iteration can run

→ Simple trick: Can you reverse the loop w/o getting wrong results?

Loop-carried Dependencies

■ Can we parallelize or vectorize the loop?

```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2) {
    for (int i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
    }
}
```



- Parallelization: no
(except for very specific loop schedules)
- Vectorization: yes
(iff vector length is shorter than any distance of any dependency)

Example: Loop not Countable

■ “Loop not Countable” plus “Assumed Dependencies”

```
typedef struct {
    float* data;
    size_t size;
} vec_t;

void vec_eltwise_product(vec_t* a, vec_t* b, vec_t* c) {
    size_t i;
    for (i = 0; i < a->size; i++) {
        c->data[i] = a->data[i] * b->data[i];
    }
}
```


In a Time Before OpenMP 4.0

■ Support required vendor-specific extensions

- Programming models (e.g., Intel® Cilk Plus)
- Compiler pragmas (e.g., `#pragma vector`)
- Low-level constructs (e.g., `_mm_add_pd()`)

```
#pragma omp parallel for
#pragma vector always
#pragma ivdep
for (int i = 0; i < N; i++) {
    a[i] = b[i] + ...;
}
```

You need to trust your compiler to do the “right” thing.

SIMD Loop Construct

■ Vectorize a loop nest

- Cut loop into chunks that fit a SIMD vector register
- No parallelization of the loop body

■ Syntax (C/C++)

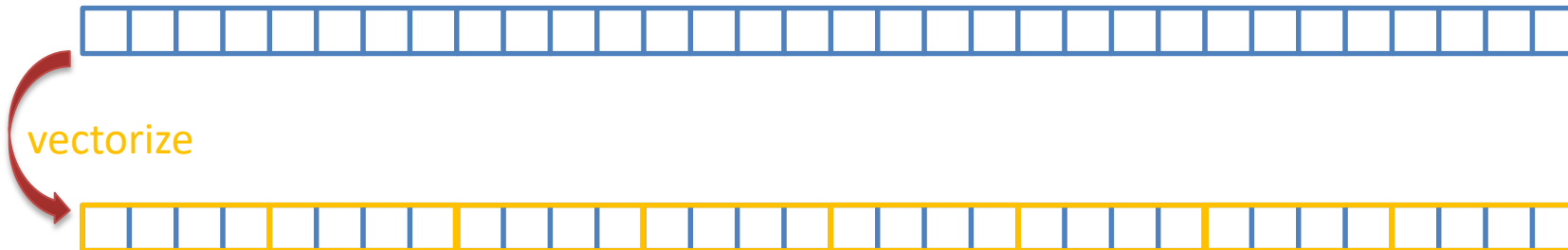
```
#pragma omp simd [clause[[, clause],...]  
for-loops
```

■ Syntax (Fortran)

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

Example

```
float sprod(float *a, float *b, int n) {  
    float sum = 0.0f;  
    #pragma omp simd reduction(+:sum)  
    for (int k=0; k<n; k++)  
        sum += a[k] * b[k];  
    return sum;  
}
```



Data Sharing Clauses

- `private (var-list) :`
Uninitialized vectors for variables in *var-list*



- `firstprivate (var-list) :`
Initialized vectors for variables in *var-list*



- `reduction (op: var-list) :`
Create private variables for *var-list* and apply reduction operator *op* at the end of the construct



SIMD Loop Clauses

■ `safelen (length)`

- Maximum number of iterations that can run concurrently without breaking a dependence
- In practice, maximum vector length

■ `linear (list[:linear-step])`

- The variable's value is in relationship with the iteration number
 - $x_i = x_{\text{orig}} + i * \text{linear-step}$

■ `aligned (list[:alignment])`

- Specifies that the list items have a given alignment
- Default is alignment for the architecture

■ `collapse (n)`

SIMD Worksharing Construct

■ Parallelize and vectorize a loop nest

- Distribute a loop's iteration space across a thread team
- Subdivide loop chunks to fit a SIMD vector register

■ Syntax (C/C++)

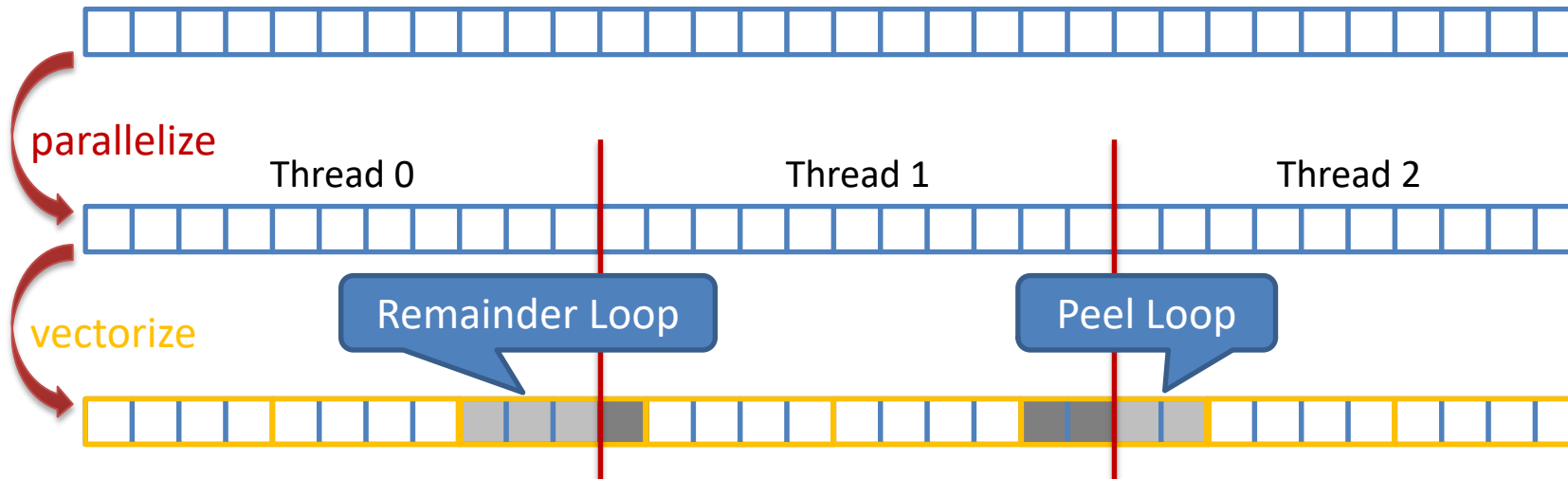
```
#pragma omp for simd [clause[[, clause],...]
for-loops
```

■ Syntax (Fortran)

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

Example

```
float sprod(float *a, float *b, int n) {  
    float sum = 0.0f;  
    #pragma omp for simd reduction(+:sum)  
    for (int k=0; k<n; k++)  
        sum += a[k] * b[k];  
    return sum;  
}
```



Be Careful What You Wish For...

```
float sprod(float *a, float *b, int n) {
    float sum = 0.0f;
    #pragma omp for simd reduction(+:sum) \
                               schedule(static, 5)
    for (int k=0; k<n; k++)
        sum += a[k] * b[k];
    return sum;
}
```

- You should choose chunk sizes that are multiples of the SIMD length
 - Remainder loops are not triggered
 - Likely better performance
- In the above example ...
 - and AVX2, the code will only execute the remainder loop!
 - and SSE, the code will have one iteration in the SIMD loop plus one in the remainder loop!

OpenMP 4.5 Simplifies SIMD Chunks

```
float sprood(float *a, float *b, int n) {  
    float sum = 0.0f;  
    #pragma omp for simd reduction(+:sum) \  
                                   schedule(simd: static, 5)  
    for (int k=0; k<n; k++)  
        sum += a[k] * b[k];  
    return sum;  
}
```

- Chooses chunk sizes that are multiples of the SIMD length
 - First and last chunk may be slightly different to fix alignment and to handle loops that are not exact multiples of SIMD width
 - Remainder loops are not triggered
 - Likely better performance

SIMD Function Vectorization

```
float min(float a, float b) {
    return a < b ? a : b;
}

float distsq(float x, float y) {
    return (x - y) * (x - y);
}

void example() {
#pragma omp parallel for simd
    for (i=0; i<N; i++) {
        d[i] = min(distsq(a[i], b[i]), c[i]);
    }
}
```

SIMD Function Vectorization

- Declare one or more functions to be compiled for calls from a SIMD-parallel loop

- Syntax (C/C++):

```
#pragma omp declare simd [clause[[, clause],...]  
[#pragma omp declare simd [clause[[, clause],...]]  
[...]  
function-definition-or-declaration
```

- Syntax (Fortran):

```
!$omp declare simd (proc-name-list)
```

SIMD Function Vectorization

```
#pragma omp declare simd
```

```
float min(float a, float b) {  
    return a < b ? a : b;  
}
```

```
_ZGVZN16vv_min(%zmm0, %zmm1):  
vminps %zmm1, %zmm0, %zmm0  
ret
```

```
#pragma omp declare simd
```

```
float distsq(float x, float y)  
    return (x - y) * (x - y);  
}
```

```
_ZGVZN16vv_distsq(%zmm0, %zmm1):  
vsubps %zmm0, %zmm1, %zmm2  
vmulps %zmm2, %zmm2, %zmm0  
ret
```

```
void example() {
```

```
#pragma omp parallel for simd
```

```
    for (i=0; i<N; i++) {  
        d[i] = min(distsq(a[i], b[i]), c[i]);  
    }  
}
```

```
vmovups (%r14,%r12,4), %zmm0  
vmovups (%r13,%r12,4), %zmm1  
call _ZGVZN16vv_distsq  
vmovups (%rbx,%r12,4), %zmm1  
call _ZGVZN16vv_min
```

SIMD Function Vectorization

- `simdlen (length)`
 - generate function to support a given vector length
- `uniform (argument-list)`
 - argument has a constant value between the iterations of a given loop
- `inbranch`
 - function always called from inside an if statement
- `notinbranch`
 - function never called from inside an if statement
- `linear (argument-list[:linear-step])`
- `aligned (argument-list[:alignment])`

inbranch & notinbranch

```
#pragma omp declare simd inbranch
```

```
float do_stuff(float x) {  
    /* do something */  
    return x * 2.0;  
}
```

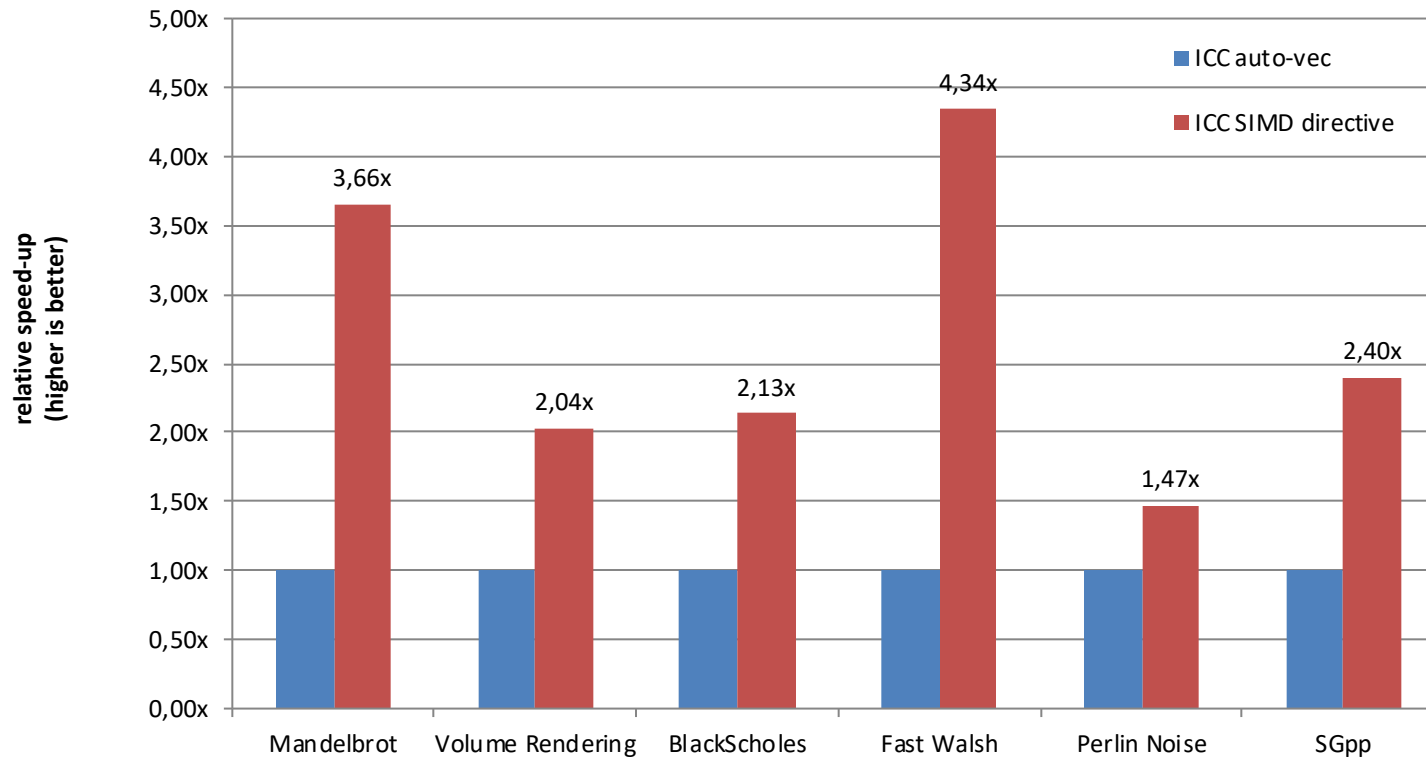
```
vec8 do_stuff_v(vec8 x, mask m) {  
    /* do something */  
    vmulpd x{m}, 2.0, tmp  
    return tmp;  
}
```

```
void example() {  
    #pragma omp simd
```

```
    for (int i = 0; i < N; i++)  
        if (a[i] < 0.0)  
            b[i] = do_stuff(a[i]);  
}
```

```
for (int i = 0; i < N; i+=8) {  
    vcmp_lt &a[i], 0.0, mask  
    b[i] = do_stuff_v(&a[i], mask);  
}
```

SIMD Constructs & Performance



M.Klemm, A.Duran, X.Tian, H.Saito, D.Caballero, and X.Martorell. Extending OpenMP with Vector Constructs for Modern Multicore SIMD Architectures. In Proc. of the Intl. Workshop on OpenMP, pages 59-72, Rome, Italy, June 2012. LNCS 7312.

OpenMP: Memory Access

Example: Loop Parallelization

- Assume the following: you have learned that *load imbalances* can severely impact performance and a *dynamic* loop schedule may prevent this:

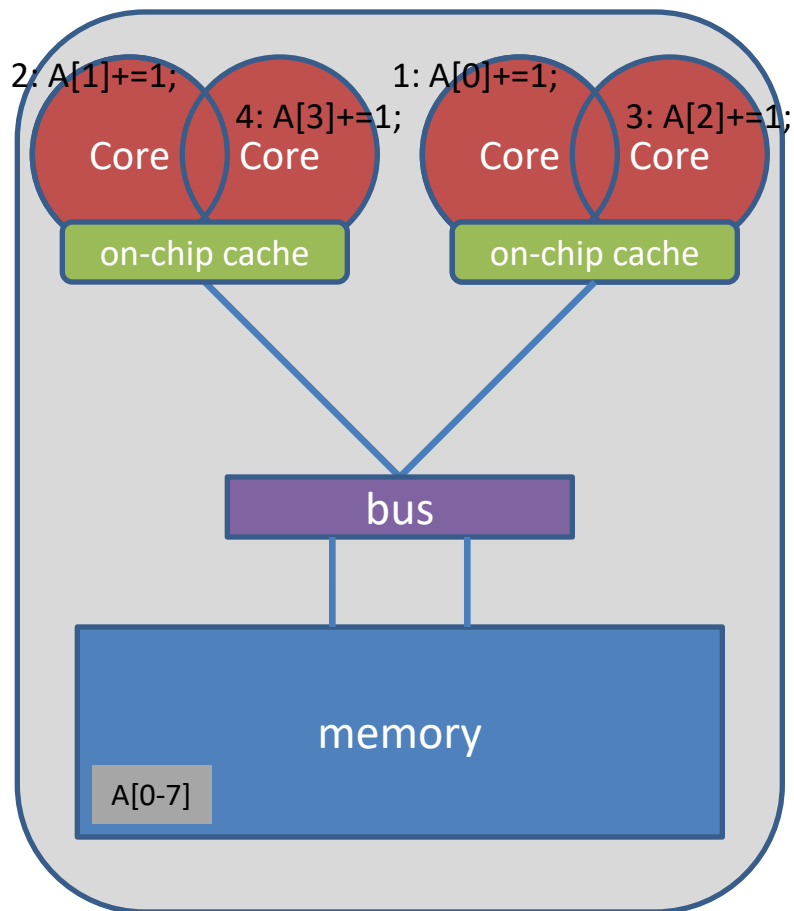
→ What is the issue with the following code:

```
double* A;  
A = (double*) malloc(N * sizeof(double));  
/* assume some initialization of A */  
  
#pragma omp parallel for schedule(dynamic, 1)  
for (int i = 0; i < N; i++) {  
    A[i] += 1.0;  
}
```

→ How is A accessed? Does that affect performance?

False Sharing

- **False Sharing: Parallel accesses to the same cache line may have a significant performance impact!**



Caches are organized in lines of typically 64 bytes: integer array a[0-4] fits into one cache line.

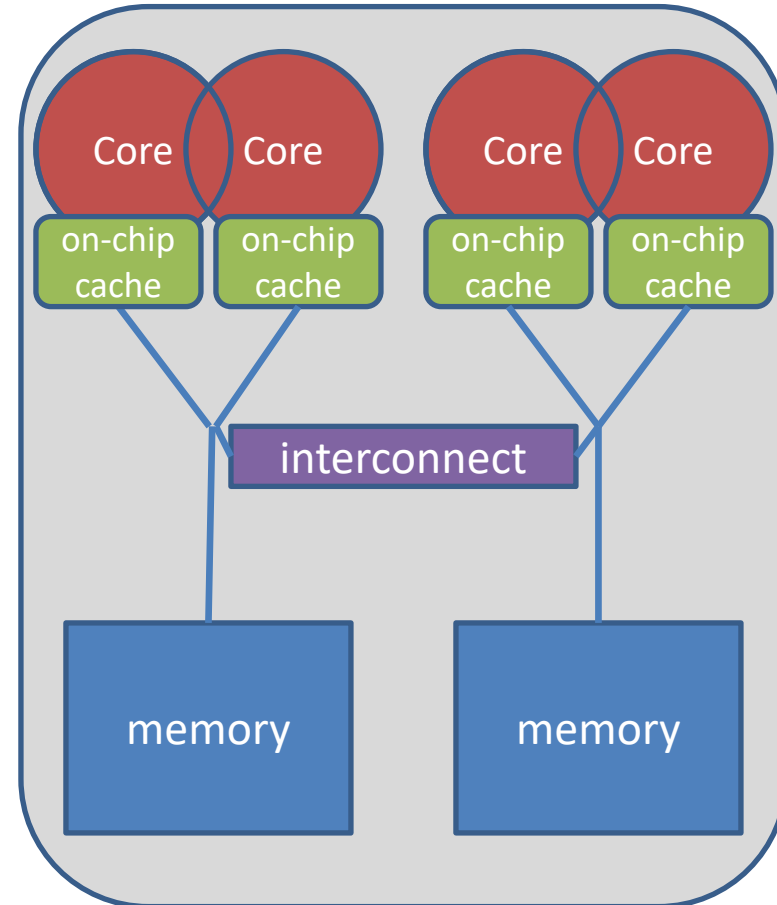
Whenever one element of a cache line is updated, the whole cache line is Invalidated.

Local copies of a cache line have to be re-loaded from the main memory and the computation may have to be repeated.

How To Distribute The Data ?

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

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

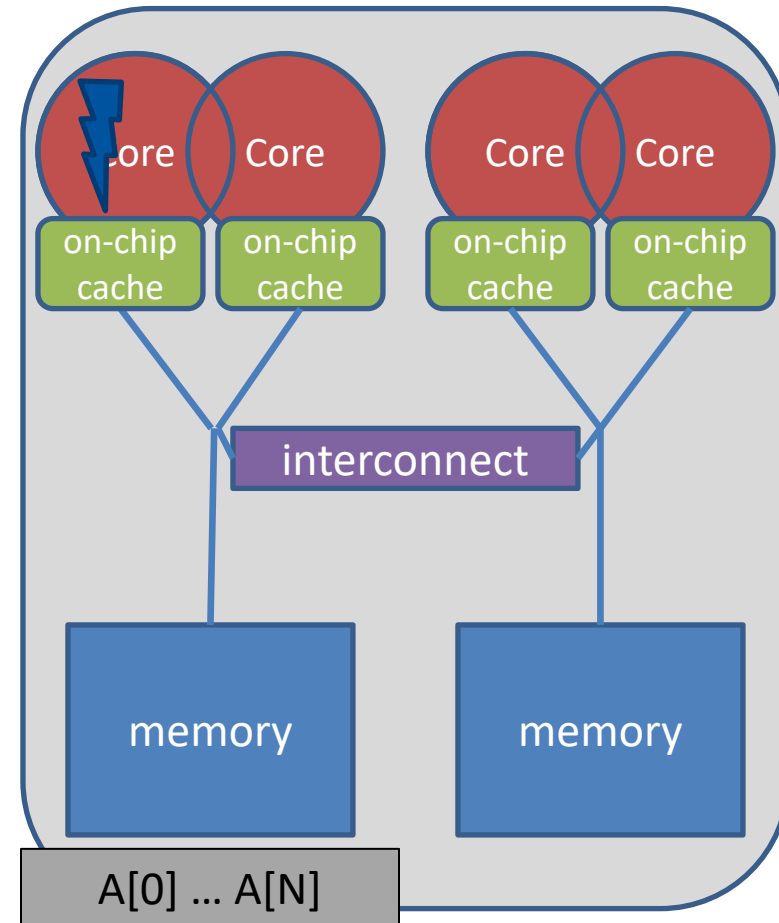


Non-uniform Memory

- 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

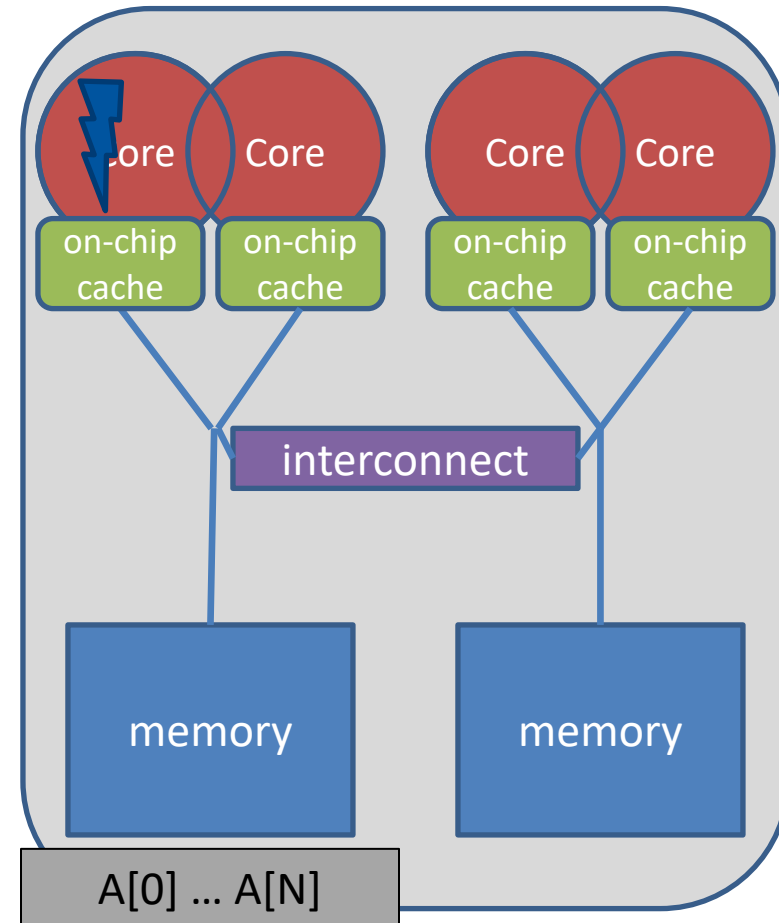
- Important aspect on cc-NUMA systems
 - If not optimal, longer memory access times and hotspots
- Placement comes from the Operating System
 - This is therefore Operating System dependent
- Windows, Linux and Solaris all use the “First Touch” placement policy by default
 - May be possible to override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

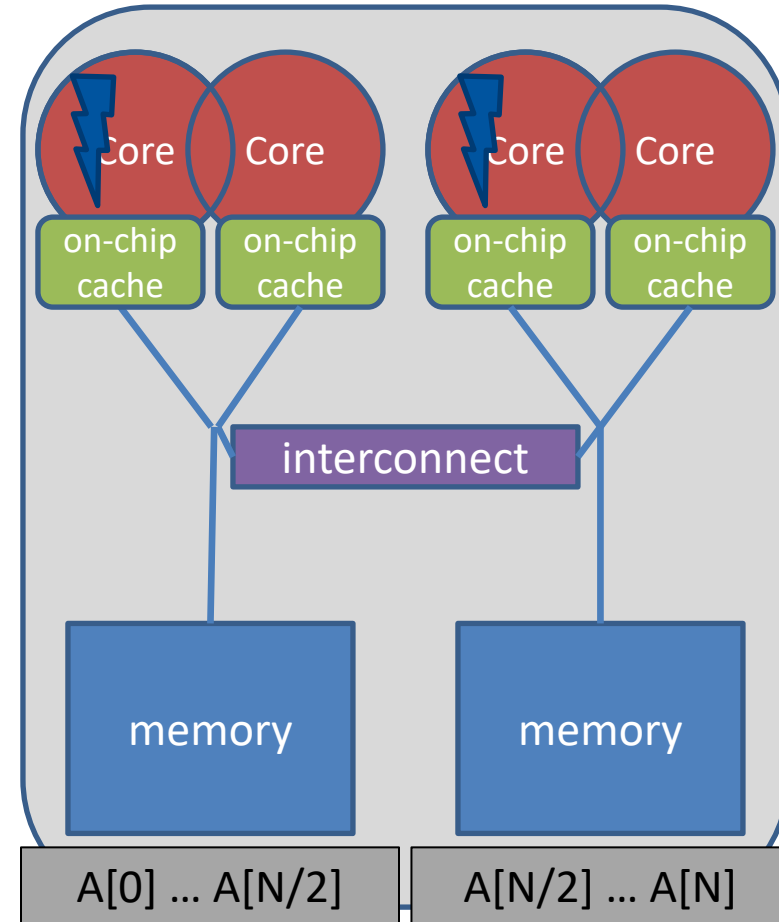
```
for (int i = 0; i < N; i++) {  
    A[i] = 0.0;  
}
```



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**

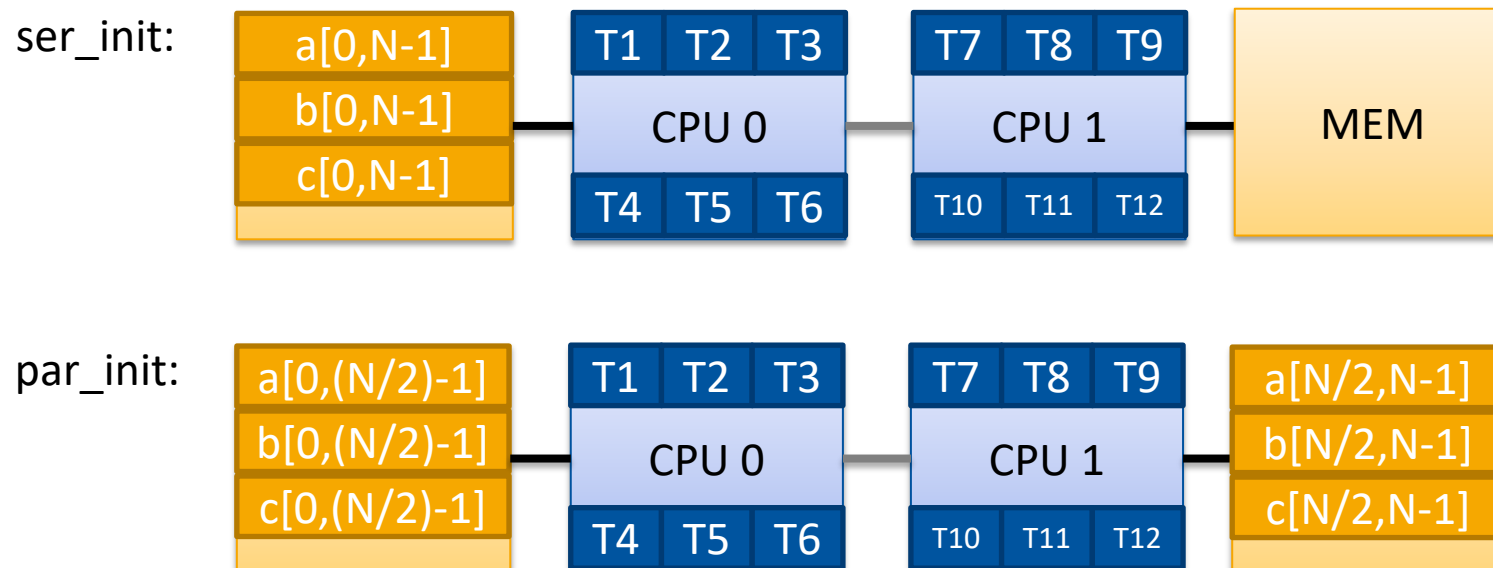
```
double* A;  
A = (double*)  
    malloc(N * sizeof(double));  
  
omp_set_num_threads(2);  
  
#pragma omp parallel for  
for (int i = 0; i < N; i++) {  
    A[i] = 0.0;  
}
```



Serial vs. Parallel Initialization

- Stream example on 2 socket system 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`

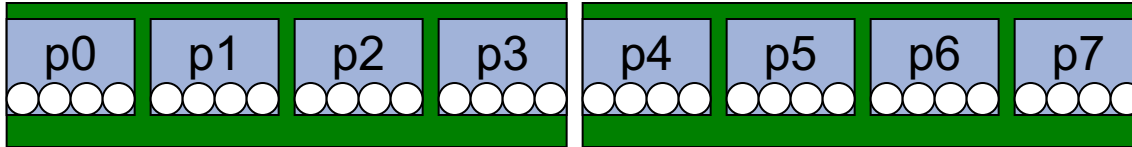
■ Define a set of OpenMP Thread Affinity Policies

- 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

- 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.

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

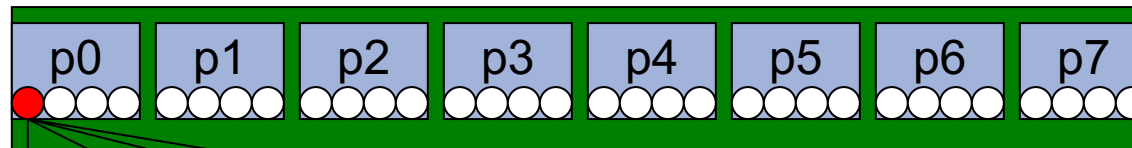
```
OMP_PLACES=(0,1,2,3), (4,5,6,7), ... = (0-3):8:4 = cores
```

```
#pragma omp parallel proc_bind(spread) num_threads(4)
```

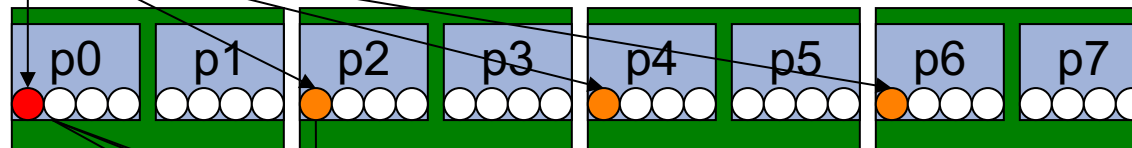
```
#pragma omp parallel proc_bind(close) num_threads(4)
```

■ Example

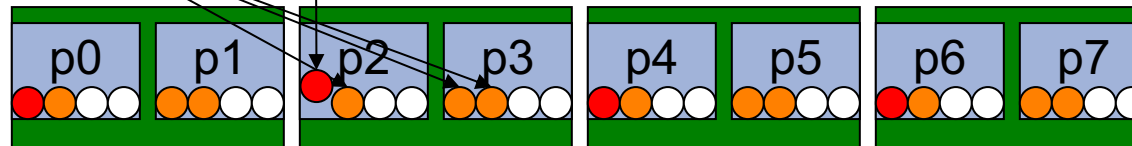
→ initial



→ spread 4

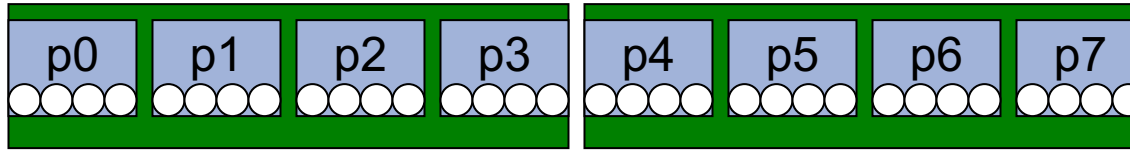


→ close 4



More Examples (1/3)

- Assume the following machine:



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

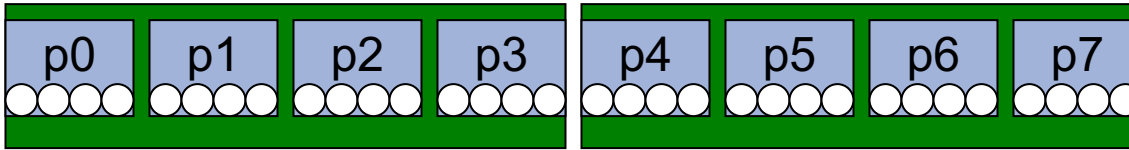
- Parallel Region with two threads, one per socket

→ `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)`

Places API (1/2)

- 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)

- 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

- 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()	a	omp_get_ancestor_thread_num() at level-1
T	omp_get_num_teams()	H	hostname
L	omp_get_level()	P	process identifier
n	omp_get_thread_num()	i	native thread identifier
N	omp_get_num_threads()	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

- 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 \geq 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.
- **Next Touch:** Binding of pages to NUMA nodes is removed and pages are migrated to the location of the next „touch“. Well-supported in Solaris, expensive to implement in Linux.
- **Automatic Migration:** No support for this in current operating 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`

Improving Tasking Performance: Task Affinity

- 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

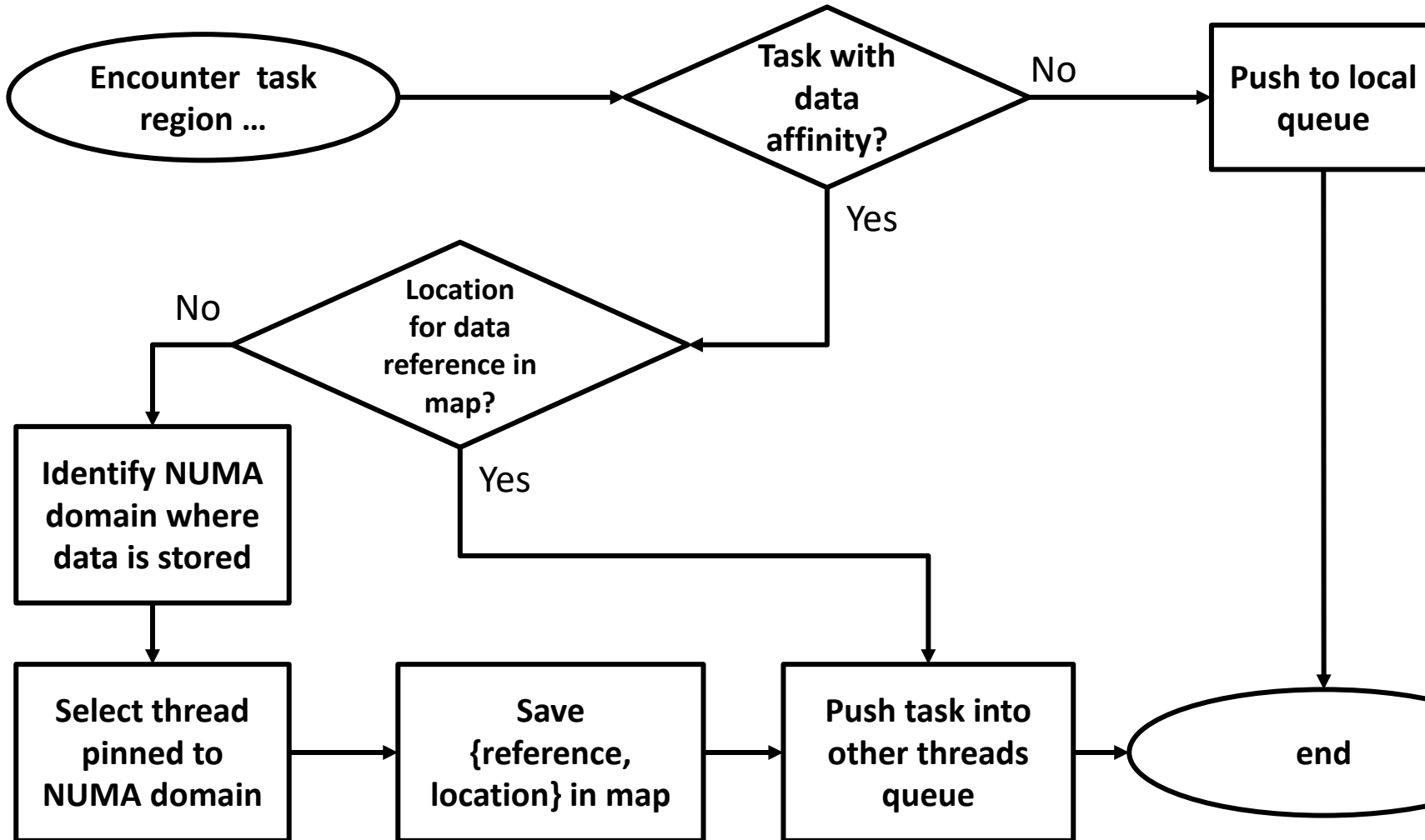
■ Excerpt from task-parallel STREAM

```
1  #pragma omp task \  
2      shared(a, b, c, scalar) \  
3      firstprivate(tmp_idx_start, tmp_idx_end) \  
4      affinity( a[tmp_idx_start] )  
5  {  
6      int i;  
7      for(i = tmp_idx_start; i <= tmp_idx_end; i++)  
8          a[i] = b[i] + scalar * c[i];  
9  }
```

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

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

Selected LLVM implementation details

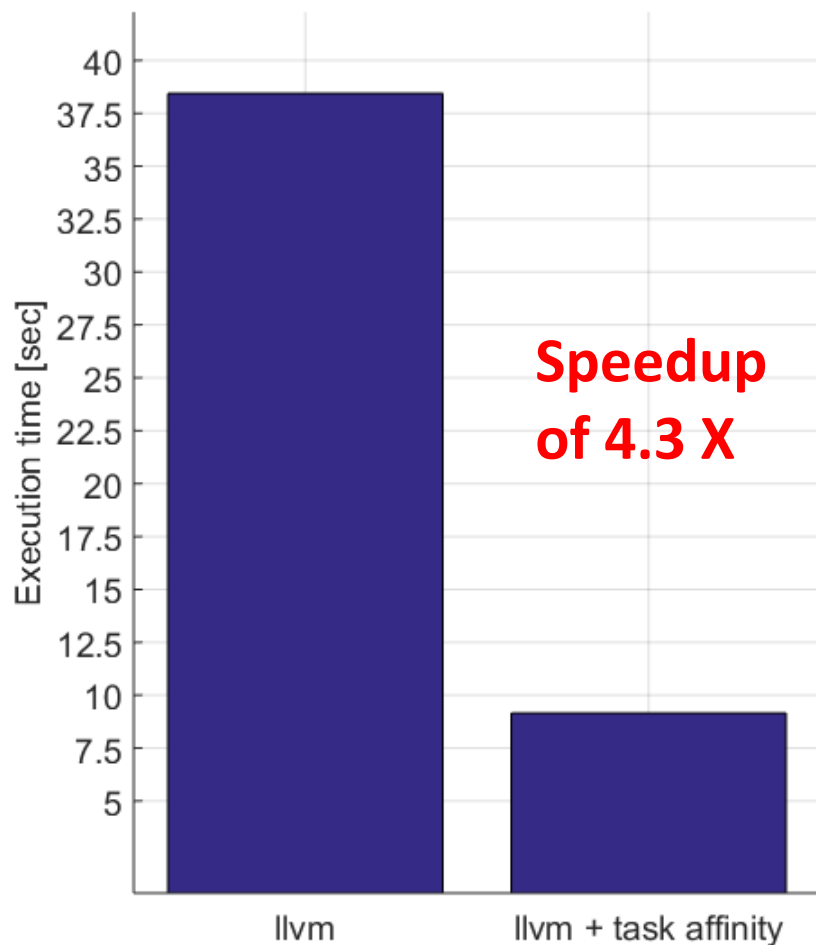


A map is introduced to store location information of data that was previously used

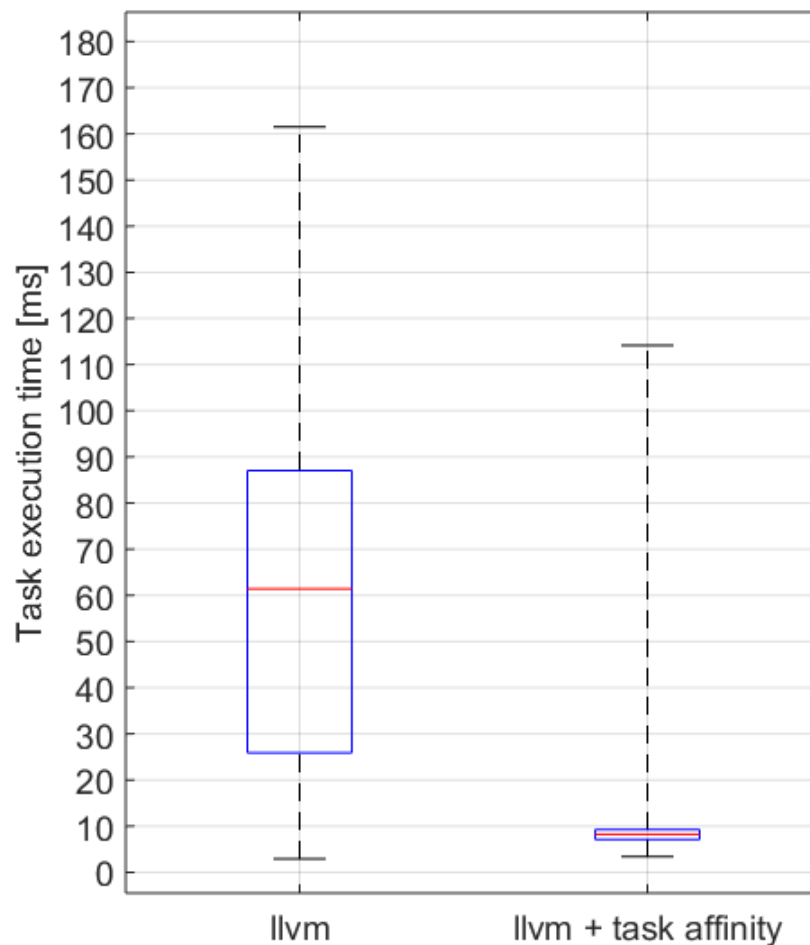
Jannis Klinkenberg, Philipp Samfass, Christian Terboven, Alejandro Duran, Michael Klemm, Xavier Teruel, Sergi Mateo, Stephen L. Olivier, and Matthias S. Müller. **Assessing Task-to-Data Affinity in the LLVM OpenMP Runtime.** Proceedings of the 14th International Workshop on OpenMP, IWOMP 2018. September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

Evaluation

Program runtime
Median of 10 runs



Distribution of single task execution times



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

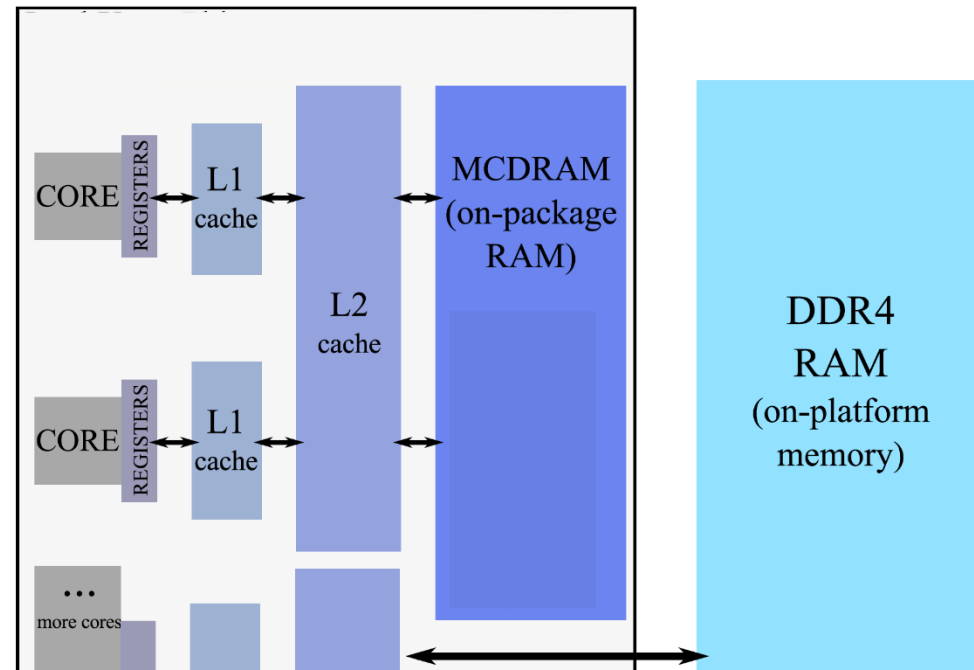
Summary

- 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

Managing Memory Spaces

Different kinds of memory

- Traditional DDR-based memory
- High-bandwidth memory
- Non-volatile memory
- ...



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

■ Selection of a certain kind of memory

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

- `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

- `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)
```


■ Storage resources with explicit support in OpenMP:

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

→ Exact selection of memory space is implementation-def.

→ Pre-defined allocators available to work with these



OPENMP OFFLOAD PROGRAMMING

Dr.-Ing. Michael Klemm

Principal Engineer

Extreme Computing SW and Systems

Chief Executive Officer

OpenMP Architecture Review Board

NOTICES AND DISCLAIMERS

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration.

No product or component can be absolutely secure.

Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. For more complete information about performance and benchmark results, visit <http://www.intel.com/benchmarks>.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit <http://www.intel.com/benchmarks>.

Intel Advanced Vector Extensions (Intel AVX)* provides higher throughput to certain processor operations. Due to varying processor power characteristics, utilizing AVX instructions may cause a) some parts to operate at less than the rated frequency and b) some parts with Intel Turbo Boost Technology 2.0 to not achieve any or maximum turbo frequencies. Performance varies depending on hardware, software, and system configuration and you can learn more at <http://www.intel.com/go/turbo>.

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Cost reduction scenarios described are intended as examples of how a given Intel-based product, in the specified circumstances and configurations, may affect future costs and provide cost savings. Circumstances will vary. Intel does not guarantee any costs or cost reduction.

Intel does not control or audit third-party benchmark data or the web sites referenced in this document. You should visit the referenced web site and confirm whether referenced data are accurate.

© Intel Corporation and OpenMP Architecture Review Board. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. OpenMP is a trademark of the OpenMP Architecture Review Board. Other names and brands may be claimed as the property of others.

Agenda

OpenMP Architecture Review Board

Introduction to OpenMP Offload Features

Case Study: NWChem TCE CCSD(T)

INTRODUCTION TO OPENMP OFFLOAD FEATURES

Running Example for this Presentation: saxpy

```
void saxpy() {  
    float a, x[SZ], y[SZ];  
    // left out initialization  
    double t = 0.0;  
    double tb, te;  
    tb = omp_get_wtime();  
#pragma omp parallel for firstprivate(a)  
    for (int i = 0; i < SZ; i++) {  
        y[i] = a * x[i] + y[i];  
    }  
    te = omp_get_wtime();  
    t = te - tb;  
    printf("Time of kernel: %lf\n", t);  
}
```

Timing code (not needed, just to have a bit more code to show 😊)

This is the code we want to execute on a target device (i.e., GPU)

Timing code (not needed, just to have a bit more code to show 😊)

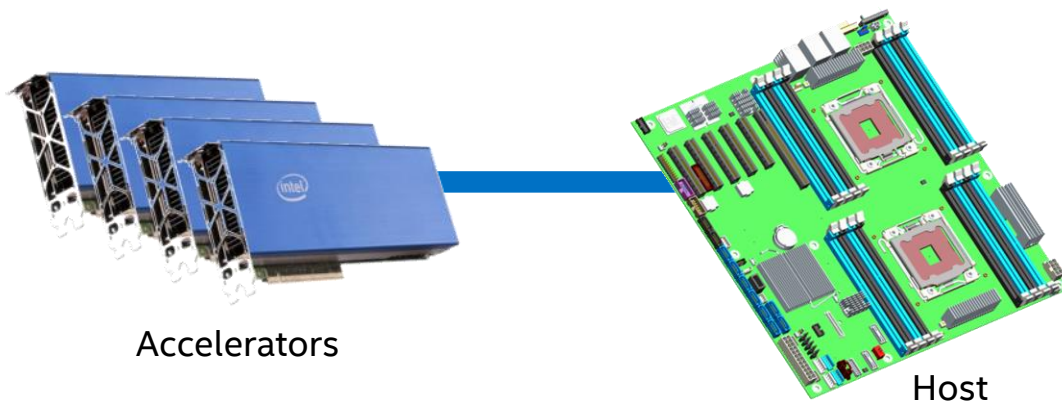
Don't do this at home! Use a BLAS library for this!

Device Model

As of version 4.0 the OpenMP API supports accelerators/coprocessors

Device model:

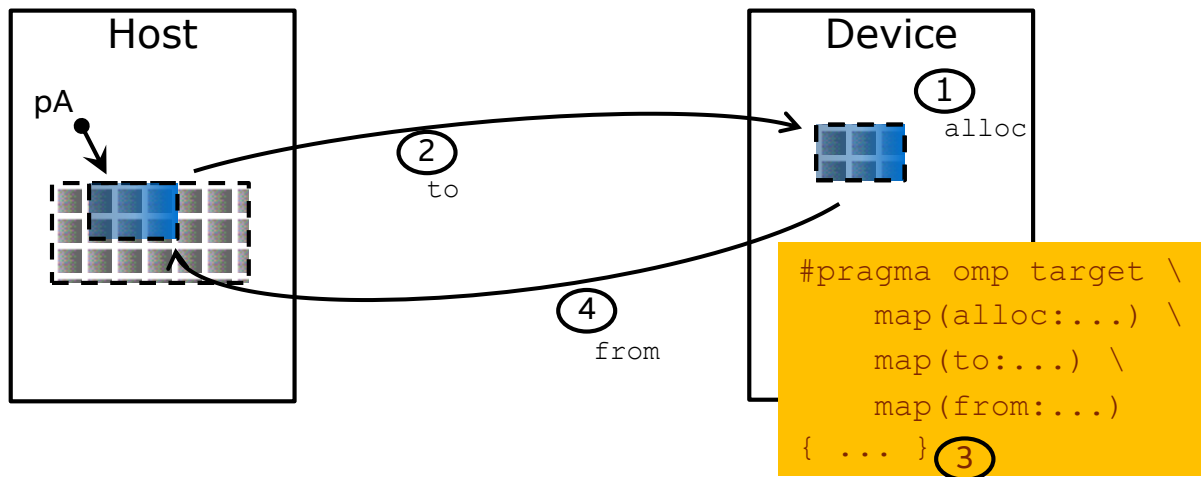
- One host for “traditional” multi-threading
- Multiple accelerators/coprocessors of the same kind for offloading



Execution Model

Offload region and data environment is lexically scoped

- Data environment is destroyed at closing curly brace
- Allocated buffers/data are automatically released



OpenMP for Devices - Constructs

Transfer control and data from the host to the device

Syntax (C/C++)

```
#pragma omp target [clause[[,] clause],...]  
structured-block
```

Syntax (Fortran)

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

Clauses

```
device(scalar-integer-expression)  
map([{alloc | to | from | tofrom}:] list)  
if(scalar-expr)
```

Example: saxpy

```
void saxpy() {  
    float a, x[SZ], y[SZ];  
    double t = 0.0;  
    double tb, te;  
    tb = omp_get_wtime();  
    #pragma omp target "map(tofrom:y[0:SZ])"  
    for (int i = 0; i < SZ; i++) {  
        y[i] = a * x[i] + y[i];  
    }  
    te = omp_get_wtime();  
    t = te - tb;  
    printf("Time of kernel: %lf\n", t);  
}
```

The compiler identifies variables that are used in the target region.

All accessed arrays are copied from host to device and back

a
x[0:SZ]
y[0:SZ]

Presence check: only transfer if not yet allocated on the device.

x[0:SZ]
y[0:SZ]

Copying x back is not necessary: it was not changed.

icc -qnextgen -fioopenmp -fopenmp-targets=spir64 -o axpy axpy.c

Example: saxpy

```
subroutine saxpy(a, x, y, n)
  use iso_fortran_env
  integer :: n, i
  real(kind=real32) :: a
  real(kind=real32), dimension(n) :: x
  real(kind=real32), dimension(n) :: y

  !$omp target "map(tofrom:y(1:n))"
  do i=1,n
    y(i) = a * x(i) + y(i)
  end do
  !$omp end target
end subroutine
```

The compiler identifies variables that are used in the target region.

All accessed arrays are copied from host to device and back

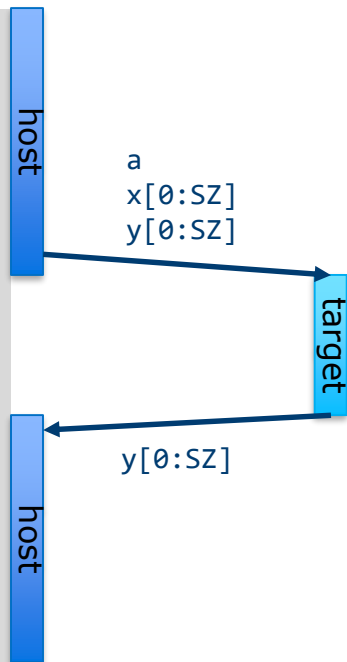
Presence check: only transfer if not yet allocated on the device.

Copying x back is not necessary: it was not changed.

```
ifort -qnextgen -fiopenmp -fopenmp-targets=spir64 -o axpy axpy.f90
```

Example: saxpy

```
void saxpy() {  
    double a, x[SZ], y[SZ];  
    double t = 0.0;  
    double tb, te;  
    tb = omp_get_wtime();  
    #pragma omp target map(to:x[0:SZ]) \  
                        map(tofrom:y[0:SZ])  
    for (int i = 0; i < SZ; i++) {  
        y[i] = a * x[i] + y[i];  
    }  
    te = omp_get_wtime();  
    t = te - tb;  
    printf("Time of kernel: %lf\n", t);  
}
```

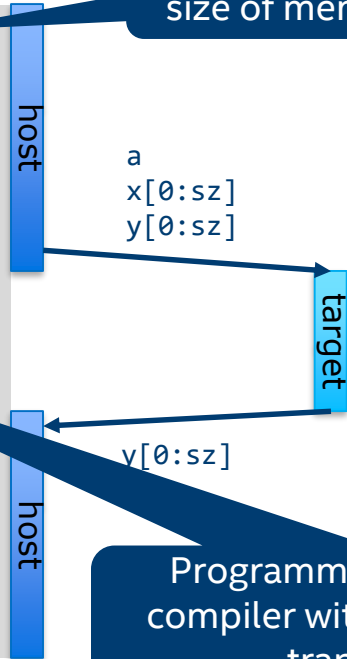


```
icc -qnextgen -fiopenmp -fopenmp-targets=spir64 -o axpy axpy.c
```

Example: saxpy

```
void saxpy(float a, float* x, float* y,
           int sz) {
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
    #pragma omp target map(to:x[0:sz]) \
                      map(tofrom:y[0:sz])
    for (int i = 0; i < sz; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```

The compiler cannot determine the size of memory behind the pointer.



Programmers have to help the compiler with the size of the data transfer needed.

```
icc -qnextgen -fiopenmp -fopenmp-targets=spir64 -o axpy axpy.c
```

Creating Parallelism on the Target Device

The `target` construct transfers the control flow to the target device

- Transfer of control is sequential and synchronous
- This is intentional!

OpenMP separates offload and parallelism

- Programmers need to explicitly create parallel regions on the target device
- In theory, this can be combined with any OpenMP construct
- In practice, there is only a useful subset of OpenMP features for a target device such as a GPU, e.g., no I/O, limited use of base language features.

Example: saxpy

```
void saxpy(float a, float* x, float* y,
           int sz) {
    #pragma omp target map(to:x[0:sz]) \
                    map(tofrom(y[0:sz]))
    #pragma omp parallel for simd
    for (int i = 0; i < sz; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

host

target

host

Create a team of threads to execute the loop in parallel using SIMD instructions.

GPUs are multi-level devices:
SIMD, threads, thread blocks

```
icc -qnextgen -fopenmp -fopenmp-targets=spir64 -o axpy axpy.c
```

teams Construct

Support multi-level parallel devices

Syntax (C/C++):

```
#pragma omp teams [clause[[,] clause],...]  
structured-block
```

Syntax (Fortran):

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

Clauses

```
num_teams(integer-expression), thread_limit(integer-expression)  
default(shared | firstprivate | private none)  
private(list), firstprivate(list), shared(list), reduction(operator:list)
```


Multi-level Parallel saxpy

Manual code transformation

- Tile the loops into an outer loop and an inner loop
- Assign the outer loop to “teams” (OpenCL: work groups)
- Assign the inner loop to the “threads” (OpenCL: work items)

```
void saxpy(float a, float* x, float* y, int sz) {  
#pragma omp target teams map(to:x[0:sz]) map(tofrom:y[0:sz])  
    {  
        int bs = n / omp_get_num_teams();  
#pragma omp distribute  
        for (int i = 0; i < sz; i += bs) {  
#pragma omp parallel for simd firstprivate(i,bs)  
            for (int ii = i; ii < i + bs; ii++) {  
                y[ii] = a * x[ii] + y[ii];  
            }  
        }  
    }  
}
```

Multi-level Parallel saxpy

For convenience, OpenMP defines composite constructs to implement the required code transformations

```
void saxpy(float a, float* x, float* y, int sz) {  
    #pragma omp target teams distribute parallel for simd \  
        num_teams(num_blocks) map(to:x[0:sz]) map(tofrom:y[0:sz])  
    for (int i = 0; i < sz; i++) {  
        y[i] = a * x[i] + y[i];  
    }  
}
```

```
subroutine saxpy(a, x, y, n)  
    ! Declarations omitted  
    !$omp target teams distribute parallel do simd &  
    !$omp&        num_teams(num_blocks) map(to:x) map(tofrom:y)  
    do i=1,n  
        y(i) = a * x(i) + y(i)  
    end do  
    !$omp end target teams distribute parallel do simd  
end subroutine
```

Profiling and Debugging Environment Variables

```
nuc1 .../axpy> LIBOMPTARGET_DEBUG=1 ./axpy
Libomptarget --> Loading RTLS...
Libomptarget --> Loading library 'libomptarget.rtl.nios2.so'...
Libomptarget --> Unable to load library 'libomptarget.rtl.nios2.so': libomptarget.rtl.nios2.so: cannot open shared object file: No such file or directory!
Libomptarget --> Loading library 'libomptarget.rtl.x86_64_mic.so'...
Libomptarget --> Unable to load library 'libomptarget.rtl.x86_64_mic.so': libomptarget.rtl.x86_64_mic.so: cannot open shared object file: No such file or directory!
Libomptarget --> Loading library 'libomptarget.rtl.openc1.so'...
Libomptarget --> Successfully loaded library 'libomptarget.rtl.openc1.so'!
Libomptarget --> Optional interface: __tgt_rtl_data_submit_nowait
Libomptarget --> Optional interface: __tgt_rtl_data_retrieve_nowait
Libomptarget --> Optional interface: __tgt_rtl_manifest_data_for_region
Libomptarget --> Optional interface: __tgt_rtl_data_alloc_base
Libomptarget --> Optional interface: __tgt_rtl_data_alloc_user
Libomptarget --> Optional interface: __tgt_rtl_run_target_team_nd_region
Libomptarget --> Optional interface: __tgt_rtl_run_target_region_nowait
Libomptarget --> Optional interface: __tgt_rtl_run_target_team_region_nowait
Libomptarget --> Optional interface: __tgt_rtl_run_target_team_nd_region_nowait
Libomptarget --> Registering RTL libomptarget.rtl.openc1.so supporting 2 devices!
nuc1 .../axpy> LIBOMPTARGET_PROFILE=T,msec ./axpy
LIBOMPTARGET_PROFILE:
-- DATA-READ: 0.065 msec
-- DATA-WRITE: 0.188 msec
-- EXEC-__omp_offloading_34_2fc1e3f8_main_l113: 1.470 msec
-- EXEC-__omp_offloading_34_2fc1e3f8_saxpy_offload_target_data_env_l86: 0.063 msec
-- EXEC-__omp_offloading_34_2fc1e3f8_saxpy_offload_target_l71: 0.070 msec
-- EXEC-__omp_offloading_34_2fc1e3f8_saxpy_offload_target_l54: 0.105 msec
```

Optimize Data Transfers

Reduce the amount of time spent transferring data

- Use map clauses to enforce direction of data transfer
- Use target data, target enter data, target exit data constructs to keep data environment on the target device (see backup for syntax)

```
void example() {  
    float tmp[N], data_in[N], float data_out[N];  
    #pragma omp target data map(alloc:tmp[:N]) \  
        map(to:a[:N],b[:N]) \  
        map(tofrom:c[:N])  
  
    {  
        zeros(tmp, N);  
        compute_kernel_1(tmp, a); // uses target  
        saxpy(2.0f, tmp, b);  
        compute_kernel_2(tmp, b); // uses target  
        saxpy(2.0f, c, tmp);  
    }  
}
```

```
void zeros(float* a, int n) {  
    #pragma omp target teams distribute parallel for  
    for (int i = 0; i < n; i++)  
        a[i] = 0.0f;  
}
```

```
void saxpy(float a, float* y, float* x, int n) {  
    #pragma omp target teams distribute parallel for  
    for (int i = 0; i < n; i++)  
        y[i] = a * x[i] + y[i];  
}
```

target data Construct Syntax

Create scoped data environment and transfer data from the host to the device and back

Syntax (C/C++)

```
#pragma omp target data [clause[[,] clause],...]  
structured-block
```

Syntax (Fortran)

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

Clauses

```
device(scalar-integer-expression)  
map([{alloc | to | from | tofrom | release | delete}]:] list)  
if(scalar-expr)
```

target update Construct Syntax

Issue data transfers to or from existing data device environment

Syntax (C/C++)

```
#pragma omp target update [clause[[,] clause],...]
```

Syntax (Fortran)

```
!$omp target update [clause[[,] clause],...]
```

Clauses

device(*scalar-integer-expression*)

to(*list*)

from(*list*)

if(*scalar-expr*)

Example: target data and target update

```
#pragma omp target data device(0) map(alloc:tmp[:N]) map(to:input[:N]) map(from:res)
{
#pragma omp target device(0)
#pragma omp parallel for
    for (i=0; i<N; i++)
        tmp[i] = some_computation(input[i], i);

    update_input_array_on_the_host(input);

#pragma omp target update device(0) to(input[:N])

#pragma omp target device(0)
#pragma omp parallel for reduction(+:res)
    for (i=0; i<N; i++)
        res += final_computation(input[i], tmp[i], i)
}
```

host

target

host

target

host

Asynchronous Offloads

OpenMP target constructs are synchronous by default

- The encountering host thread awaits the end of the target region before continuing
- The `nowait` clause makes the target constructs asynchronous (in OpenMP speak: they become an OpenMP task)

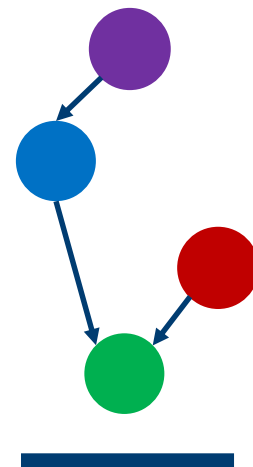
```
#pragma omp task                                depend(out:in1)
  init_data(in1);

#pragma omp target map(to:in1[:N]) map(from:out1[:N]) nowait depend(in:in1) depend(out:out1)
  compute_1(in1, out1, N);

#pragma omp target map(to:in2[:N]) map(from:out3[:N]) nowait depend(out:out2)
  compute_3(in2, out3, N);

#pragma omp target map(to:out2[:N]) map(to:out3[:N]) nowait depend(in:out1) depend(in:out2)
  compute_4(out1, out2, N);

#pragma omp taskwait
```



CASE STUDY: NWCHEM TCE CCSD(T)

TCE: Tensor Contraction Engine

CCSD(T): Coupled-Cluster with Single, Double, and perturbative Triple replacements

NWChem

Computational chemistry software package

- Quantum chemistry
- Molecular dynamics

Designed for large-scale supercomputers

Developed at the EMSL at PNNL

- EMSL: Environmental Molecular Sciences Laboratory
- PNNL: Pacific Northwest National Lab

URL: <http://www.nwchem-sw.org>

Finding Offload Candidates

Requirements for offload candidates

- Compute-intensive code regions (kernels)
- Highly parallel
- Compute scaling stronger than data transfer, e.g., compute $O(n^3)$ vs. data size $O(n^2)$

Example Kernel (1 of 27 in total)

```
subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,  
1          h7d,triplexx,t2sub,v2sub)  
c  Declarations omitted.  
double precision triplexx(h3d*h2d,h1d,p6d,p5d,p4d)  
double precision t2sub(h7d,p4d,p5d,h1d)  
double precision v2sub(h3d*h2d,p6d,h7d)  
!$omp target „presence?(triplexx,t2sub,v2sub)”  
!$omp teams distribute parallel do private(p4,p5,p6,h2,h3,h1,h7)  
do p4=1,p4d  
do p5=1,p5d  
do p6=1,p6d  
do h1=1,h1d  
do h7=1,h7d  
do h2h3=1,h3d*h2d  
triplexx(h2h3,h1,p6,p5,p4)=triplexx(h2h3,h1,p6,p5,p4)  
1  - t2sub(h7,p4,p5,h1)*v2sub(h2h3,p6,h7)  
end do  
end do  
end do  
end do  
end do  
end do  
!$omp end teams distribute parallel do  
!$omp end target  
end subroutine
```

1.5GB data transferred
(host to device)

1.5GB data transferred
(device to host)

All kernels expose the same structure

7 perfectly nested loops

Some kernels contain inner product loop
(then, 6 perfectly nested loops)

Trip count per loop is equal to “tile size”
(20-30 in production)

Naïve data allocation (tile size 24)

- Per-array transfer for each target construct
- triplexx: 1458 MB
- t2sub, v2sub: 2.5 MB each

Invoking the Kernels / Data Management

Simplified pseudo-code of the actual

```
!$omp target enter data alloc(triplexx(1:tr_size))
c   for all tiles
    do ...
      call zero_triplexx(triplexx)
      do ...
        call comm_and_sort(t2sub, v2sub)
!$omp target data map(to:t2sub(t2_size)) map(to:v2sub(v2_size))
      if (...)
        call sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,h4,h7,triplexx,t2sub,v2sub)
      end if
c     same for sd_t_d1_2 until sd_t_d1_9
!$omp target end data
    end do
    do ...
c     Similar structure for sd_t_d2_1 until sd_t_d2_9, incl. target data
    end do
    call sum_energy(energy, triplexx)
  end do
!$omp target exit data release(triplexx(1:size))
```

Allocate 1.5GB data once, stays on device.

Update 2x2.5MB of data for (potentially) multiple kernels.

Reduced data transfers:

- triplexx:
 - allocated once
 - always kept on the target
- t2sub, v2sub:
 - allocated after comm.
 - kept for (multiple) kernel invocations

Invoking the Kernels / Data Management

Simplified pseudo-code of the actual

```
!$omp target enter data alloc(triplexx(1:tr_size))
c   for all tiles
do ...
  call zero_triplexx(triplexx)
do ...
  call comm_and_sort(t2sub, v2sub)
!$omp target data map(to:t2sub(t2_size)) map(to:v2sub(v2_size))
  if (...)
    call sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,h7,triplexx)
  end if
c   same for sd_t_d1_2 until sd_t_d1_9
!$omp target end data
end do
do ...
c   Similar structure for sd_t_d2_1 until sd_t_d2_9, inc
end do
  call sum_energy(energy, triplexx)
end do
!$omp target exit data release(triplexx(1:size))
```

Allocate 1
once, stays

Update 2x2.5
(potentially n

```
subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
1         h7d,triplexx,t2sub,v2sub)
  Declarations omitted.
  double precision triplexx(h3d*h2d,h1d,p6d,p5d,p4d)
  double precision t2sub(h7d,p4d,p5d,h1d)
  double precision v2sub(h3d*h2d,p6d,h7d)
!$omp target „presence?(triplexx,t2sub,v2sub)“
!$omp teams distribute parallel do private(p4,p5,p6,h2,h3,h1,h7)
  do p4=1,p4d
  do p5=1,p5d
  do p6=1,p6d
  do h1=1,h1d
  do h7=1,h7d
  do h2h3=1,h3d
    triplexx(h2h3,
1    - t2sub(h7
  end do
  end do
  end do
  end do
  end do
  end do
!$omp end teams distribute parallel do
!$omp end target
end subroutine
```

Presence check determines that
arrays have been allocated in the
device data environment already.

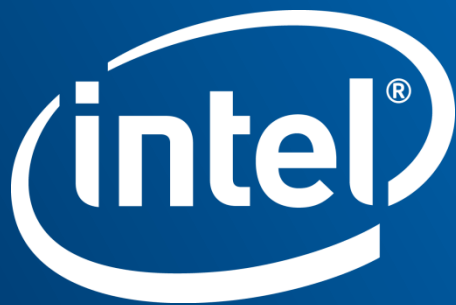
Summary

OpenMP API is ready to use Intel discrete GPUs for offloading compute

- Mature offload model w/ support for asynchronous offload/transfer
- Tightly integrates with OpenMP multi-threading on the host

More, advanced features (not covered here)

- Memory management API
- Interoperability with native data management
- Interoperability with native streaming interfaces
- Unified shared memory support



Tools for OpenMP Programming

OpenMP Tools

■ Correctness Tools

→ ThreadSanitizer

→ Intel Inspector XE (or whatever the current name is)

■ Performance Analysis

→ Performance Analysis basics

→ Overview on available tools

Data Race

- Data Race: the typical OpenMP programming error, when:
 - two or more threads access the same memory location, and
 - at least one of these accesses is a write, and
 - the accesses are not protected by locks or critical regions, and
 - the accesses are not synchronized, e.g. by a barrier.
- Non-deterministic occurrence: e.g. the sequence of the execution of parallel loop iterations is non-deterministic
 - In many cases *private* clauses, *barriers* or *critical regions* are missing
- Data races are hard to find using a traditional debugger

ThreadSanitizer: Overview

- Correctness checking for threaded applications
- Integrated in clang and gcc compiler
- Low runtime overhead: 2x – 15x
- Used to find data races in browsers like Chrome and Firefox

ThreadSanitizer: Usage

```
module load clang
```

Module in Aachen.

<https://pruners.github.io>

Compile the program with clang compiler:

```
clang -fsanitize=thread -fopenmp -g myprog.c -o myprog
```

```
clang++ -fsanitize=thread -fopenmp -g myprog.cpp  
-o myprog
```

```
gfortran -fsanitize=thread -fopenmp -g myprog.f -c
```

```
clang -fsanitize=thread -fopenmp -lgfortran myprog.o  
-o myprog
```

- Execute:

```
OMP_NUM_THREADS=4 ./myprog
```

- Understand and correct the detected threading errors

ThreadSanitizer: Example

```
1 #include <stdio.h>
2
3 int main(int argc, char **argv) {
4     int a = 0;
5     #pragma omp parallel
6     {
7         if (a < 100) {
8             #pragma omp critical
9             a++;
10        }
11    }
12 }
```

WARNING: ThreadSanitizer: data race

Read of size 4 at 0x7fffffffddcd by thread T2:
#0 .omp_outlined. race.c:7
(race+0x0000004a6dce)
#1 __kmp_invoke_microtask <null>
(libomp_tsan.so)

Previous write of size 4 at 0x7fffffffddcd by main thread:
#0 .omp_outlined. race.c:9
(race+0x0000004a6e2c)
#1 __kmp_invoke_microtask <null>
(libomp_tsan.so)

■ Detection of

→ Memory Errors

→ Deadlocks

→ Data Races

■ Support for

→ WIN32-Threads, Posix-Threads, Intel Threading Building Blocks and OpenMP

■ Features

→ Binary instrumentation gives full functionality

→ Independent stand-alone GUI for Windows and Linux



PI example / 1

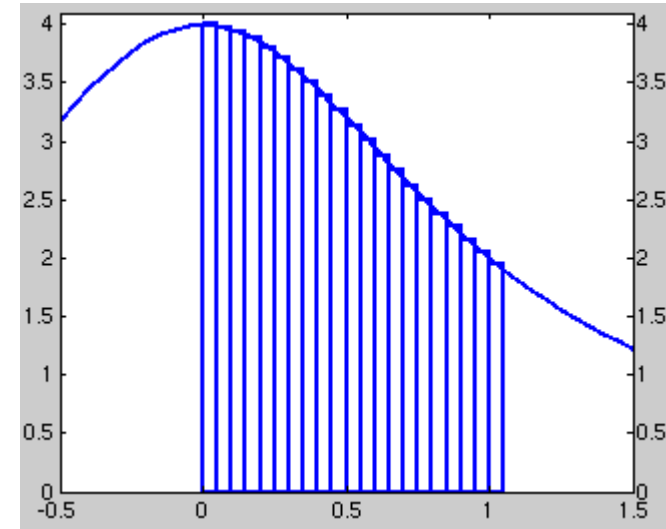
```
double f(double x)
{
    return (4.0 / (1.0 + x*x));
}
```

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

```

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

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



PI example / 2

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

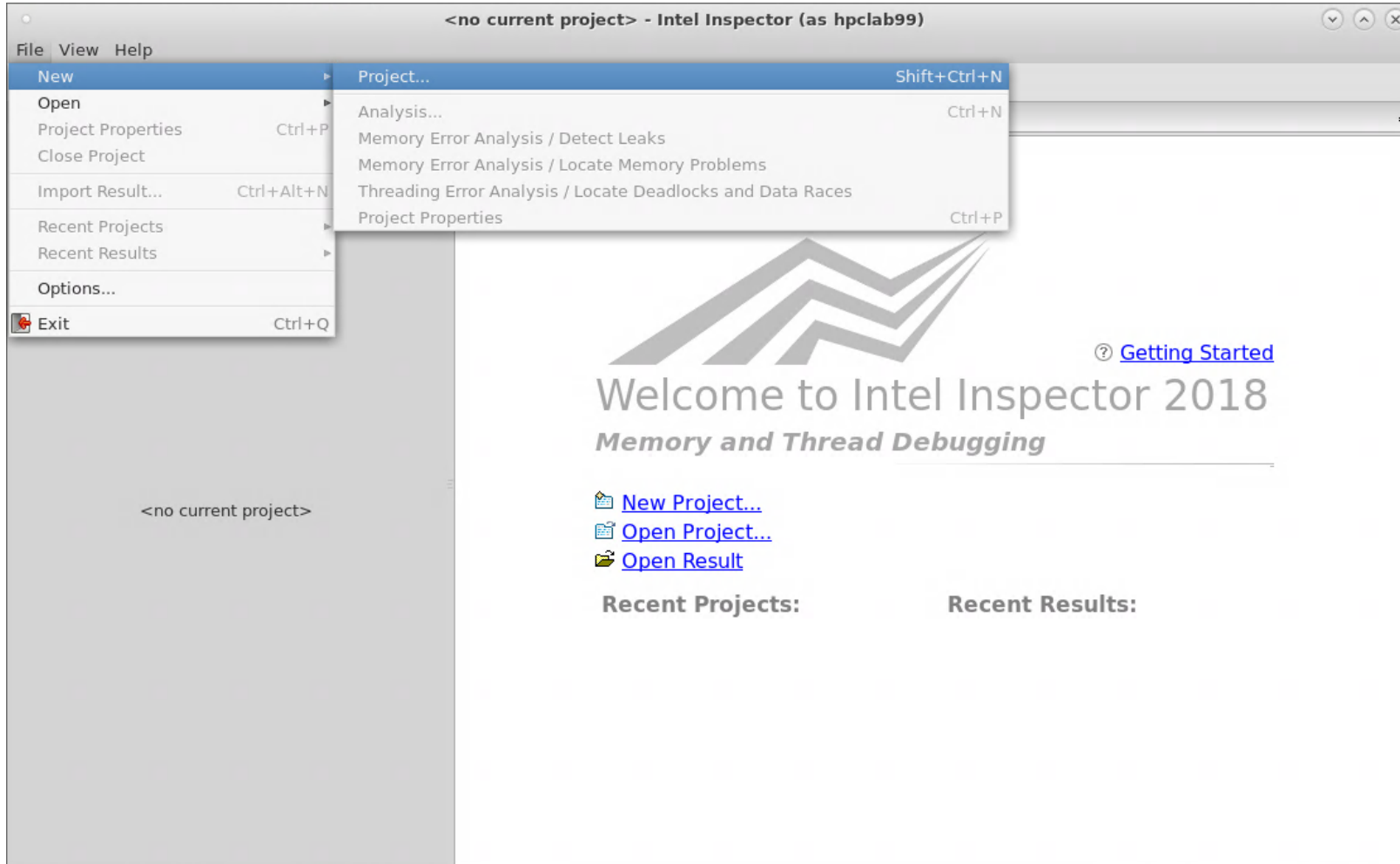
double CalcPi (int n)
{
    const double fH = 1.0 / (double) n;
    double fSum = 0.0;
    double fX;
    int i;

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

What if we
would have
forgotten this?

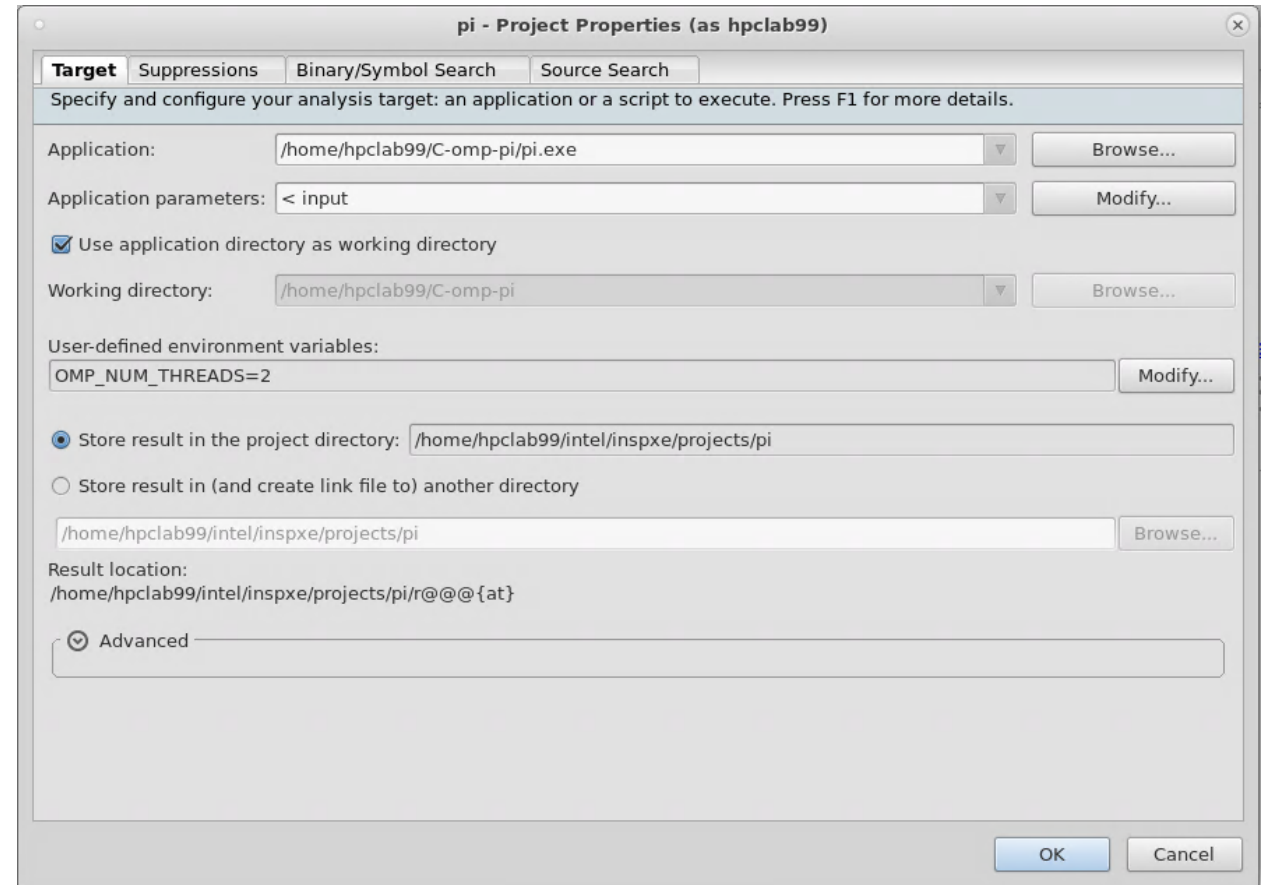
Inspector XE: create project / 1

```
$ module load Inspector ; inspxe-gui
```



Inspector XE: create project / 2

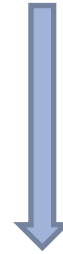
- ensure that multiple threads are used
- choose a small dataset (really!), execution time can increase 10X – 1000X



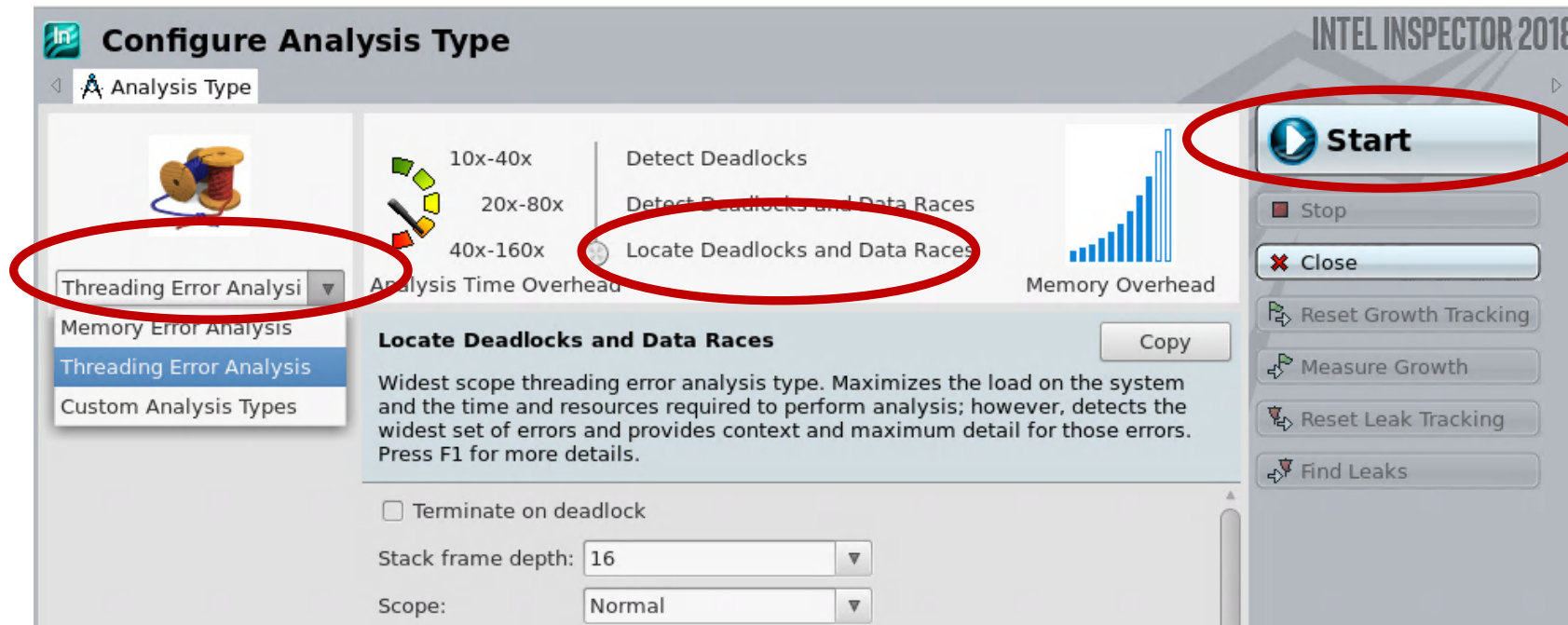
Inspector XE: configure analysis

Threading Error Analysis Modes

1. Detect Deadlocks
2. Detect Deadlocks and Data Races
3. Locate Deadlocks and Data Races



more details,
more overhead



Inspector XE: results / 1

- 1 detected problems
- 2 filters
- 3 code location
- 4 Timeline

The screenshot displays the Intel Inspector 2018 interface. The main window title is "/home/hpclab99/intel/inspxe/projects/pi - Intel Inspector (as hpclab99)". The interface is divided into several panes:

- Problems Table:** A table listing detected issues. The first row is highlighted and has a yellow circle '1' next to it. The table has columns for ID, Type, Sources, Modules, and State.
- Filters Panel:** Located on the right, it shows filters for Severity (Error), Type (Data race), Source (pi.c), Module (pi.exe), and State (New). A yellow circle '2' is next to the Severity filter.
- Code Locations Panel:** Located at the bottom left, it shows the source code for the detected data race. It has columns for Description, Source, Function, Module, and Variable. A yellow circle '3' is next to the code snippet.
- Timeline Panel:** Located at the bottom right, it shows the execution timeline for OMP Master Thread #0 (23581) and OMP Worker Thread #1 (23717). A yellow circle '4' is next to the timeline.

ID	Type	Sources	Modules	State
P1	Data race	pi.c	pi.exe	New
	Data race	pi.c:72	pi.exe	New
	Data race	pi.c:72	pi.exe	New

Severity	Count
Error	1 item(s)

Type	Count
Data race	1 item(s)

Source	Count
pi.c	1 item(s)

Module	Count
pi.exe	1 item(s)

State	Count
New	1 item(s)

Description	Source	Function	Module	Variable
Read	pi.c:72	CalcPi	pi.exe	
<pre>70 { 71 fX = fH * ((double)i + 0 72 fSum += f(fX); 73 } 74 return fH * fSum;</pre>				
Write	pi.c:72	CalcPi	pi.exe	
<pre>70 { 71 fX = fH * ((double)i + 0 72 fSum += f(fX); 73 } 74 return fH * fSum;</pre>				

Timeline: OMP Master Thread #0 (23581), OMP Worker Thread #1 (23717)

Inspector XE: results / 2

- 1 Source Code producing the issue – double click opens an editor
- 2 Corresponding Call Stack

Data race INTEL INSPECTOR 2018

Target Analysis Type Collection Log Summary Sources

Read - Thread OMP Master Thread #0 (23581) (pi.exe!CalcPi - pi.c:72)

```

pi.c Disassembly (pi.exe!0x111f)
67 //#pragma omp parallel for private(i, fX) reduction(+:fSum)
68 #pragma omp parallel for private(i, fX)
69   for (i = iRank; i < n; i += iNumProcs)
70   {
71       fX = fH * ((double)i + 0.5);
72       fSum += f(fX);
73   }
74   return fH * fSum;
75 }
76
    
```

Call Stack

```

pi.exe!CalcPi - pi.c:72
pi.exe!CalcPi - pi.c:68
pi.exe!_start
    
```

Write - Thread OMP Worker Thread #1 (23717) (pi.exe!CalcPi - pi.c:72)

```

pi.c Disassembly (pi.exe!0x1395)
67 //#pragma omp parallel for private(i, fX) reduction(+:fSum)
68 #pragma omp parallel for private(i, fX)
69   for (i = iRank; i < n; i += iNumProcs)
70   {
71       fX = fH * ((double)i + 0.5);
72       fSum += f(fX);
73   }
74   return fH * fSum;
75 }
76
    
```

Call Stack

```

pi.exe!CalcPi - pi.c:72
    
```

Inspector XE: results / 3

- 1 Source Code producing the issue – double click opens an editor
- 2 Corresponding Call Stack

The missing reduction is detected.

Data race

Target Analysis Type Collection Log Summary Sources

Read - Thread OMP Master Thread #0 (23581) (pi.exe!CalcPi - pi.c:72)

pi.c Disassembly (pi.exe!0x111f)

```

67 //#pragma omp parallel for private(i, fX) reduction(+:fSum)
68 #pragma omp parallel for private(i, fX)
69   for (i = iRank; i < n; i += iNumProcs)
70   {
71     fX = fH * ((double)i + 0.5);
72     fSum += f(fX);
73   }
74   return fH * fSum;
75 }
76

```

Call Stack

- pi.exe!CalcPi - pi.c:72
- pi.exe!CalcPi - pi.c:68
- pi.exe!_start

Write - Thread OMP Worker Thread #1 (23717) (pi.exe!CalcPi - pi.c:72)

pi.c Disassembly (pi.exe!0x1395)

```

67 //#pragma omp parallel for private(i, fX) reduction(+:fSum)
68 #pragma omp parallel for private(i, fX)
69   for (i = iRank; i < n; i += iNumProcs)
70   {
71     fX = fH * ((double)i + 0.5);
72     fSum += f(fX);
73   }
74   return fH * fSum;
75 }
76

```

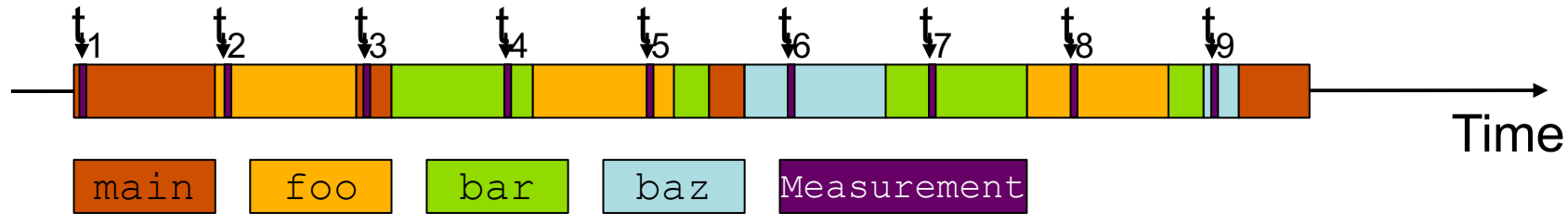
Call Stack

- pi.exe!CalcPi - pi.c:72

Sampling vs. Instrumentation

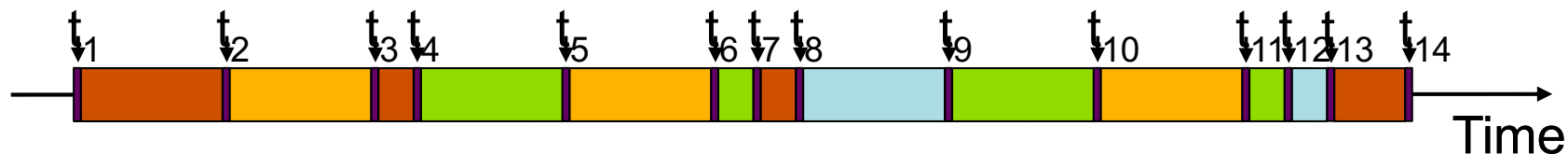
Sampling

- Running program is periodically interrupted to take measurement
- Statistical* inference of program behavior
- Works with unmodified executables



Instrumentation

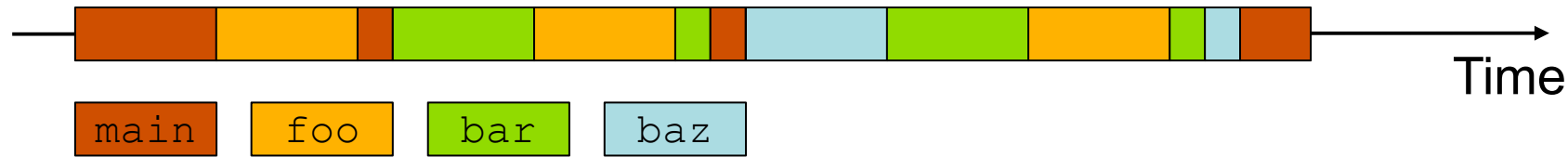
- Every event of interest is captured directly
- More detailed and *exact* information
- Typically: recompile for instrumentation



Tracing vs. Profiling

Trace

- Chronologically ordered sequence of event records

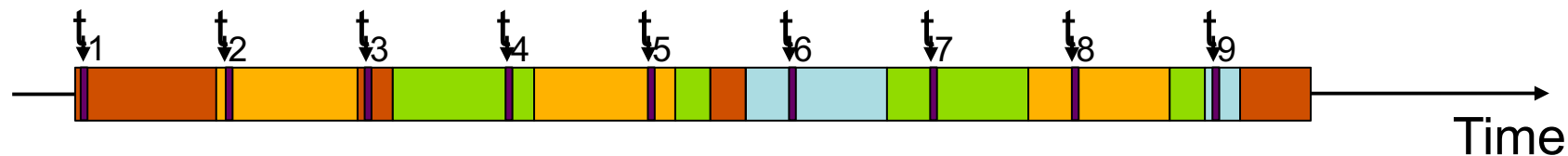


Profile from instrumentation

- Aggregated information



Profile from sampling



OMPT support for sampling

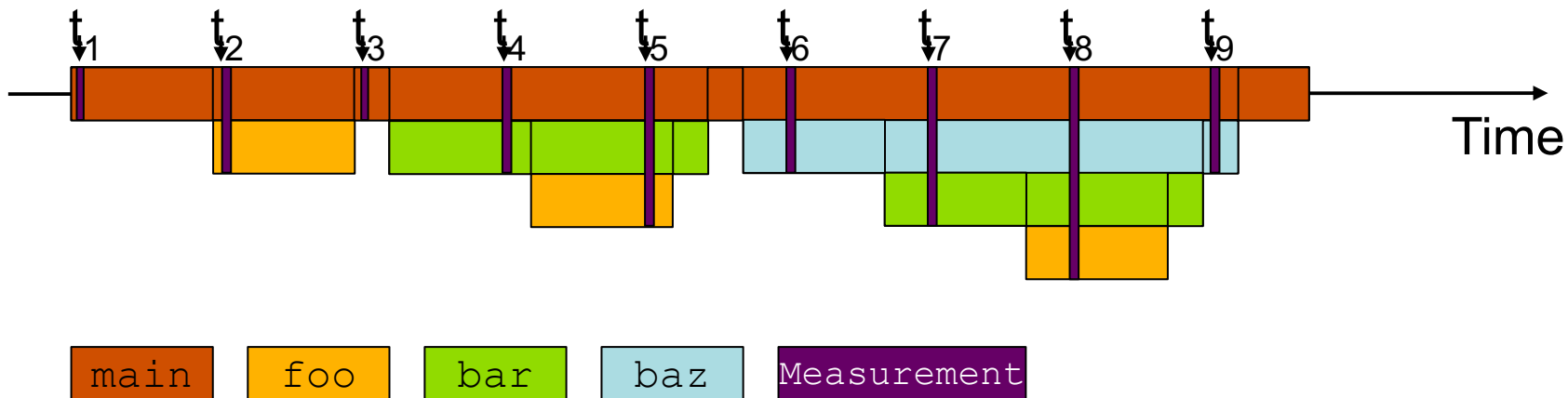
- OMPT defines states like *barrier-wait*, *work-serial* or *work-parallel*

- Allows to collect OMPT state statistics in the profile
- Profile break down for different OMPT states

```
void foo() {}
void bar() {foo();}
void baz() {bar();}
int main()
{foo();bar();baz();
 return 0;}
```

- OMPT provides frame information

- Allows to identify OpenMP runtime frames.
- Runtime frames can be eliminated from call trees



OMPT support for instrumentation

- OMPT provides event callbacks
 - Parallel begin / end
 - Implicit task begin / end
 - Barrier / taskwait
 - Task create / schedule
- Tool can instrument those callbacks
- OpenMP-only instrumentation might be sufficient for some use-cases

```
void foo() {}  
void bar() {  
    #pragma omp task  
    foo();}  
void baz() {  
    #pragma omp task  
    bar();}  
int main() {  
    #pragma omp parallel sections  
    {foo();bar();baz();}  
    return 0;}  
}
```

VI-HPS Tools / 1

- Virtual institute – high productivity supercomputing
- Tool development
- Training:
 - VI-HPS/PRACE tuning workshop series
 - SC/ISC tutorials
- Many performance tools available under vi-hps.org
 - → tools → VI-HPS Tools Guide
 - Tools-Guide: flyer with a 2 page summary for each tool

Data collection

- Score-P : instrumentation based profiling / tracing
- Extrae : instrumentation based profiling / tracing

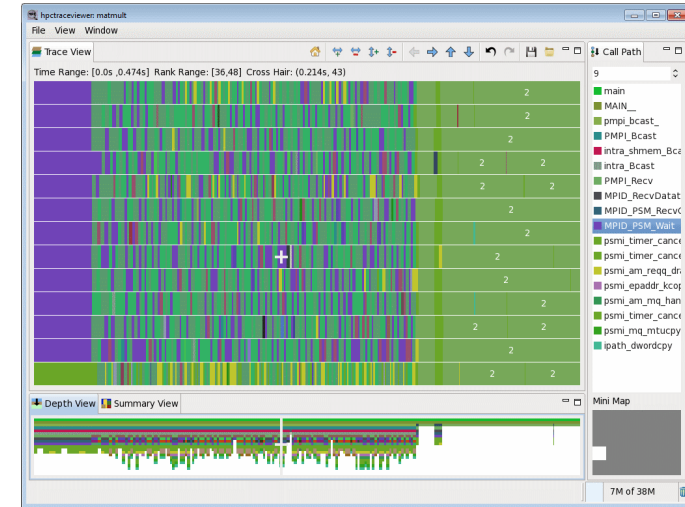
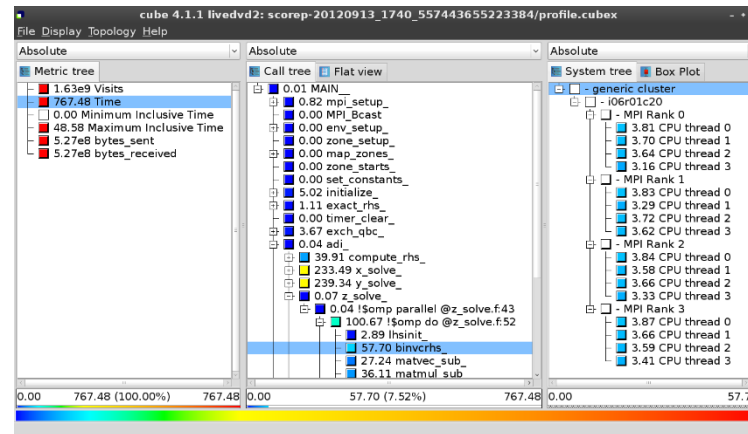
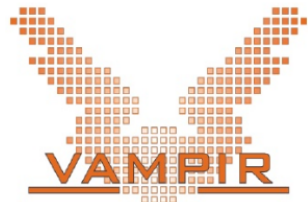
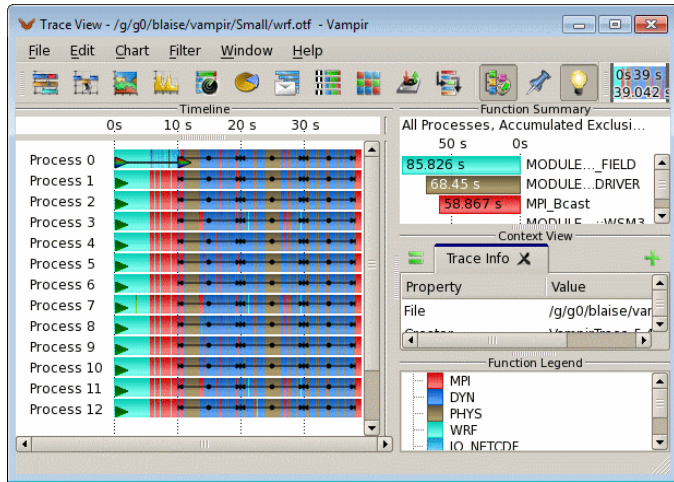
Data processing

- Scalasca : trace-based analysis

Data presentation

- ARM Map, ARM performance report
- CUBE : display for profile information
- Vampir : display for trace data (commercial/test)
- Paraver : display for extrae data
- Tau : visualization

Performance tools GUI



HPC Toolkit



Correctness:

- Data Races are very hard to find, since they do not show up every program run.
- Intel Inspector XE or ThreadSanitizer help a lot in finding these errors.
- Use really small datasets, since the runtime increases significantly.

Performance:

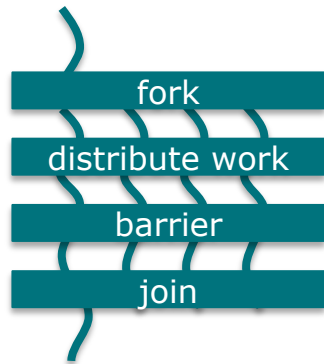
- Start with simple performance measurements like hotspots analyses and then focus on these hot spots.
- In OpenMP applications analyze the waiting time of threads. Is the waiting time balanced?
- Hardware counters might help for a better understanding of an application, but they might be hard to interpret.

OpenMP Parallel Loops

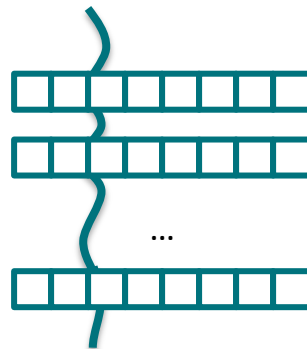
loop Construct

- Existing loop constructs are tightly bound to execution model:

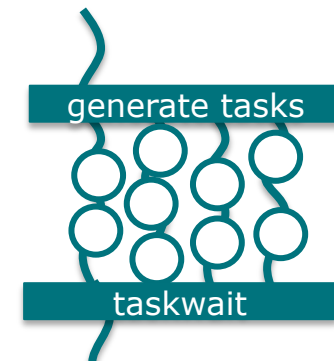
```
#pragma omp parallel for
for (i=0; i<N;++i) {...}
```



```
#pragma omp simd
for (i=0; i<N;++i) {...}
```



```
#pragma omp taskloop
for (i=0; i<N;++i) {...}
```



- The `loop` construct is meant to tell OpenMP about truly parallel semantics of a loop.

OpenMP Fully Parallel Loops

```
int main(int argc, const char* argv[]) {
    float *x = (float*) malloc(n * sizeof(float));
    float *y = (float*) malloc(n * sizeof(float));
    // Define scalars n, a, b & initialize x, y

#pragma omp parallel
#pragma omp loop
    for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
    }
}
```

Loop Constructs, Syntax

■ Syntax (C/C++)

```
#pragma omp loop [clause[[, clause],...]  
for-loops
```

■ Syntax (Fortran)

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

Loop Constructs, Clauses

- `bind(binding)`

- Binding region the loop construct should bind to

- One of: `teams`, `parallel`, `thread`

- `order(concurrent)`

- Tell the OpenMP compiler that the loop can be executed in any order.

- Default!

- `collapse(n)`

- `private(list)`

- `lastprivate(list)`

- `reduction(reduction-id: list)`

Extensions to Existing Constructs

- Existing loop constructs have been extended to also have truly parallel semantics.

- C/C++ Worksharing:**

```
#pragma omp [for|simd] order(concurrent) \  
                [clause[[,] clause],...]  
  
for-loops
```

- Fortran Worksharing:**

```
!$omp [do|simd] order(concurrent) &  
                [clause[[,] clause],...]  
  
do-loops  
[!$omp end [do|simd}]
```

DOACROSS Loops

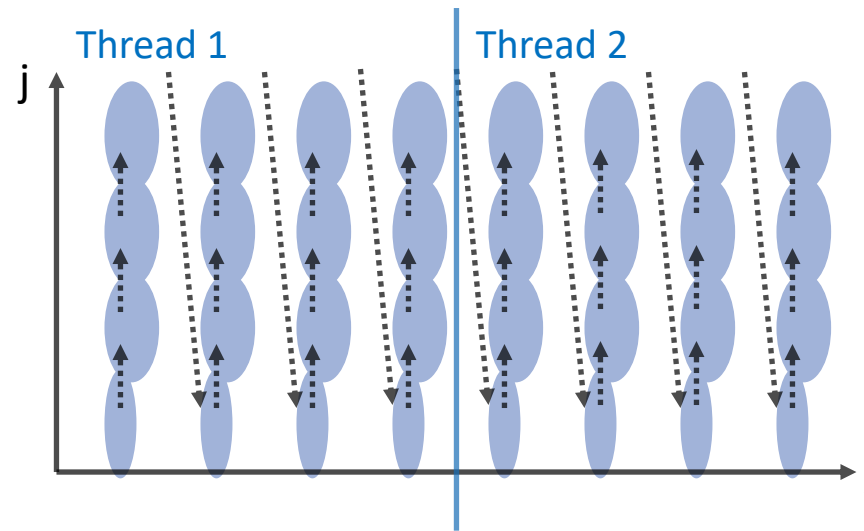
DOACROSS Loops

- “DOACROSS” loops are loops with special loop schedules
 - Restricted form of loop-carried dependencies
 - Require fine-grained synchronization protocol for parallelism
- Loop-carried dependency:
 - Loop iterations depend on each other
 - Source of dependency must be scheduled before sink of the dependency
- DOACROSS loop:
 - Data dependency is an invariant for the execution of the whole loop nest

Parallelizable Loops

- A parallel loop cannot not have any loop-carried dependencies (simplified just a little bit!)

```
for (int i = 1; i < N; ++i) {  
    for (int j = 1; j < M; ++j) {  
        b[i][j] = f(b[i][j],  
                   b[i][j], a[i][j]);  
    }  
}
```



.....> execution order
———> dependency

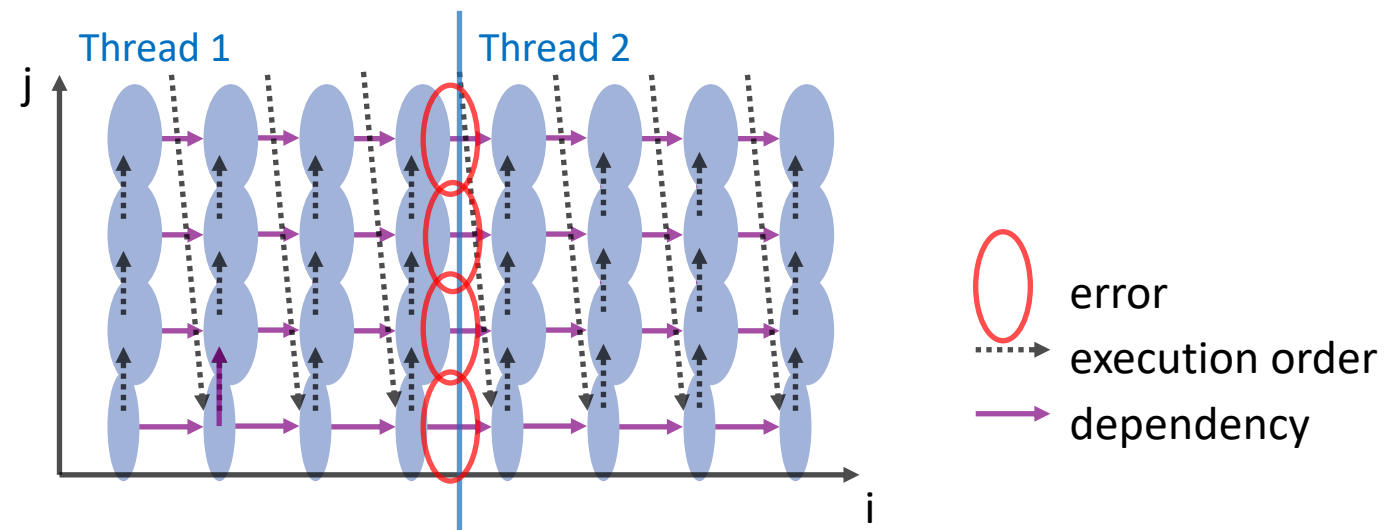
Non-parallelizable Loops

- If there is a loop-carried dependency, a loop cannot be parallelized anymore (“easily” that is)

```

for (int i = 1; i < N; ++i) {
    for (int j = 1; j < M; ++j) {
        b[i][j] = f(b[i-1][j],
                   b[i][j-1], a[i][j]);
    }
}

```



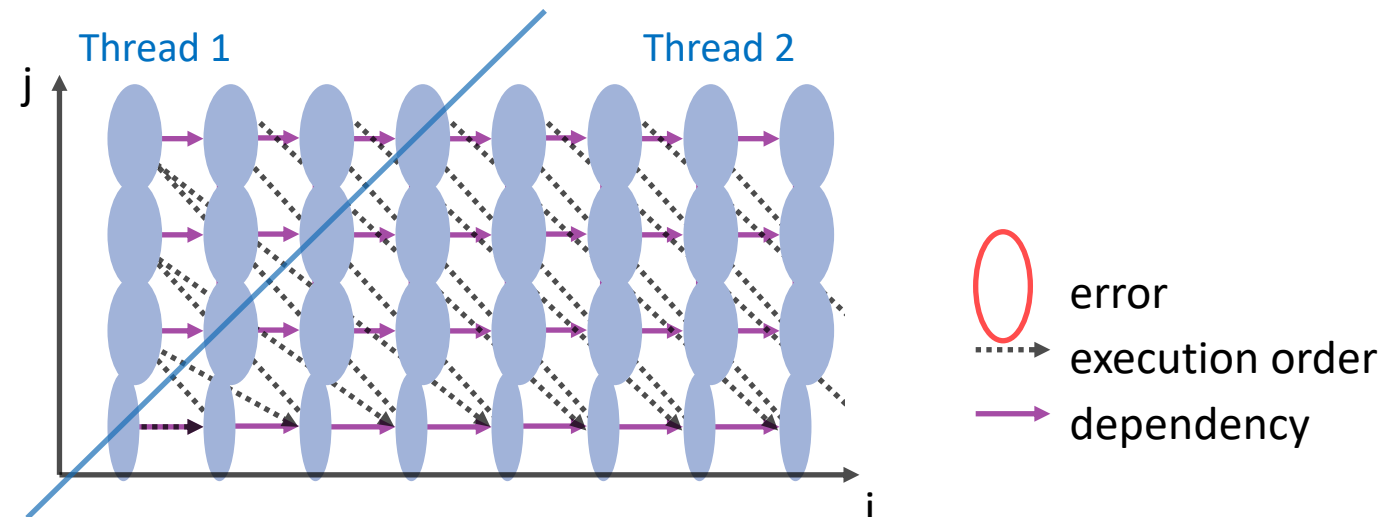
Wavefront-Parallel Loops

- If the data dependency is invariant, then skewing the loop helps remove the data dependency

```

for (int i = 1; i < N; ++i) {
  for (int j = i+1; j < i+N; ++j) {
    b[i][j-i] = f(b[i-1][j-i],
                  b[i][j-i-1], a[i][j]);
  }
}

```



DOACROSS Loops with OpenMP

- OpenMP 4.5 extends the notion of the ordered construct to describe loop-carried dependencies

- Syntax (C/C++):

```
#pragma omp for ordered(d) [clause[[, clause],...]  
for-loops
```

and

```
#pragma omp ordered [clause[[, clause],...]
```

where *clause* is one of the following:

```
depend(source)
```

```
depend(sink:vector)
```

- Syntax (Fortran):

```
!$omp do ordered(d) [clause[[, clause],...]  
do-loops
```

```
!$omp ordered [clause[[, clause],...]
```

Example

- The ordered clause tells the compiler about loop-carried dependencies and their distances

```
#pragma omp parallel for ordered(2)
for (int i = 1; i < N; ++i) {
    for (int j = 1; j < M; ++j) {
        #pragma omp ordered depend(sink:i-1,j) depend(sink:i,j-1)
            b[i][j] = f(b[i-1][j],
                       b[i][j-1], a[i][j]);
    }
    #pragma omp ordered depend(source)
}
```

Example: 3D Gauss-Seidel

```
#pragma omp for ordered(2) private(j,k)
for (i = 1; i < N-1; ++i) {
    for (j = 1; j < N-1; ++j)    {
        #pragma omp ordered depend(sink: i-1,j-1) depend(sink: i-1,j) \
            depend(sink: i-1,j+1) depend(sink: i,j-1)
        for (k = 1; k < N-1; ++k) {
            double tmp1 = (p[i-1][j-1][k-1] + p[i-1][j-1][k] + p[i-1][j-1][k+1]
                + p[i-1][j][k-1] + p[i-1][j][k] + p[i-1][j][k+1]
                + p[i-1][j+1][k-1] + p[i-1][j+1][k] + p[i-1][j+1][k+1]);
            double tmp2 = (p[i][j-1][k-1] + p[i][j-1][k] + p[i][j-1][k+1]
                + p[i][j][k-1] + p[i][j][k] + p[i][j][k+1]
                + p[i][j+1][k-1] + p[i][j+1][k] + p[i][j+1][k+1]);
            double tmp3 = (p[i+1][j-1][k-1] + p[i+1][j-1][k] + p[i+1][j-1][k+1]
                + p[i+1][j][k-1] + p[i+1][j][k] + p[i+1][j][k+1]
                + p[i+1][j+1][k-1] + p[i+1][j+1][k] + p[i+1][j+1][k+1]);
            p[i][j][k] = (tmp1 + tmp2 + tmp3) / 27.0;
        }
    }
    #pragma omp ordered depend(source)
}
}
```

Cancellation

OpenMP 3.1 Parallel Abort

- Once started, parallel execution cannot be aborted in OpenMP 3.1
 - Code regions must always run to completion
 - (or not start at all)
- Cancellation in OpenMP 4.0 provides a best-effort approach to terminate OpenMP regions
 - Best-effort: not guaranteed to trigger termination immediately
 - Triggered “as soon as” possible

Cancellation Constructs

- Two constructs:

- Activate cancellation:

```
C/C++:    #pragma omp cancel  
Fortran:  !$omp cancel
```

- Check for cancellation:

```
C/C++:    #pragma omp cancellation point  
Fortran:  !$omp cancellation point
```

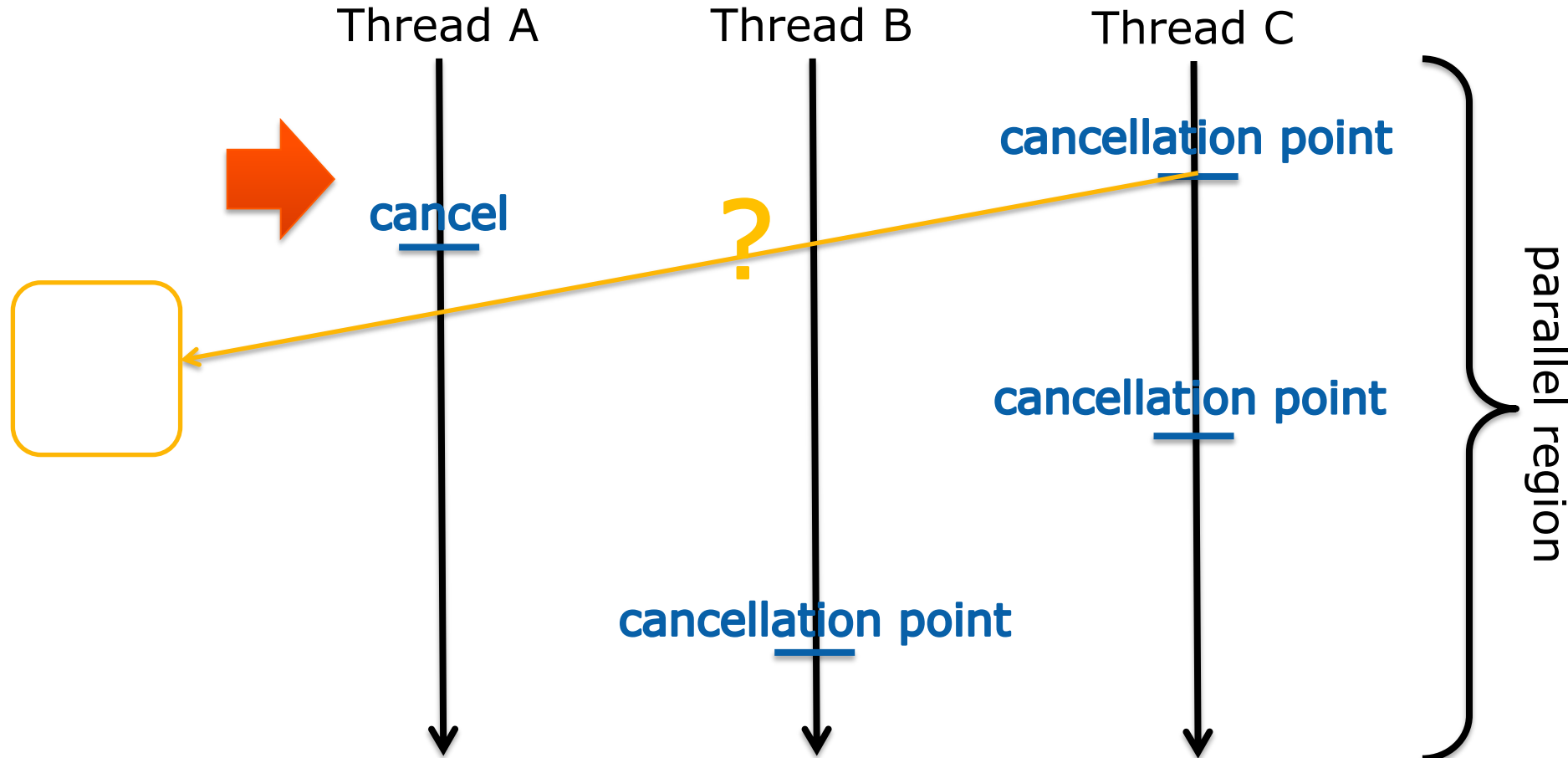
- Check for cancellation only a certain points

- Avoid unnecessary overheads

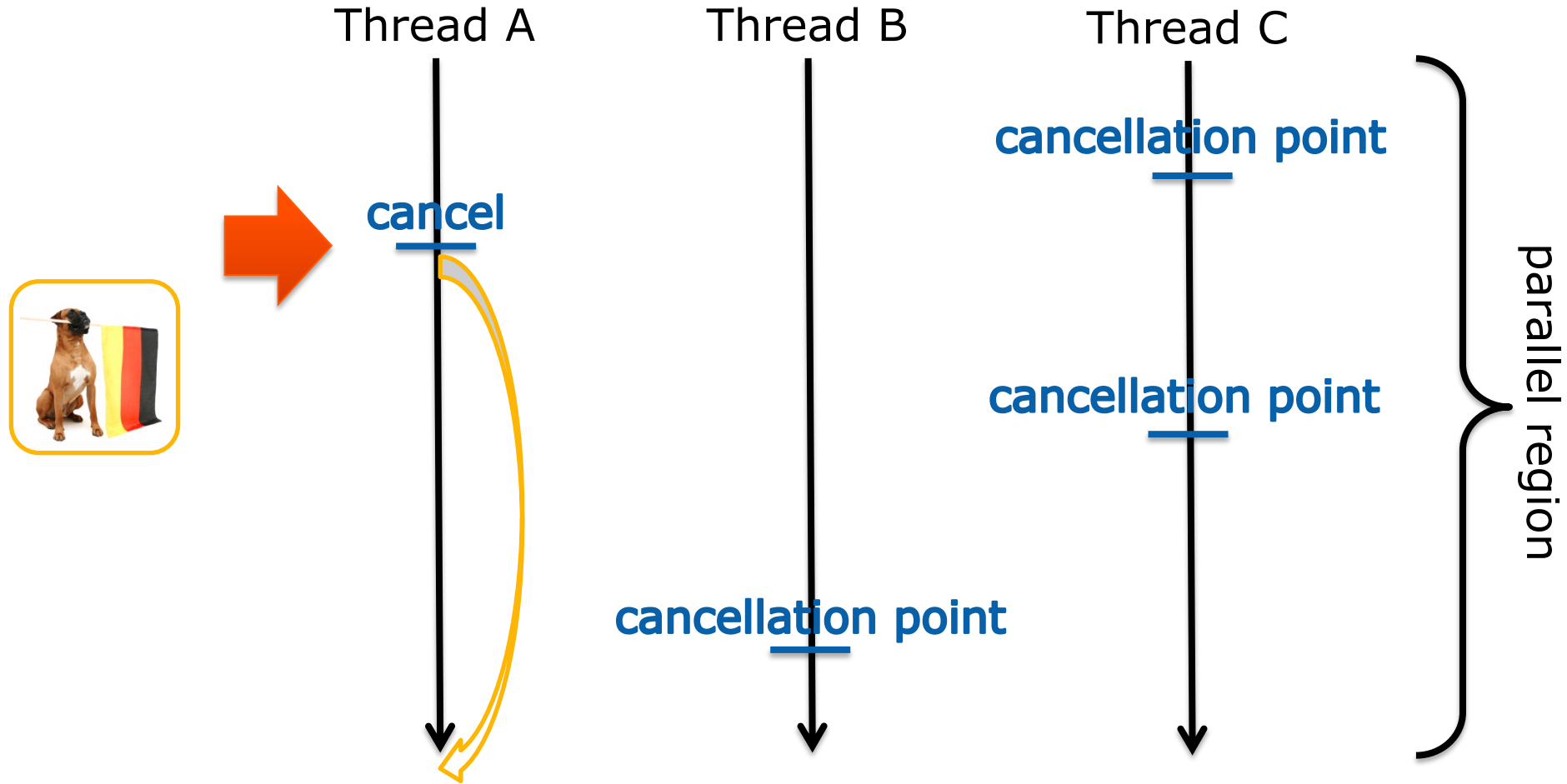
- Programmers need to reason about cancellation

- Cleanup code needs to be added manually

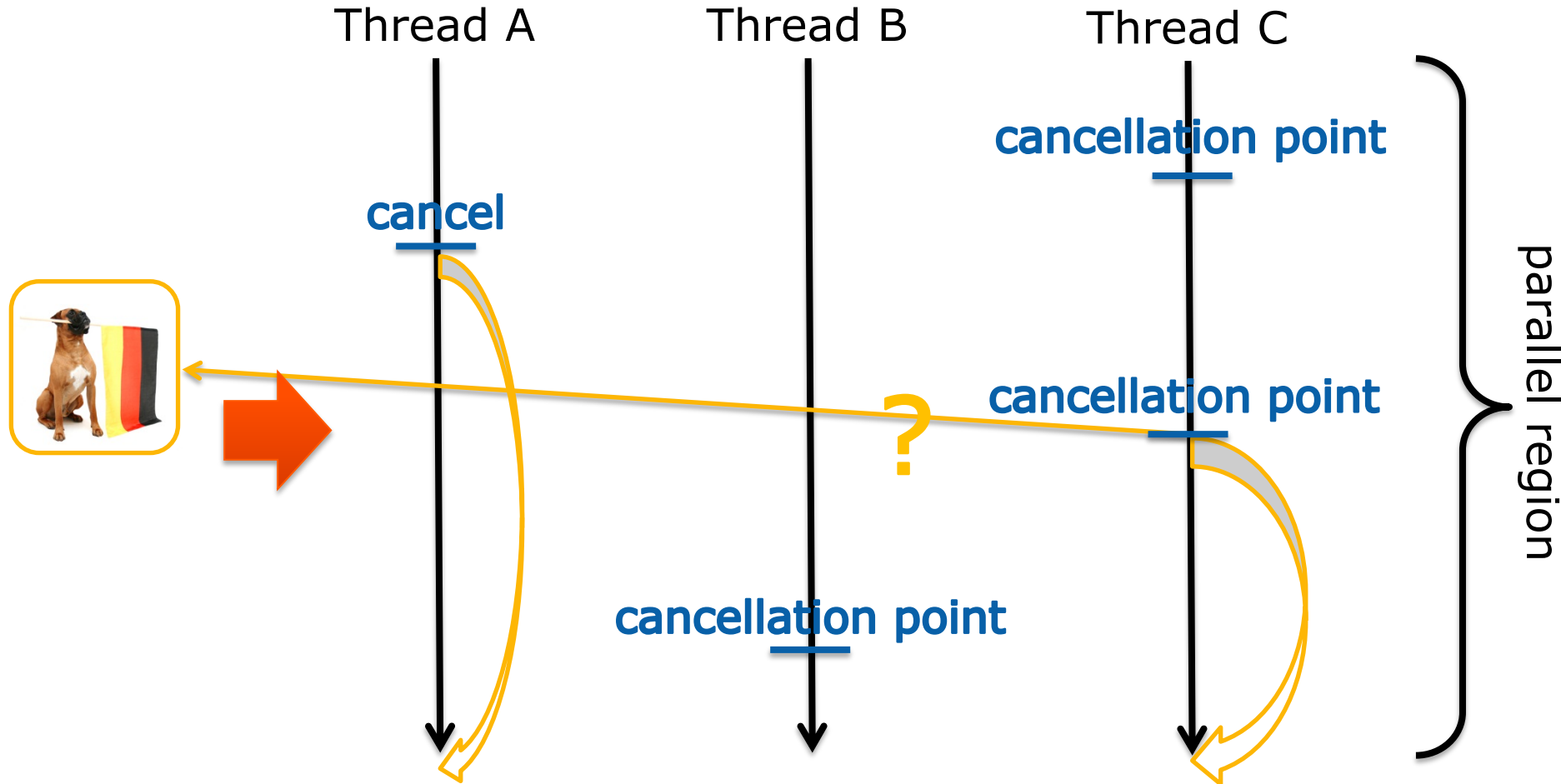
Cancellation Semantics



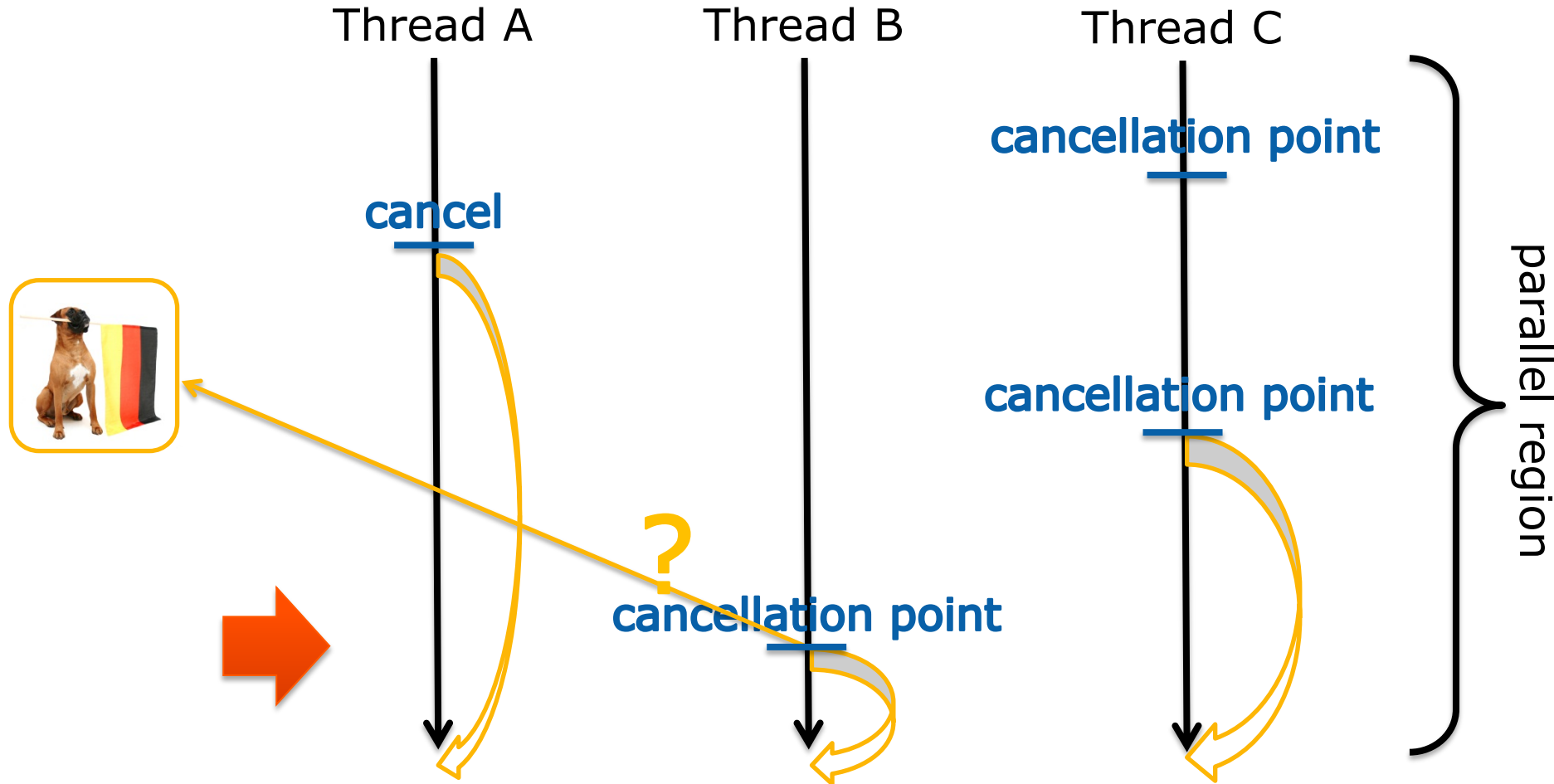
Cancellation Semantics



Cancellation Semantics



Cancellation Semantics



cancel Construct

■ Syntax:

```
#pragma omp cancel construct-type-clause [ [, ]if-clause ]  
!$omp cancel construct-type-clause [ [, ]if-clause ]
```

■ Clauses:

```
parallel  
sections  
for (C/C++)  
do (Fortran)  
taskgroup  
if (scalar-expression)
```

■ Semantics

- Requests cancellation of the inner-most OpenMP region of the type specified in *construct-type-clause*
- Lets the encountering thread/task proceed to the end of the canceled region

cancellation point Construct

■ Syntax:

```
#pragma omp cancellation point construct-type-clause  
!$omp cancellation point construct-type-clause
```

■ Clauses:

```
parallel  
sections  
for (C/C++)  
do (Fortran)  
taskgroup
```

■ Semantics

- Introduces a user-defined cancellation point
- Pre-defined cancellation points:
 - implicit/explicit barriers regions
 - cancel regions

Cancellation of OpenMP Tasks

- Cancellation only acts on tasks grouped by the `taskgroup` construct
 - The encountering tasks jumps to the end of its task region
 - Any executing task will run to completion (or until they reach a cancellation point region)
 - Any task that has not yet begun execution may be discarded (and is considered completed)
- Tasks cancellation also occurs, if a parallel region is canceled.
 - But not if cancellation affects a worksharing construct.

Task Cancellation Example

```
binary_tree_t* search_tree_parallel(binary_tree_t* tree, int value) {
    binary_tree_t* found = NULL;
    #pragma omp parallel shared(found,tree,value)
    {
        #pragma omp master
        {
            #pragma omp taskgroup
            {
                found = search_tree(tree, value);
            }
        }
    }
    return found;
}
```


Task Cancellation Example

```
binary_tree_t* search_tree(
    binary_tree_t* tree, int value,
    int level) {
    binary_tree_t* found = NULL;
    if (tree) {
        if (tree->value == value) {
            found = tree;
        }
        else {
            #pragma omp task shared(found)
            {
                binary_tree_t* found_left;
                found_left =
                    search_tree(tree->left, value);
                if (found_left) {
                    #pragma omp atomic write
                    found = found_left;
                    #pragma omp cancel taskgroup
                }
            }
        }
    }
}
```

```
#pragma omp task shared(found)
{
    binary_tree_t* found_right;
    found_right =
        search_tree(tree->right, value);
    if (found_right) {
        #pragma omp atomic write
        found = found_right;
        #pragma omp cancel taskgroup
    }
}
#pragma omp taskwait
}
return found;
}
```

Advanced Task Synchronization

Asynchronous API Interaction

- Some APIs are based on asynchronous operations
 - MPI asynchronous send and receive
 - Asynchronous I/O
 - CUDA and OpenCL stream-based offloading
 - In general: any other API/model that executes asynchronously with OpenMP (tasks)
- Example: CUDA memory transfers


```
do_something();  
cudaMemcpyAsync(dst, src, nbytes, cudaMemcpyDeviceToHost, stream);  
do_something_else();  
cudaStreamSynchronize(stream);  
do_other_important_stuff(dst);
```

- Programmers need a mechanism to marry asynchronous APIs with the parallel task model of OpenMP
 - How to synchronize completions events with task execution?



Try 1: Use just OpenMP Tasks

```
void cuda_example() {  
#pragma omp task // task A  
  {  
    do_something();  
    cudaMemcpyAsync(dst, src, nbytes, cudaMemcpyDeviceToHost, stream);  
  }  
#pragma omp task // task B  
  {  
    do_something_else();  
  }  
#pragma omp task // task C  
  {  
    cudaStreamSynchronize(stream);  
    do_other_important_stuff(dst);  
  }  
}
```



Race condition between the tasks A & C,
task C may start execution before
task A enqueues memory transfer.

- This solution does not work!

Try 2: Use just OpenMP Tasks Dependences

```
void cuda_example() {  
#pragma omp task depend(out:stream) // task A  
  {  
    do_something();  
    cudaMemcpyAsync(dst, src, nbytes, cudaMemcpyDeviceToHost, stream);  
  }  
#pragma omp task // task B  
  {  
    do_something_else();  
  }  
#pragma omp task depend(in:stream) // task C  
  {  
    cudaStreamSynchronize(stream);  
    do_other_important_stuff(dst);  
  }  
}
```

Synchronize execution of tasks through dependence. May work, but task C will be blocked waiting for the data transfer to finish

- This solution may work, but
 - takes a thread away from execution while the system is handling the data transfer.
 - may be problematic if called interface is not thread-safe

OpenMP Detachable Tasks

- OpenMP 5.0 introduces the concept of a detachable task
 - Task can detach from executing thread without being “completed”
 - Regular task synchronization mechanisms can be applied to await completion of a detached task
 - Runtime API to complete a task
- Detached task events: `omp_event_t` datatype
- Detached task clause
`detach(event)`
- Runtime API
`void omp_fulfill_event(omp_event_t *event)`

Detaching Tasks

```
omp_event_t *event;  
void detach_example() {  
#pragma omp task detach(event)  
  {  
    important_code();  
  } ①  
#pragma omp taskwait ② ④  
}
```

Some other thread/task:

```
omp_fulfill_event(event); ③
```

1. Task detaches
2. taskwait construct cannot complete
3. Signal event for completion
4. Task completes and taskwait can continue

Putting It All Together

```
void CUDART_CB callback(cudaStream_t stream, cudaError_t status, void *cb_dat) {
    ③ omp_fulfill_event((omp_event_t *) cb_data);
}

void cuda_example() {
    omp_event_t *cuda_event;
#pragma omp task detach(cuda_event) // task A
    {
        do_something();
        cudaMemcpyAsync(dst, src, nbytes, cudaMemcpyDeviceToHost, stream);
        cudaStreamAddCallback(stream, callback, cuda_event, 0);
    ① }
#pragma omp task // task B
    do_something_else();

#pragma omp taskwait ② ④
#pragma omp task // task C
    {
        do_other_important_stuff(dst);
    } }
```



1. Task A detaches
2. taskwait does not continue
3. When memory transfer completes, callback is invoked to signal the event for task completion
4. taskwait continues, task C executes

Removing the taskwait Construct

```
void CUDART_CB callback(cudaStream_t stream, cudaError_t status, void *cb_dat) {  
    ② omp_fulfill_event((omp_event_t *) cb_data);  
}  
  
void cuda_example() {  
    omp_event_t *cuda_event;  
#pragma omp task depend(out:dst) detach(cuda_event) // task A  
    {  
        do_something();  
        cudaMemcpyAsync(dst, src, nbytes, cudaMemcpyDeviceToHost, stream);  
        ① cudaStreamAddCallback(stream, callback, cuda_event, 0);  
    }  
#pragma omp task // task B  
    do_something_else();  
  
#pragma omp task depend(in:dst) // task C  
    {  
        ③ do_other_important_stuff(dst);  
    }  
}
```

1. Task A detaches and task C will not execute because of its unfulfilled dependency on A
2. When memory transfer completes, callback is invoked to signal the event for task completion
3. Task A completes and C's dependency is fulfilled

OpenMP API Version 5.0

State of the Union

Architecture Review Board

The mission of the OpenMP ARB (Architecture Review Board) is to standardize directive-based multi-language high-level parallelism that is performant, productive and portable.



Development Process of the Specification

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



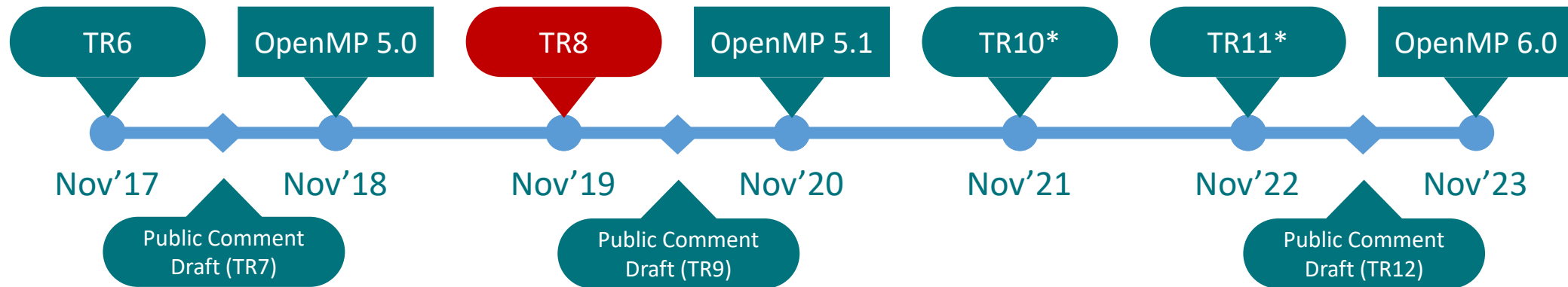
- Release process for specifications:



OpenMP Roadmap

■ OpenMP has a well-defined roadmap:

- 5-year cadence for major releases
- One minor release in between
- (At least) one Technical Report (TR) with feature previews in every year



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

Highlights of TR8 and Version 5.1 Plans

- Some significant extensions to existing functionality
 - The `interop` construct improves native device support (e.g., CUDA streams)
 - Major improvements to `declare variant` construct
 - Support for mapping (translated) function pointers
 - The `assume` directive supports optimization hints (and well-defined OpenMP subsets)
 - The `error` directive supports user-defined warnings and errors
 - Added the `tile` directive, the first of many possible loop transformation directives
 - Expect to add one more transformation in OpenMP 5.1 (probably `unroll` but still TBD)
 - Initial extensions to specify OpenMP directives as C++ attributes (more to come in 5.1)
 - Full support for C11, C18, C++11, C++14, C++17, close for Fortran 2008
- OpenMP 5.1 feature freeze will occur in May 2020
 - May add `taskloop` affinity and dependences (`inoutset` dependences already added)

OpenMP API Version 6.0 Outlook – Plans

- Better support for descriptive and prescriptive control
- More support for memory affinity and complex memory hierarchies
- Support for pipelining, other computation/data associations
- Continued improvements to device support
 - Extensions of deep copy support (serialize/deserialize functions)
- Task-only, unshackled or free-agent threads
- Event-driven parallelism
- 72 in progress 5.1 issues; 19 issues already deferred to 6.0

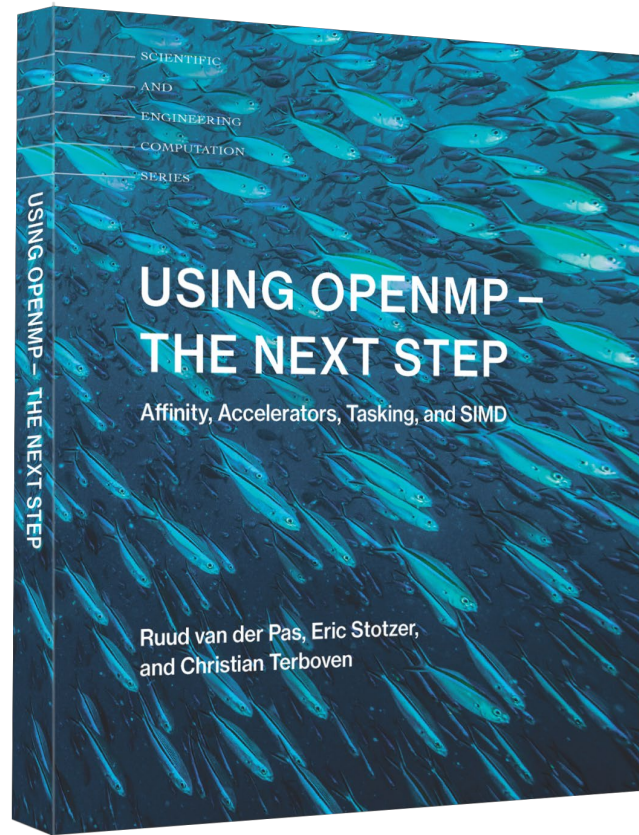
Printed OpenMP API Specification



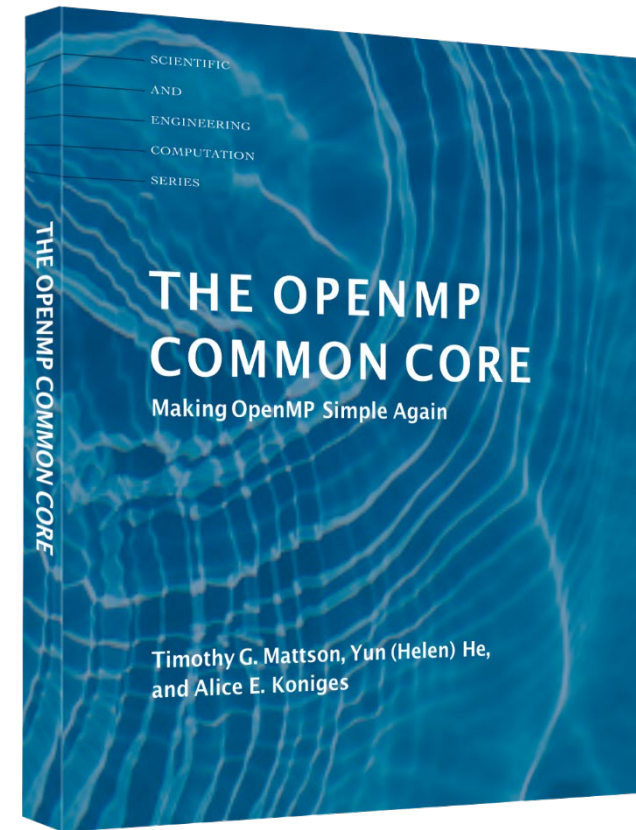
- Save your printer-ink and get the full specification as a paperback book!
 - Always have the spec in easy reach.
 - Includes the entire specification with the same pagination and line numbers as the PDF.
 - Available at a near-wholesale price.

- Get yours at Amazon at <http://bit.ly/spec-50>

Recent Books about OpenMP



Covers all of the OpenMP 4.5 features, 2017



Introduces the OpenMP Common Core, 2019

Help Us Shape the Future of OpenMP

- OpenMP continues to grow
 - 33 members currently
- You can contribute to our annual releases
- Attend IWOMP, become a cOMPunity member
- OpenMP membership types now include less expensive memberships
 - Please get in touch with me if you are interested



Visit www.openmp.org for more information