Progamming the OpenMP API

Taskloop & Dependences

Tasking Use Cases

Tasking Use Case: Fibonacci (Recursion)

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

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

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

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

  #pragma omp taskwait

  return(fn1 + fn2);
}
```

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

Tasking Use Case: Cholesky (Synchronization)

```
void cholesky(int ts, int nt, double* a[nt][nt]) {
  for (int k = 0; k < nt; k++) {
    potrf(a[k][k], ts, ts);
    // Triangular systems
    for (int i = k + 1; i < nt; i++) {
      #pragma omp task
      trsm(a[k][k], a[k][i], ts, ts);
    }
    #pragma omp taskwait
    // Update trailing matrix
    for (int i = k + 1; i < nt; i++) {
      for (int j = k + 1; j < i; j++) {
        #pragma omp task
        dgemm(a[k][i], a[k][j], a[j][i], ts, ts);
      }
      #pragma omp task
      syrk(a[k][i], a[i][i], ts, ts);
    }
    #pragma omp taskwait
  }
}
```

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

Tasking Use Case: saxpy (Blocking/Tiling)

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

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

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

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

The `taskloop` Construct

Tasking Use Case: saxpy (taskloop)

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

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

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

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

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

  - → Hiding the internal details
  - → Grain size ~ Tile size (TS) → but implementation decides exact grain size

The taskloop Construct

- Task generating construct: decompose a loop into chunks, create a task for each loop chunk

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

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

- Where clause is one of:

Data Environment

- → shared(list)
- → private(list)
- → firstprivate(list)
- → lastprivate(list)
- → default(sh | *pr* | *fp* | none)
- → reduction(r-id: list)
- → in_reduction(r-id: list)

Chunks/Grain

- → grainsize(grain-size)
- → num_tasks(num-tasks)

Cutoff Strategies

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

Scheduler (R/H)

- → untied
- → priority(priority-value)

Miscellaneous

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

Taskloop decomposition approaches

- Clause: grainsize(grain-size)
  - → Chunks have at least grain-size iterations
  - → Chunks have maximum 2x grain-size iterations

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

- Clause: num_tasks(num-tasks)
  - → Create num-tasks chunks
  - → Each chunk must have at least one iteration

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

- If none of previous clauses is present, the *number of chunks* and the *number of iterations per chunk* is implementation defined
- Additional considerations:
  - → The order of the creation of the loop tasks is unspecified
  - → Taskloop creates an implicit taskgroup region; **nogroup** → no implicit taskgroup region is created

Collapsing iteration spaces with taskloop

- The collapse clause in the taskloop construct

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

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

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

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

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

Task reductions (using taskloop)

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

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

  return r;
}
```

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

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

Composite construct: taskloop simd

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

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

  - → Fortran syntax:

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

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

Worksharing vs. taskloop constructs (1/2)

```
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

Improving Tasking Performance: Task Dependences

Motivation

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

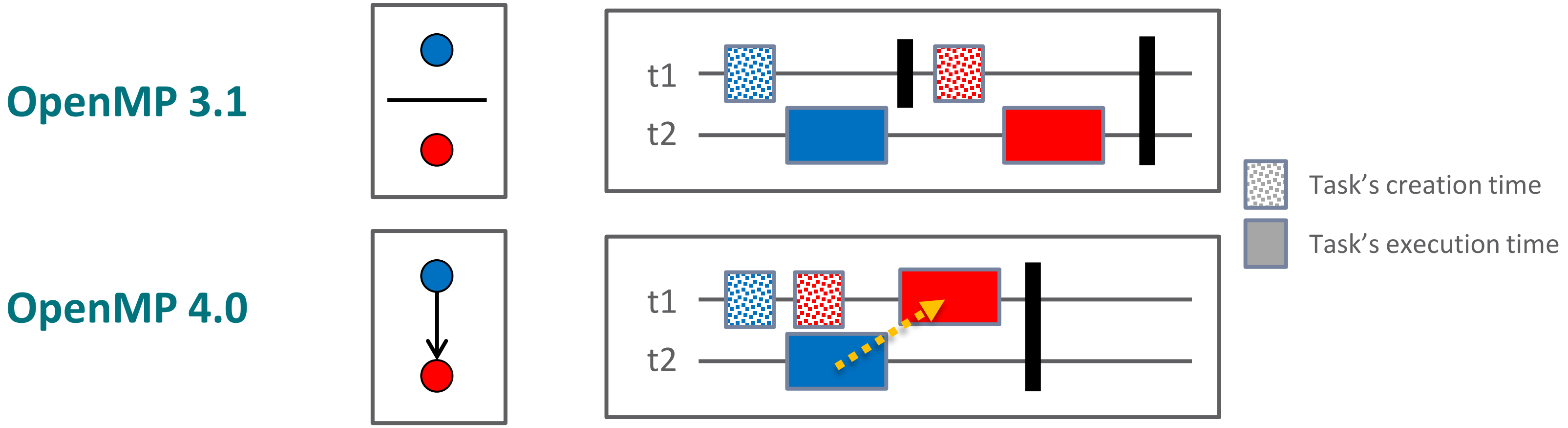
OpenMP 3.1

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

   #pragma omp taskwait

🔴 #pragma omp task
   x++;
}
```

OpenMP 4.0

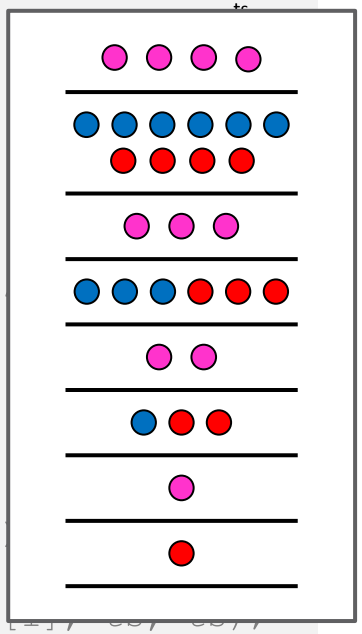
```
int x = 0;
#pragma omp parallel
#pragma omp single
{
🔵 #pragma omp task depend(in: x)
                                 d::endl;



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

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



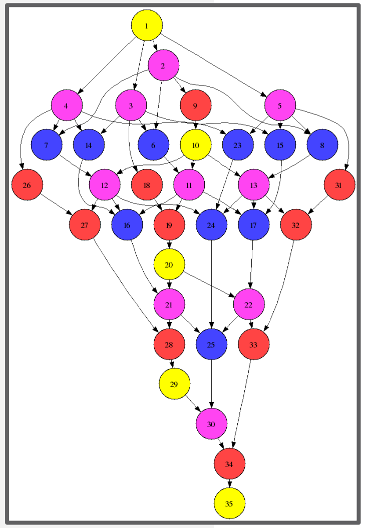
Motivation: Cholesky factorization

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

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

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



OpenMP 3.1

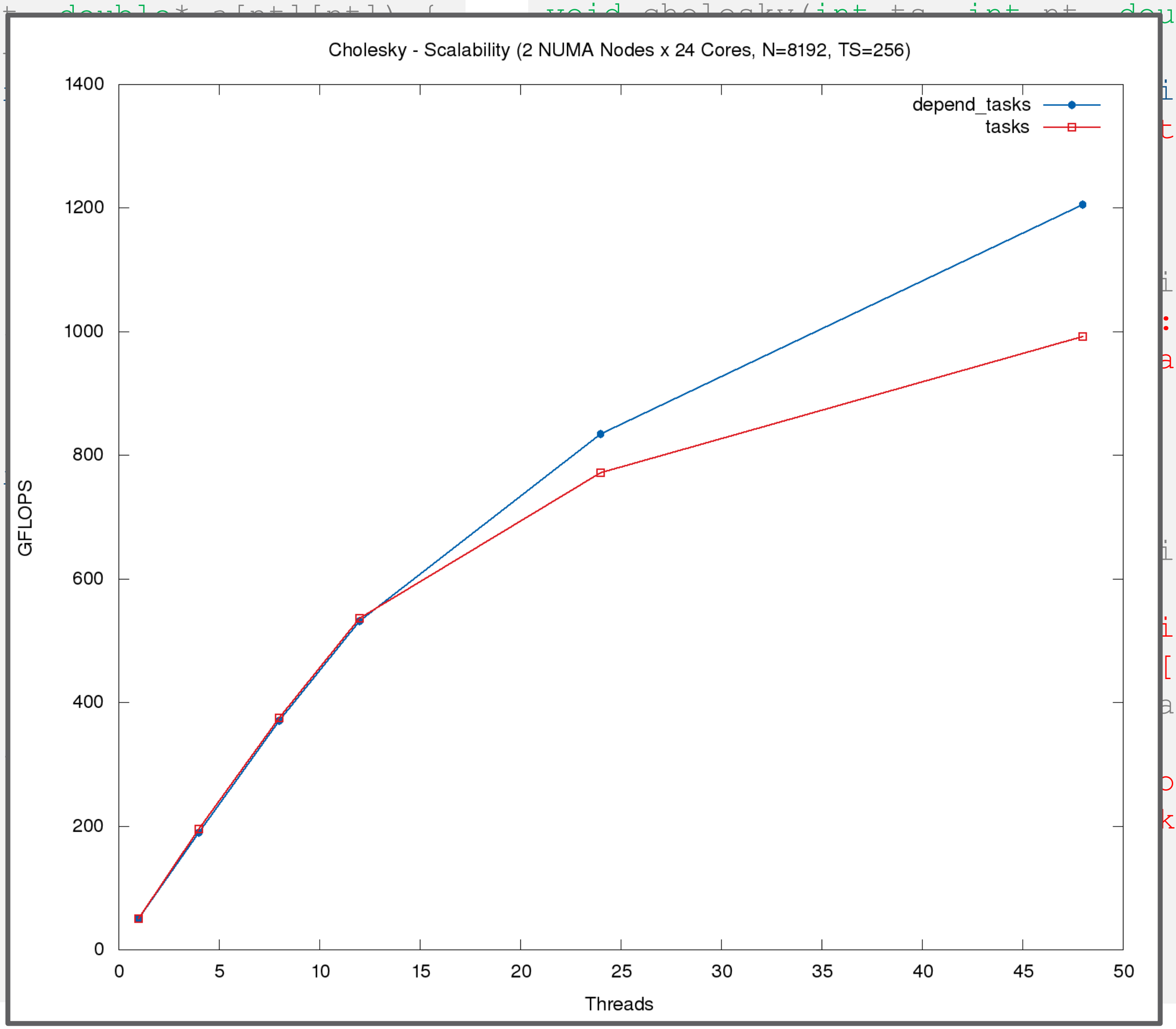
```
void cholesky(int ts, int nt, double* a[nt][nt]) {
  for (int k = 0; k < nt; k++) {
    // Diagonal Block factorization
    #pragma omp task depend(inout: a[k][k])
  ● potrf(a[k][k], ts, ts);

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

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



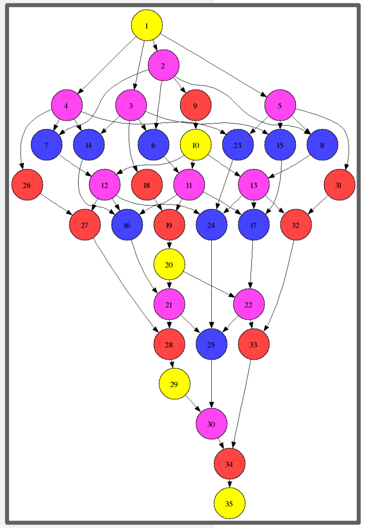
OpenMP 4.0

Motivation: Cholesky factorization

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

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

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



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

                                      ion
                                      t: a[k][k])



                                      i++) {
                                      : a[k][k])
                                      a[k][i])
                                       ts);



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

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



OpenMP 4.0

Using 2017 Intel compiler

What's in the spec

What’s in the spec: a bit of history

OpenMP 4.0

- The `depend` clause was added to the `task` construct

OpenMP 4.5

- The `depend` clause was added to the target constructs
- Support to doacross loops

OpenMP 5.0

- `lvalue` expressions in the depend clause
- New dependency type: `mutexinoutset`
- Iterators were added to the `depend` clause
- The `depend` clause was added to the `taskwait`
- Dependable objects

OpenMP 5.1

- New dependency type: `inoutset`

What’s in the spec: syntax depend clause

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

where:

→ `depend-modifier` is used to define iterators

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

→ A `list-item` may be:

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

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

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

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

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

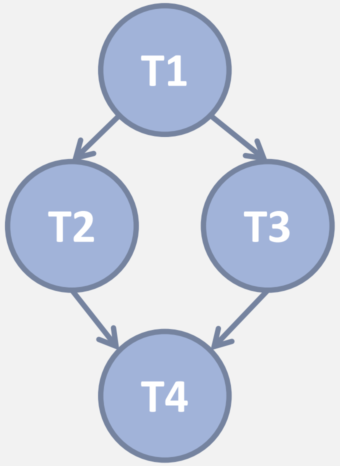
What’s in the spec: sema depend clause (1)

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

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

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

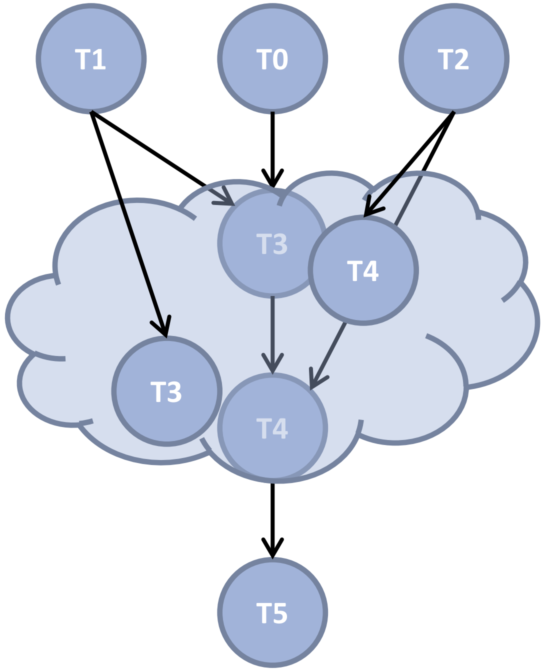
```
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(inout: x) //T1
  { ... }

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

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

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



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

- Set types: `inoutset & mutexinoutset`

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

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

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

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

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

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



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

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

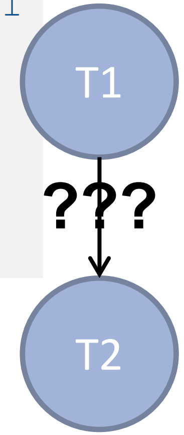
What’s in the spec: sema depend clause (3)

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

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

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

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

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

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

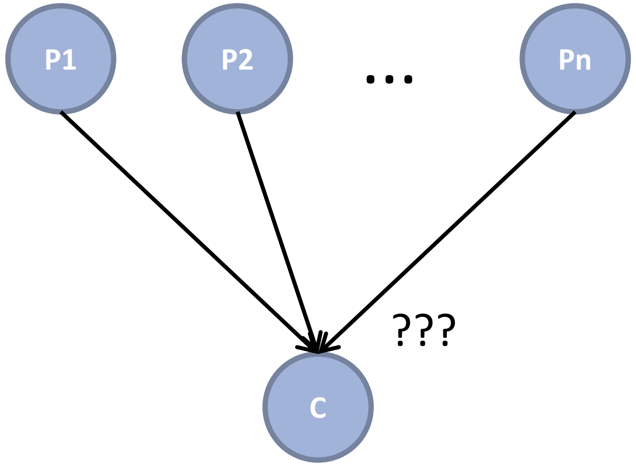


What’s in the spec: sema depend clause (4)

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

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

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

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

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



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

Philosophy

Philosophy: data-flow model

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

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

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

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
i
#
#
{

}
```

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

}
```

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

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

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

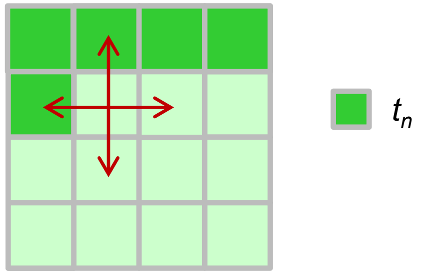
Use case

Use case: intro to Gauss-seidel

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

Access pattern analysis

For a specific t, i and j



Each cell depends on:

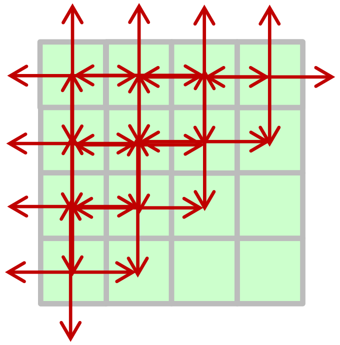
- two cells (north & west) that are computed in the current time step, and
- two cells (south & east) that were computed in the previous time step

Use case: Gauss-seidel (2)

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

1st parallelization strategy

For an specific t



🟩 t_n

We can exploit the wavefront to obtain parallelism!!

Use case : Gauss-seidel (3)

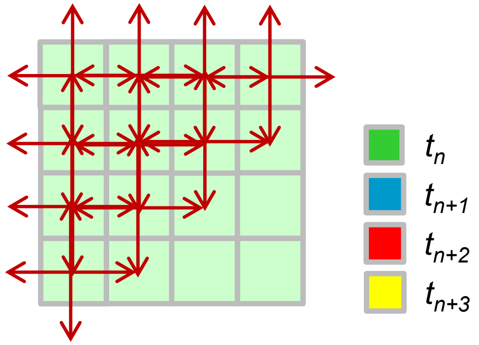
```
void gauss_seidel(int tsteps, int size, int TS, int (*p)[size]) {
  int NB = size / TS;
  #pragma omp parallel
  for (int t = 0; t < tsteps; ++t) {
    // First NB diagonals
    for (int diag = 0; diag < NB; ++diag) {
      #pragma omp for
      for (int d = 0; d <= diag; ++d) {
        int ii =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

Use case : Gauss-seidel (4)

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

2nd parallelization strategy

multiple time iterations

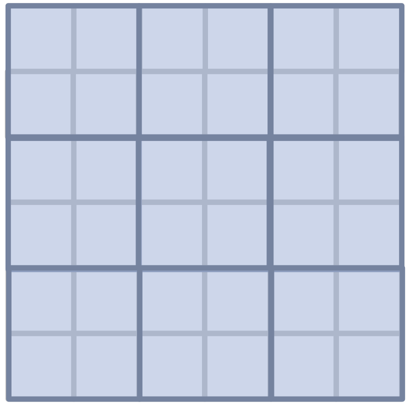


We can exploit the wavefront of multiple time steps to obtain MORE parallelism!!

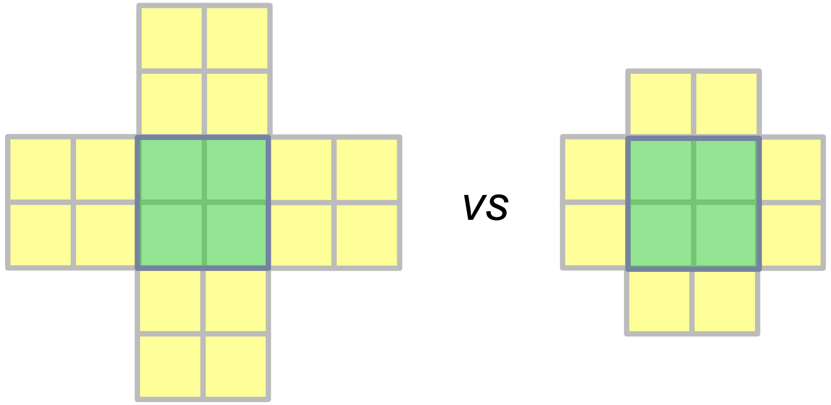
Use case : Gauss-seidel (5)

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

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

        {
          for (int i=ii; i<(1+ii)*TS; ++i)
            for (int j=jj; j<(1+jj)*TS; ++j)
               p[i][j] = 0.25 * (p[i][j-1] + p[i][j+1] +
                                 p[i-1][j] + p[i+1][j]);
        }
      }
}
```

inner matrix region

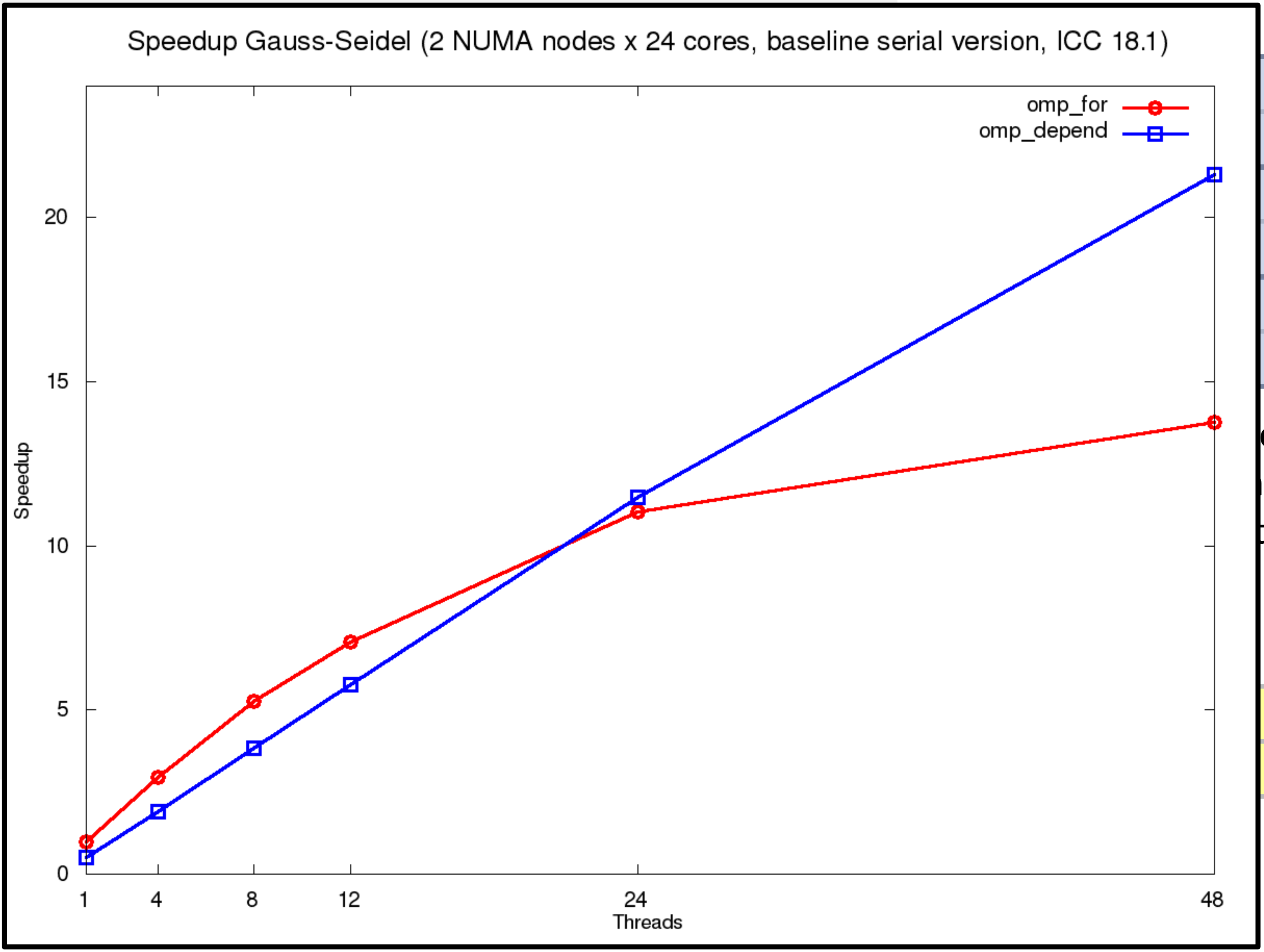


Q: Why do the input dependences depend on the whole block rather than just a column/row?

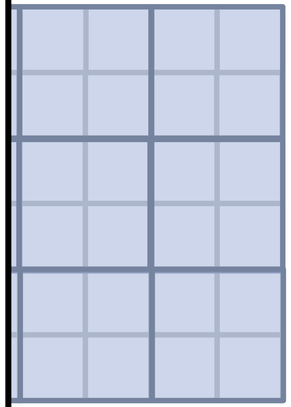


Use case : Gauss-seidel (5)

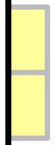
```
void gauss_seidel(i
  int NB = size / T

  #pragma omp paral
  #pragma omp singl
  for (int t = 0; t
    for (int ii=1;
      for (int jj=1
        #pragma omp
            depend(

        {
          for (int
            for (in
               p[i]

        }
      }
}
```



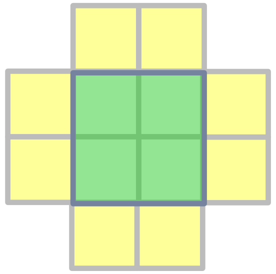
matrix region



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



vs



Advanced features: deps on `taskwait`

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

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

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

  #pragma omp taskwait depend(in: x)

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

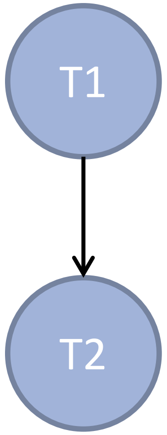
Advanced features: dependable objects (1)

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

Advanced features: dependable objects (2)

- Offer a way to manually handle dependences

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

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



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

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

  #pragma omp depobj(obj) update(in)

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

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

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:

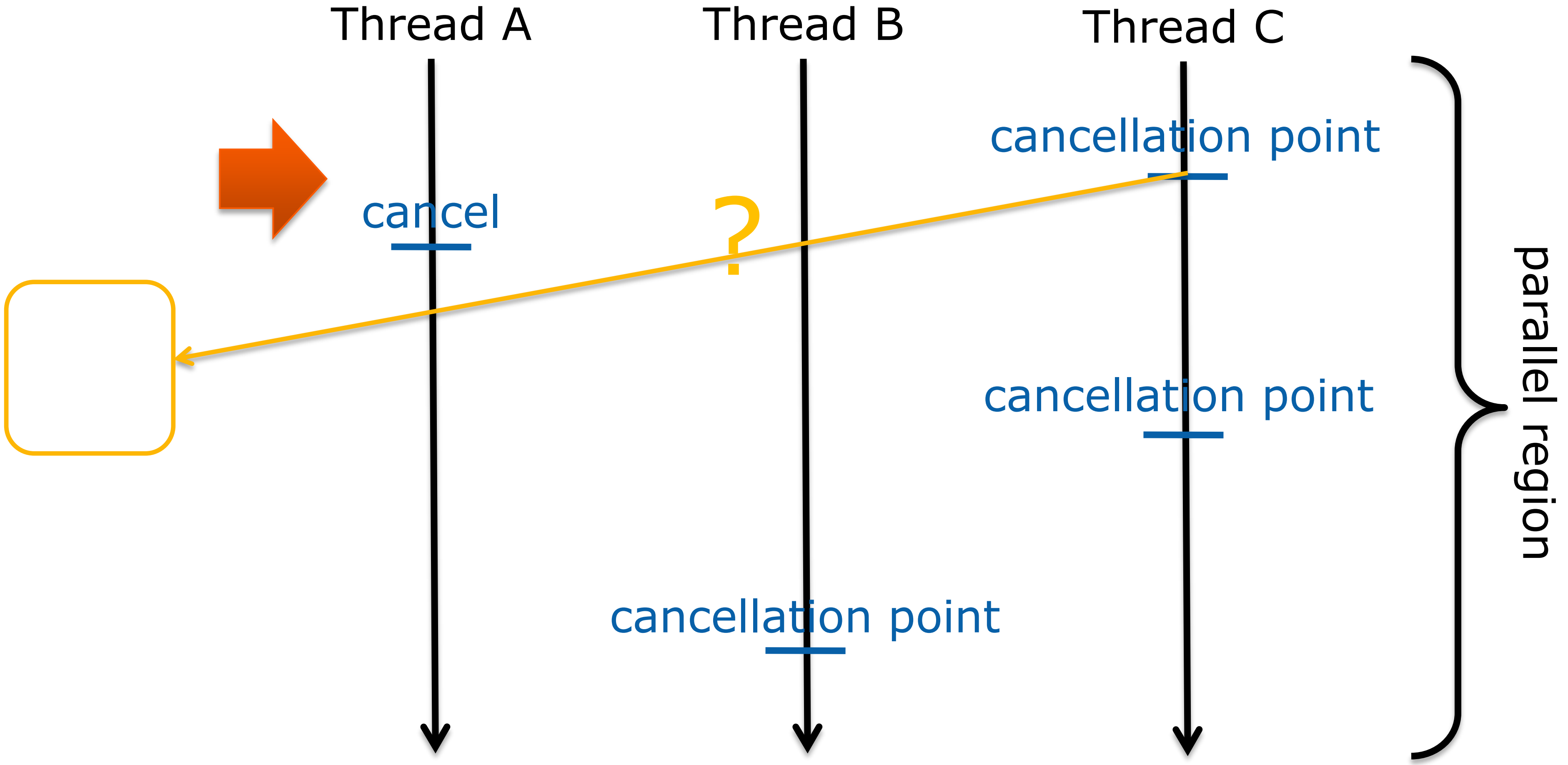
    C/C++: `#pragma omp cancel`
    Fortran: `!$omp cancel`

  - → Check for cancellation:

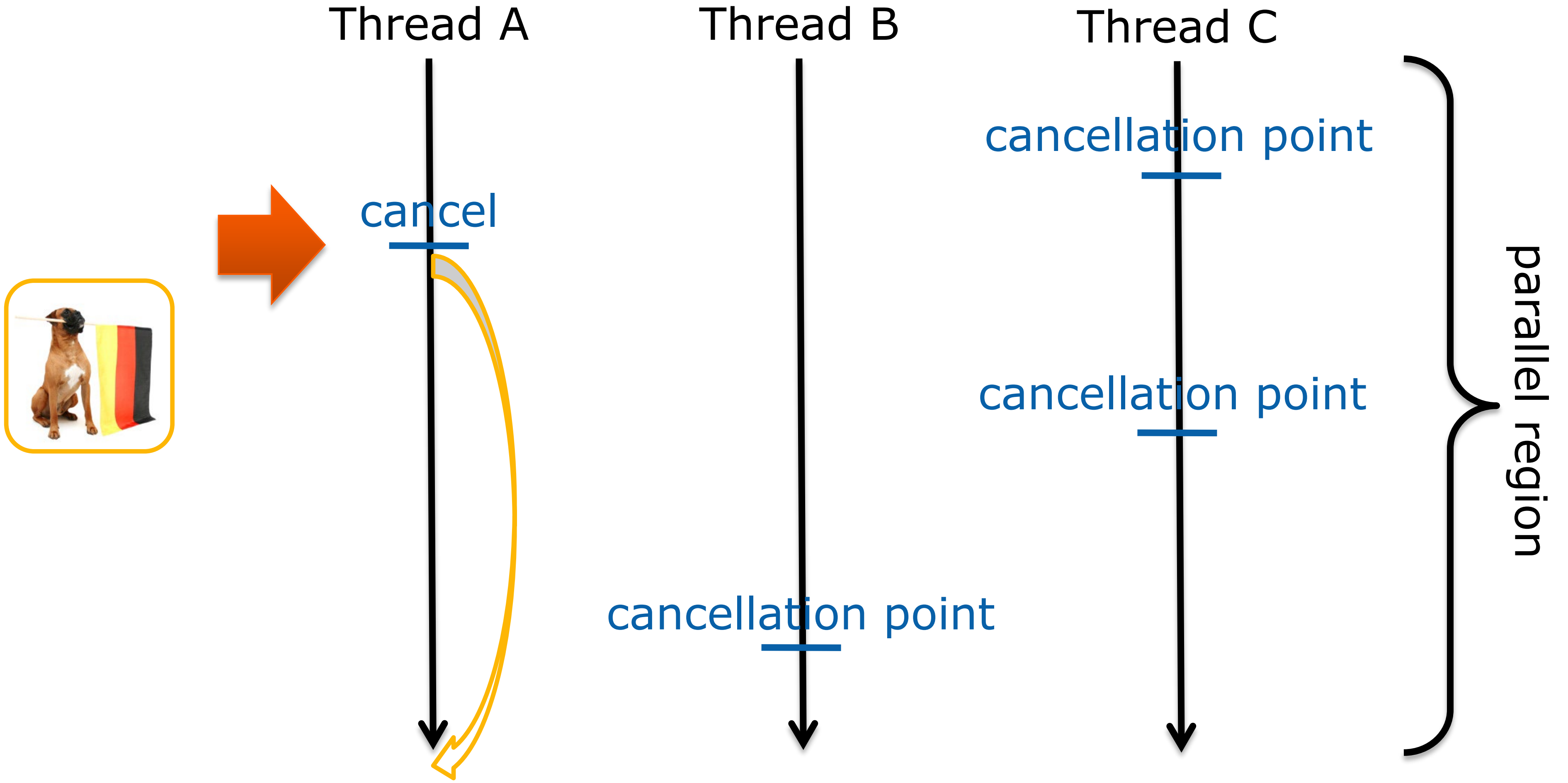
    C/C++: `#pragma omp cancellation point`
    Fortran: `!$omp cancellation point`

- Check for cancellation only a certain points
  - → Avoid unnecessary overheads
  - → Programmers need to reason about cancellation
  - → Cleanup code needs to be added manually

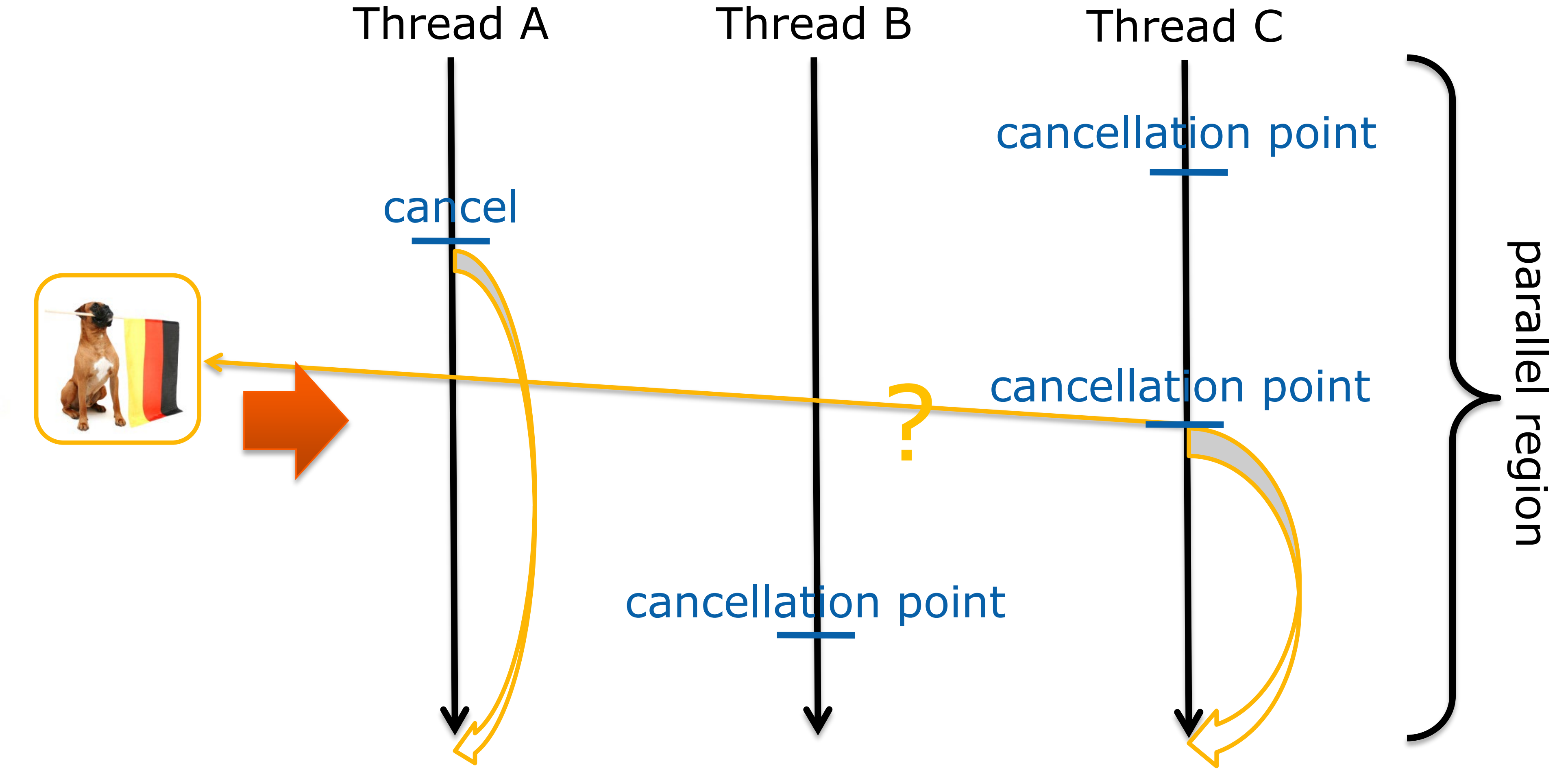
Cancellation Semantics



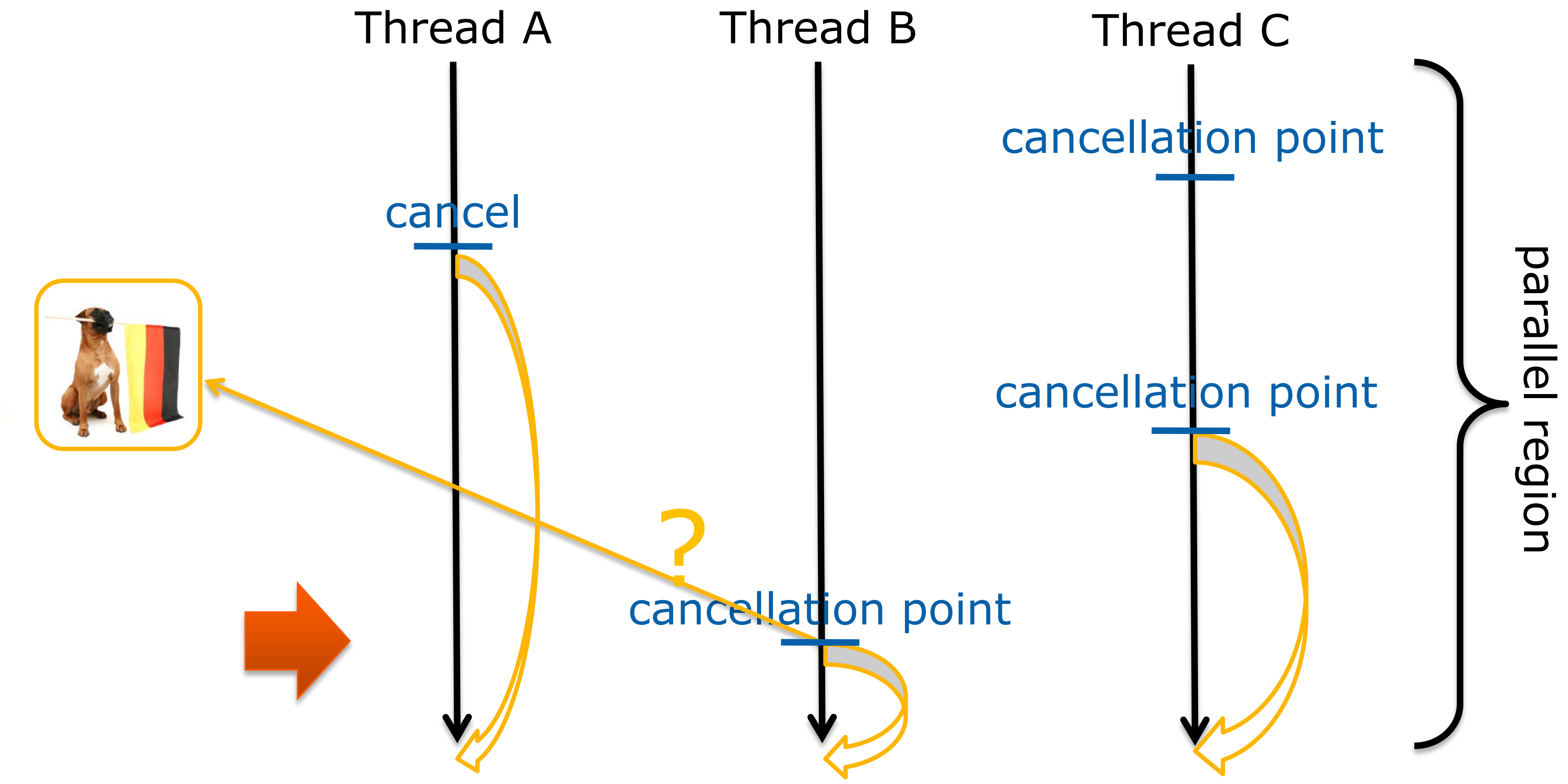
Cancellation Semantics



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, HIP, and OpenCL stream-based offloading
  - → In general: any other API/model that executes asynchronously with OpenMP (tasks)
- Example: HIP memory transfers

```
do_something();
hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
do_something_else();
hipStreamSynchronize(stream);
do_other_important_stuff(dst);
```

- Programmers need a mechanism to marry asynchronous APIs with the parallel task model of OpenMP

Try 1: Use just OpenMP Tasks

```
void hip_example() {
#pragma omp task     // task A
    {
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
    }
    #pragma omp task // task B
    {
        do_something_else();
    }
    #pragma omp task // task C
    {
        hipStreamSynchronize(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 hip_example() {
#pragma omp task depend(out:stream)    // task A
    {
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
    }
    #pragma omp task                   // task B
    {
        do_something_else();
    }
    #pragma omp task depend(in:stream) // task C
    {
        hipStreamSynchronize(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_handle_t` datatype
- Detached task clause

```
detach(event)
```

- Runtime API

```
void omp_fulfill_event(omp_event_t event)
```

Detaching Tasks

```
omp_event_handle_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
3. Signal event for completion
4. Task completes and `taskwait` can continue

Putting It All Together

```
void callback(hipStream_t stream, hipError_t status, void *cb_dat) {
 ③omp_fulfill_event(*((omp_event_handle_t *) cb_data));
}
void hip_example() {
    omp_event_handle_t hip_event;
#pragma omp task detach(hip_event) // task A
    {
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
        hipStreamAddCallback(stream, callback, &hip_event, 0);
  ①}
#pragma omp task                    // task B
        do_something_else();

#pragma omp taskwait ②④
#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 callback(hipStream_t stream, hipError_t status, void *cb_dat) {
  ②omp_fulfill_event(*((omp_event_handle_t *) cb_data));
}
void hip_example() {
    omp_event_handle_t hip_event;
#pragma omp task depend(out:dst) detach(hip_event) // task A
    {
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
        hipStreamAddCallback(stream, callback, &hip_event, 0);
  ① }
#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