Programming OpenMP

Task Dependencies

Christian Terboven

RWTH AACHEN UNIVERSITY

Improving Tasking Performance: Task dependences

Motivation

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

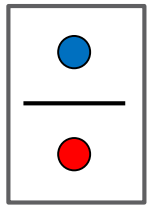
OpenMP 3.1

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

  #pragma omp taskwait

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

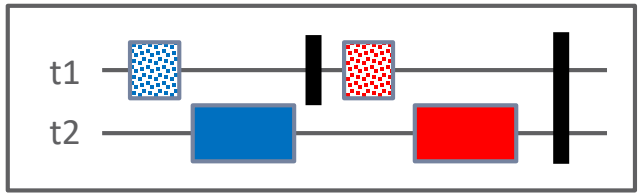
OpenMP 4.0

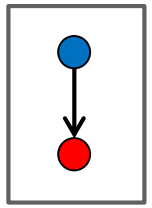
```
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(in: x)
                                d::endl;

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

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

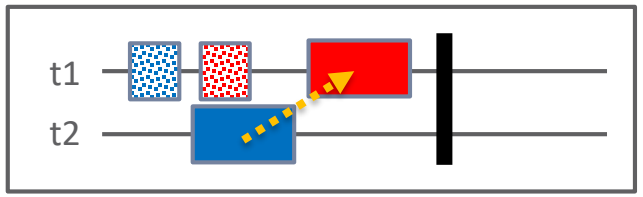


OpenMP 3.1





OpenMP 4.0



Task's creation time

Task's execution time

What's in the spec

What's in the spec: a bit of history

OpenMP 4.0

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

OpenMP 4.5

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

OpenMP 5.0

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

What's in the spec: syntax depend clause

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

where:

- `depend-modifier` is used to define iterators
- `dependency-type` may be: `in`, `out`, `inout`, `mutexinoutset` and `depobj`
- A `list-item` may be:
  - C/C++: A `lvalue` expr or an array section `depend(in: x, v[i], *p, w[10:10])`
  - Fortran: A variable or an array section `depend(in: x, v(i), w(10:20))`

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

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

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

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

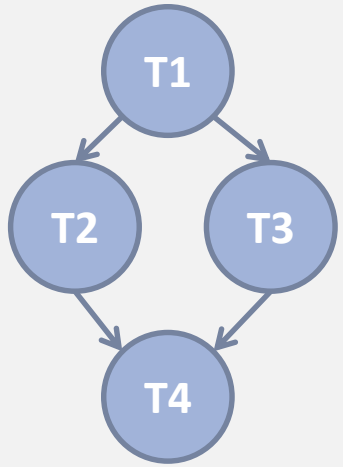
What's in the spec: depend clause (1)

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

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

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

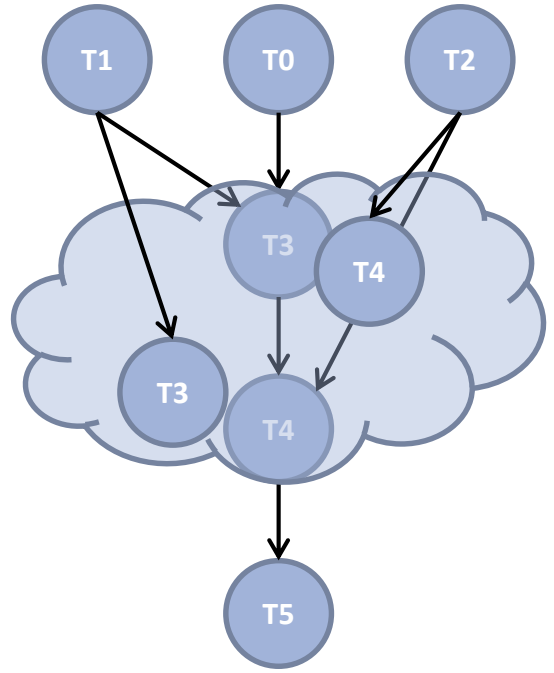
```
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(inout: x) //T1
  { ... }

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

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

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



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

- New dependency type: `mutexinoutset`

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

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

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

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

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

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



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

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

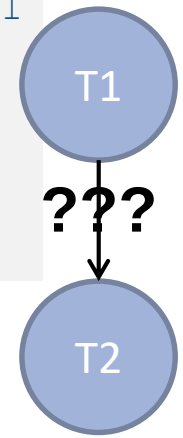
What's in the spec: depend clause (4)

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

```
//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);
}
```

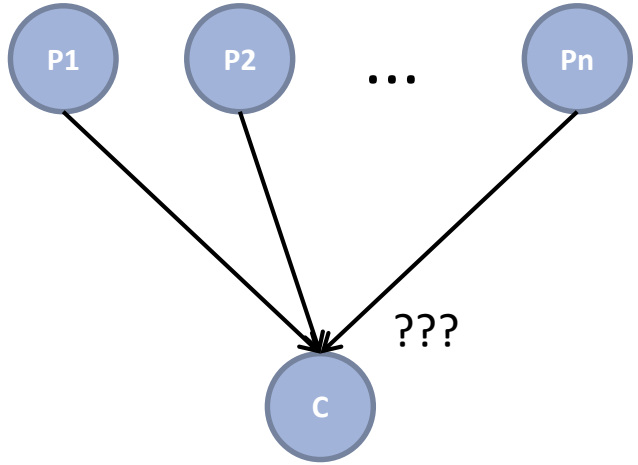


What's in the spec: depend clause (5)

- 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(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)

```cpp
//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;
}
```

```cpp
//test1_v2.cc
//test1_v3.cc
//test1_v4.cc
int x = 0, y = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(inout: x, y) //T1
  {
    x++;
    y++;
  }
  #pragma omp task depend(in: x)       //T2
  std::cout << x << std::endl;

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

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

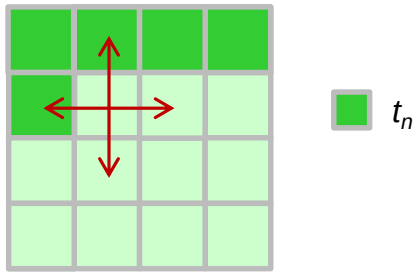
Use case

Use case: intro to Gauss-seidel

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

Access pattern analysis

For a specific t, i and j

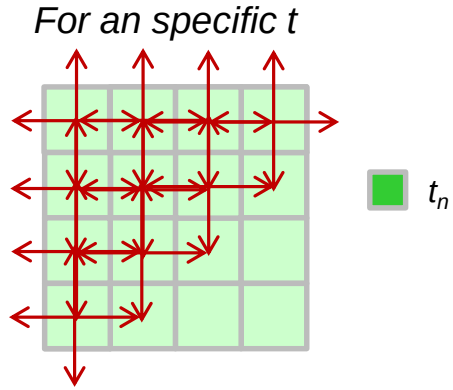


Each cell depends on:
- two cells (north & west) that are computed in the current time step, and
- two cells (south & east) that were computed in the previous time step

Use case: Gauss-seidel (2)

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

1st parallelization strategy



We can exploit the wavefront to obtain parallelism!!

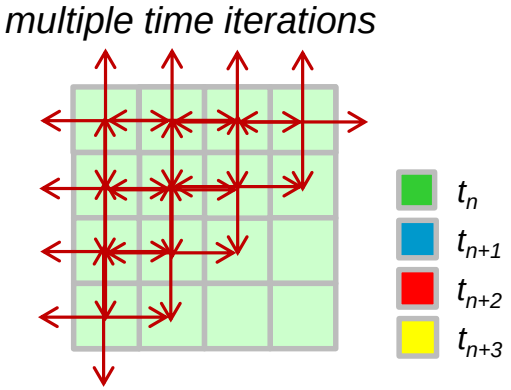
Use case : Gauss-seidel (3)

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

Use case : Gauss-seidel (4)

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

2nd parallelization strategy

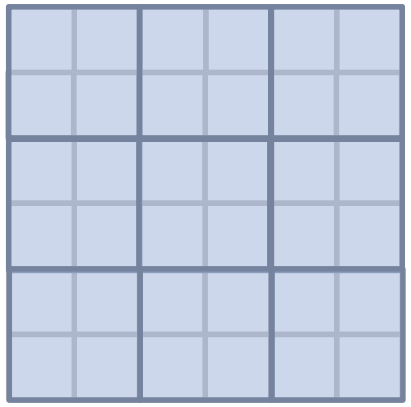


We can exploit the wavefront
of multiple time steps to obtain MORE
parallelism!!

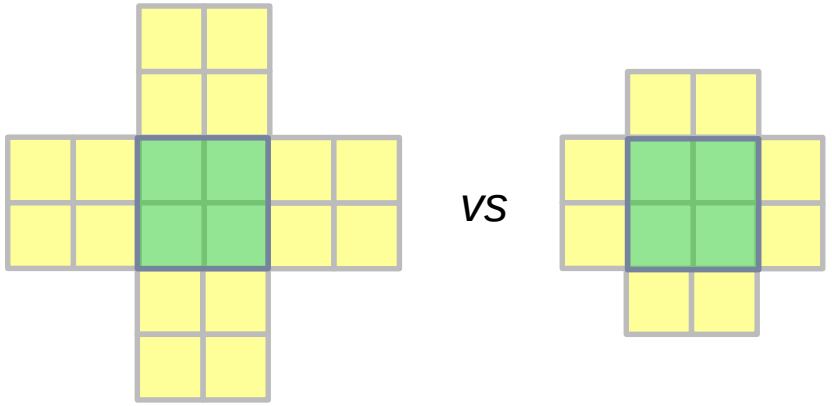
Use case : Gauss-seidel (5)

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

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

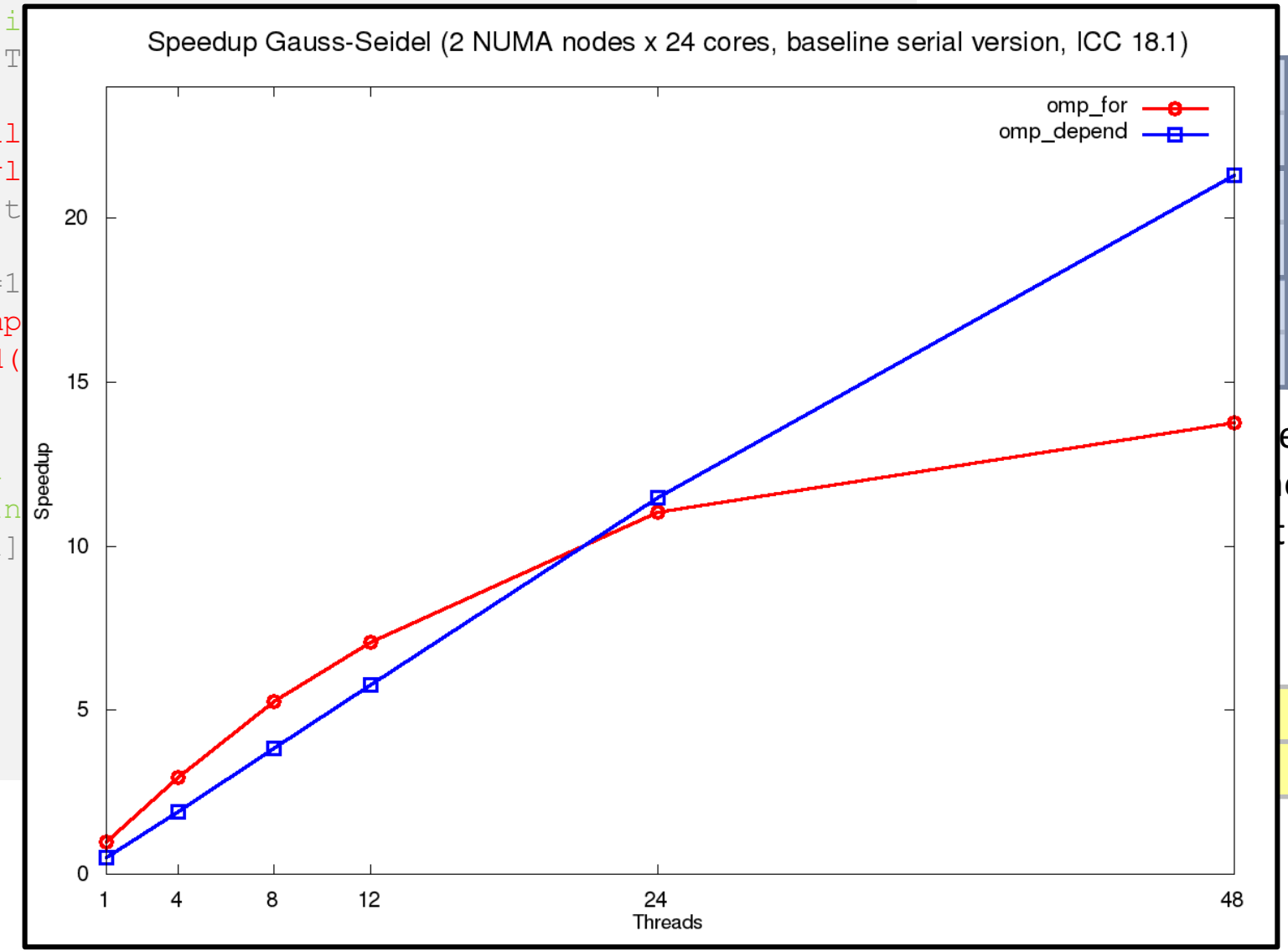


Q: Why do the input dependences depend on the whole block rather than just a column/row?

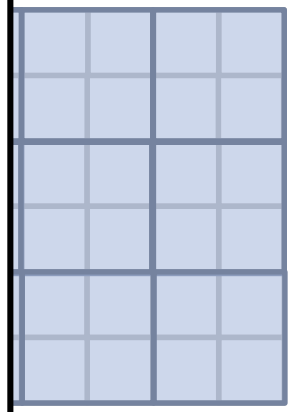


Use case : Gauss-seidel (5)

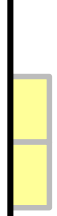
```
void gauss_seidel(i
  int NB = size / T

  #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]


        }
      }
}
```



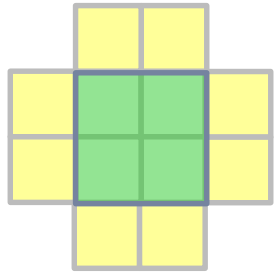
matrix region



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



vs



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;
}
```