




# OpenMP
# Tasking Concepts

TACC OpenMP Team

milfeld/lars/agomez@tacc.utexas.edu

Sides at: tinyurl.com/tacc-openmp or portal.tacc.utexas.edu/training click View Details

# Learning Objective

-- Tasking --

- Review Parallel and Worksharing Concepts
  - Forking, data scoping -- for/do construct, implicit barriers
- Basic Task Syntax and Operations
- Task Synchronization
- Running Tasks in Parallel
- Data-sharing and firstprivate Default for Tasks
- Common Use Cases for Tasks
- Task Dependences
- Taskloop

# OpenMP Pre 3.0

## Data Parallel Paradigms

- About mapping threads to loop-chunks of work
- About work independence for worksharing
  - Determine iteration independence through Read/Write (RW) analysis: mainly RaW, WaR, (WaW of concern & RaR not of concern)
- About avoiding race conditions.

RaW = Read after Write, etc.

Review Parallel and Worksharing Concepts

Forking, data scoping -- for/do construct, implicit barriers

Basic Task Syntax and Operations

Task Synchronization and Operations

Running Tasks in Parallel

Data-sharing and firstprivate Default for Tasks

Common Use Cases for Tasks

Task Dependences

Taskloop

# Parallel & Worksharing

- parallel Construct
  - Forks Threads
    - creates (forks) threads, sets data environment
    - Implied barrier at the end of a region.
- do/for, single, and sections Constructs
    - workshares iterations/block(s) of code
    - do/for
      - Uses prescribed method (static, dynamic, etc.) to schedule threads.
      - iteration limit required

# Worksharing

– Splits work up and gives CHUNKS OF INDEPENDENT work to a team of threads.

– Threads wait at an IMPLIED BARRIER.

– DATA ENVIRONMENT is set by parallel region & altered by workshare constructs: sections/single/do/for. (If you have a team of threads, you must be in a parallel region. )

# Limitation of Worksharing

- Worksharing requires ability to know number of work units before execution (loop count).
- Dynamic scheduling of parallel loop work-chunks is limited. Think of work chunks being queued to execute from a FIFO queue*.
- Ideal for "data parallel", but "task parallel" requires inconvenient construction.

*This can be somewhat limited, but reasonable for monotonically increasing/decreasing work per iteration.

Review Parallel and Worksharing Concepts

Forking, data scoping -- for/do construct, implicit barriers

Basic Task Syntax and Operations

Task Synchronization

Running Tasks in Parallel

Data-sharing and firstprivate Default for Tasks

Common Use Cases for Tasks

Task Dependences

Taskloop

# Getting Started with Tasking

- Easy to begin using for novice
- Uncomplicated syntax and specification for many cases
- Task clauses provide enough flexibility to be efficient (in programming syntax and execution) for more complicated use cases.

Is it easy? – that depends.
It is somewhat like worksharing—
There are simple and complex cases.

# What is Tasking for?

- Irregular Computing:
  - Dynamic Execution (better control of work chunks)
  - Nested Execution
  - Can Employ Dependences (& Priorities)

- Cases for Irregular Computing:
  - Execute independent iterations of while loop in parallel
  - Follow pointers until a NULL pointer is reached, performing independent work at each pointer position.
    - Note: pointer chasing is inherently serial but work at each pointer position must be independent.
  - Follow pointers & perform independent work at nodes in a graph tree
  - Order executions that have task (work) dependences

# Generating a task

- We first go over generating tasks in a serial region – computationally impractical.
- But…it is a practical base for developing concepts.

- Hold on– we'll get to executing tasks concurrently!

# Generating a task

- task is a directive, with clauses (not shown)
  - Encountering thread generates a task (the block of work); by default the task is "deferr<u>able</u>". It can execute the task immediately or defer it.
  - Deferred tasks are queued to be executed after generation, allowing generating task to continue.

Creating two tasks in a serial region.

Tasks do independent work.

C/C++

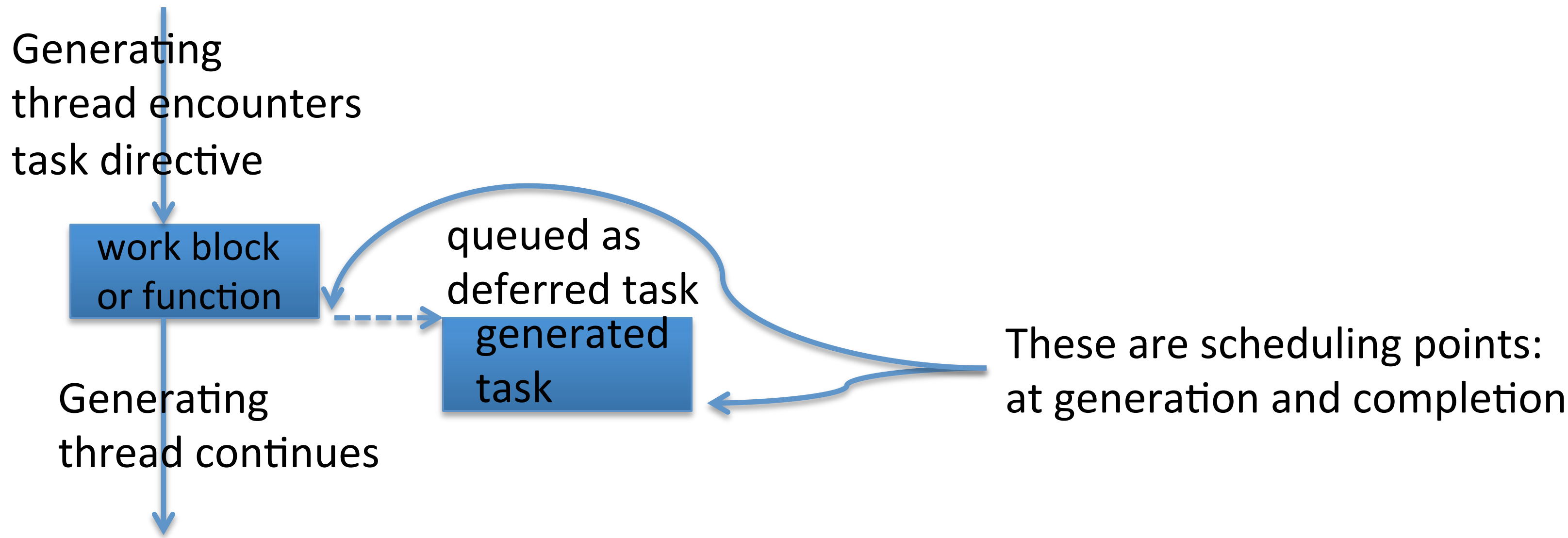
```
#pragma omp task
   foo(j);


#pragma omp task
   for(i=0;i<n;i++){…};
```

F90

```
!$omp task
   foo(j)
!$omp end task

!$omp task
   do i = 1,n; … ;enddo
!$omp end task
```

# Deferred Task

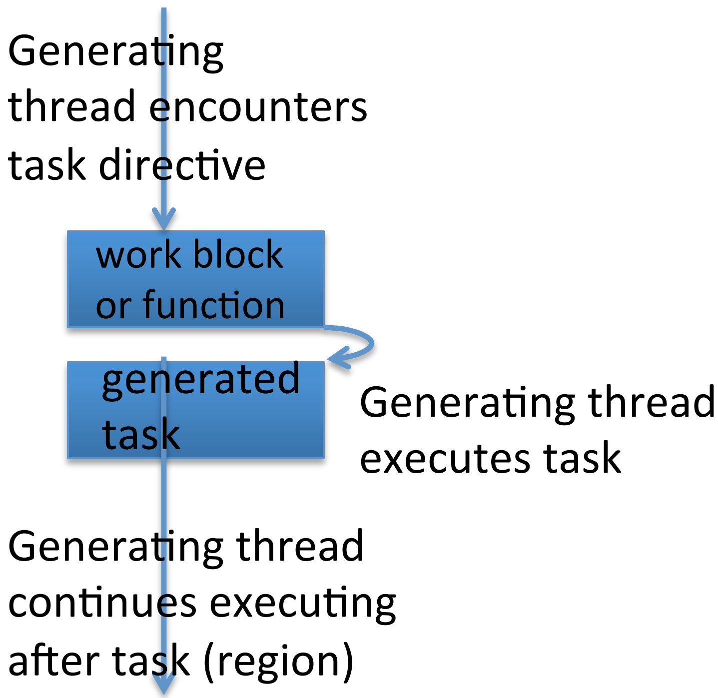
## **Deferred** task.




This thread (at a thread
scheduling point)
or another thread can
execute the queued task.

# Immediate Task

**Immediate** task.



# Tasking Syntax & Operation Explained

- If the previous code is in a serial section of program (usually not the case for a task), the task will probably execute immediately –i.e. will not run the generated task later as a deferrred task (asynchronously).
- "Deferrable" means that it can be executed later (by throwing it on the queue)—this is not required though; it may execute the task immediately (by the task generating thread) if it sees this as optimal.
- Hence, because there is just one thread available, the runtime may execute the task immediately after generation. Also, it may be submitted to the queue as a deferred task. In this latter case it will most likely be pulled from the queue and executed right away, because the runtime realizes there are no other threads to execute it.

# What is a Task

- Work generated by a task construct– yeah, yeah we know that now!  This IS an explicit task.
- Threads of a parallel region, executing the replicated work-- these are also tasks, with the distinctive name: implicit tasks*.
- Standard Spec:  a task is, a “specific instance of executable code and its data environment” generated by”…

In presentation TASK will only mean Explicit Task unless otherwise stated.

*“Each implicit task is tied to a different thread in the team” – more on this later.

- Review Parallel and Worksharing Concepts
    - Forking, data scoping -- for/do construct, implicit barriers
- Basic Task Syntax and Operations
- Task Synchronization
- Running Tasks in Parallel
- Data-sharing and firstprivate Default for Tasks
- Common Use Cases for Tasks
- Task Dependences
- Taskloop

# Synchronizing tasks (sibling tasks)

- If a task is deferred, use the taskwait construct to wait for completion at some point in the code.

C/C++

```c
#pragma omp task
{ foo(j); }


#pragma omp task
{ for(i=0;i<n;i++){…}; }

#pragma omp taskwait
```

F90

```fortran
!$omp task
   call foo(j)
!$omp end task

!$omp task
   do i = 1,n; … ;enddo
!$omp end task
!$omp taskwait
```

# Synchronizing tasks (nested tasks)

- A taskgroup construct waits for all sibling and their descendants. (Yes, tasks can generate child tasks.)

C/C++

```
#pragma taskgroup
{
   #pragma omp task
      foo(j);


   #pragma omp task
   {  for(i=0;i<n;i++)
      #pragma omp task
         foo(i);
    }
}
```

F90

```
!omp taskgroup
  !$omp task
     call foo(j)
  !$omp end task

  !$omp task
     do i = 1,n
     !$omp task
        call foo(i);
     !$omp end task
     enddo
  !$omp end task
!$omp end taskgroup
```

Nested Tasks

Review Parallel and Worksharing Concepts

Forking, data scoping -- for/do construct, implicit barriers

Basic Task Syntax

Task Synchronization

**Running Tasks in Parallel**

Data-sharing and firstprivate Default for Tasks

Common Use Cases for Tasks

Task Dependences

Taskloop

# Generating Concurrent Tasks—Running in a parallel region

- First, create a team of threads, to work on tasks.
- Use a single thread to generate tasks.

C/C++

```
#pragma omp parallel
{
  #pragma omp single
  {
    //generate multipe tasks

  }
}
```

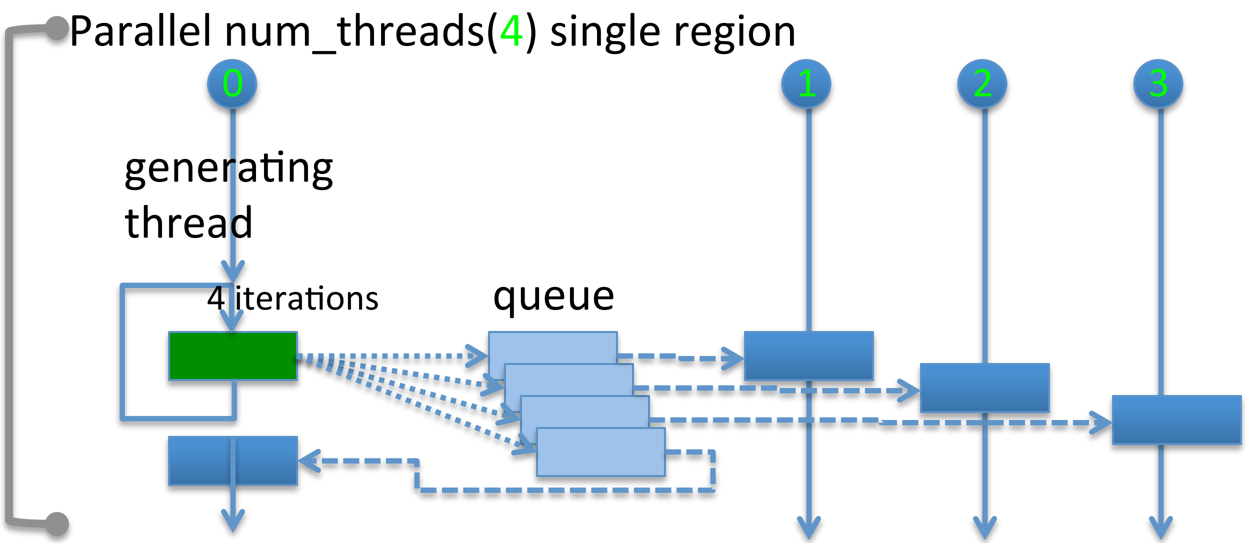
F90

```
!$omp parallel

  !$omp single

    !generate multiple tasks

  !omp end single
!$omp end parallel
```

Generate multiple tasks:  with loop (while/do while/do/for) or recursion

# Tasks in parallel region

- Threads of Team will dequeue & execute tasks
- Shared variables of parallel region are also shared by tasks
- Tasks obey explicit and implicit barriers, also taskwait & taskgroup



Review Parallel and Worksharing Concepts

Forking, data scoping -- for/do construct, implicit barriers

Basic Task Syntax and Operations

Task Synchronization

Running Tasks in Parallel

Data-sharing and firstprivate Default for Tasks

Common Use Cases for Tasks

Task Dependences

Taskloop

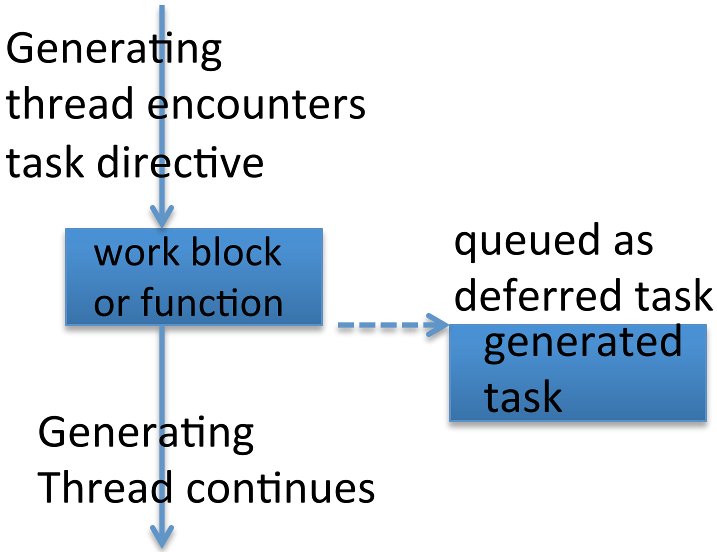
# Tasks: Basic Data-sharing Attributes

- If the task generating construct is in a parallel region any shared variables remain shared.
- for/do construct variables of a worksharing loop are private (see spec.)
- Private variables is on an enclosing construct are firstprivate for the task.

Corner Cases:
Otherwise, if a variable data-sharing attribute is not determined by prescribed rules, it is firstprivate.

# Deferred Task

**Deferred** task.




Basic Concept:

Any thread can pick up a task– but argument/var values at the generation time are needed to determine the work AFTER task is dequeued.

In a parallel region j is shared, and hence needs to be declared firstprivate.

Corner Case: In a serial region j IS NOT shared and hence will be automatically firstprivate.

```
#pragma omp task
  foo(j);


j++;
```

```
!$omp task
   foo(j)
!$omp end task

j = j+1
```

# Scheduling Optimization

- For a small number of threads and tasks, and a large diversity in task work—an imbalance will occur. Even with moderate diversity and large thread and task counts, an imbalance my be present.  The priority clause can alleviate this problem.

```
for (i=0;i<N; i++) {

     #pragma omp task priority(i)
     compute_array(array[i], M);

}
```

```
do i=1,N
   !$omp task priority(i)
   call compute_array(matrix(:, i), N)
   !$omp end task
enddo
```

Review Parallel and Worksharing Concepts

Forking, data scoping -- for/do construct, implicit barriers

Basic Task Syntax and Operations

Task Synchronization

Running Tasks in Parallel

Data-sharing and firstprivate Default for Tasks

**Common Use Cases for Tasks**

Task Dependences

Taskloop

# While loop

- firstprivate clause is necessary, since *cntr* is shared and value must be captured for work.

```c
   int cntr = 100;
#pragma omp parallel
#pragma omp single

while(cntr>0){
 #pragma omp task \
              firstprivate(cntr)

   printf("cntr=%d\n",cntr);

   cntr--;

}
```

```fortran
 integer cntr = 100
!$omp parallel
!$omp single

do while(cntr>0)
   !$omp task &
         firstprivate(cntr)

      print*,"cntr= ",cntr
   !$omp end task

   cntr = cntr - 1
enddo
!$omp end single
!$omp end parallel
```

# Exploiting tasks within while loop

- The loop is executed SERIALLY, but concurrently with the dequeued tasks.
  - So, the non-tasking loop parts should not be costly.
  - Any generated tasks can be picked up directly by other team members.

```
#pragma omp single
while(cntr>0){
  #pragma omp task firstprivate(cntr)
    printf(“cntr=%d\n”,cntr);

  cntr--;
}
```

Serial – generation

parallel – Tasking

Serial – incrementer

# Pointer Chasing

- ***ptr*** points to a C/C++ structure or F90 defined type

```c
   int *ptr;
…//initialize pointer
#pragma omp parallel
#pragma omp single

while(ptr){
  #pragma omp task firstprivate(ptr)
    process(ptr);

  ptr = ptr->next;
}
```

```fortran
 integer,pointer :: ptr
…! initialize pointer
!$omp parallel
!$omp single

do while(associated(ptr))
  !$omp task firstprivate(ptr)
       process(ptr)
  !$omp end task

 ptr = ptr%next
enddo

!$omp end single
!$omp end parallel
```

# Using Recursion in C/C++

- Starting point for tree searches.
- Note, the recursion function, not the "process", is directly tasked.

```
#pragma omp parallel
#pragma omp single
   chase(ptr);
…
void chase(node *ptr){

  if(ptr->next){
   #pramga omp task
   chase(ptr->next);
  }
   process(ptr)
}
```

Just another way
to build a queue
of tasks– but with
recursion (bunch
of context switching)

# Undeferred Tasks with **if** clause

```
while(ptr){

  usec=ptr->cost*factor;

  #pramga omp task if(usec>0.01) firstprivate(ptr)
  process(ptr)

  ptr = ptr->next;
}
```

If execution time (usec) is less than 0.01 microseconds, the ***if*** argument is false:

- Generating thread will suspend generation
- Generating thread will execute the task
- Generating thread will resume generation

# Tied Tasks (default)

- Tasks can be stopped and continued (at scheduling points). Use taskyield directive to create your own scheduling point.

- A Tied Task can only be continued by the thread that initially began the execution of the task. This can be important for keeping cached data available (hot) to the thread, and for locks.

# Untied Tasks

Syntax: … `omp task untied`

- Any thread may continue execution of the task
- Important for controlling task generation when queue becomes full:
  A single untied generator task can let another thread continue.

1. When queue is full, the task generation thread may execute queued tasks.
2. If it gets stuck, another thread can execute the untied generation task.

# Untied Task Example

```
#pragma omp task untied
{
    for(i=0; i<Nlarge;i++)
      #pragma omp task
      process(i);
}
```

Details: read this later:

- A single (outer) generating task generates a large number of tasks. These are put on a queue to execute.
- When the queue is filled, the runtime will ask the thread executing the task generation to switch at the next task generation (task constructs are task switching points) and execute a generated task (a queued task). That is, suspend the task generating task and pick up tasks from the queue.
- If there is an imbalance in work and the thread that was executing task generation gets stuck in processing one of the generated tasks, and other threads many finish their tasks. Because the "stuck" thread is tied to the generating task, no other thread can pick up the generation task to continue putting tasks on the queue.
- However, if the (outer) generating task is untied, any other thread can resume the generation task and continue to generate.

Review Parallel and Worksharing Concepts

Forking, data scoping -- for/do construct, implicit barriers

Basic Task Syntax and Operations

Task Synchronization

Running Tasks in Parallel

Data-sharing and firstprivate Default for Tasks

Common Use Cases for Tasks

Task Dependences

Taskloop

# Depend clause

- Dependences are derived from *dependence-type* and the *list* items of the depend clause.

Syntax: … `task depend(` *dependence-type* `:` *list* `)`

*dependence-type*:

**in** If a storage location of at least one list item is the same as a list item appearing with an **out** or **inout** *dependence-type* of a previously generated task, the **in** task will be dependent on the **out/inout** task.

**out**
**inout** If a storage location of at least one list item is the same as a list item appearing with an **in, out** or **inout** *dependence-type* of a previously generated task, then the generated task will be a dependent task of the previously generated task.

Comment: What?

# Depend clause (in other words)

- When a task requires a previously computed object, it has an **in**put (**in**) dependence.
- When a task is computing a value for a object that may be required later, it has an **out**put (**out**)dependence.
- When a tasking requires a previous computed object and computes a new value it has an input-output (**inout**) dependence.
- Dependences are formed through the object list – variable, element references, array sections, etc. The list items may be objects used in the routine.

Comment: OK, this is starting to make sense.

# Depend clause (another approach – RW)

- Enforces scheduling constrains of sibling tasks (or ordered loop iterations – not discussed here).

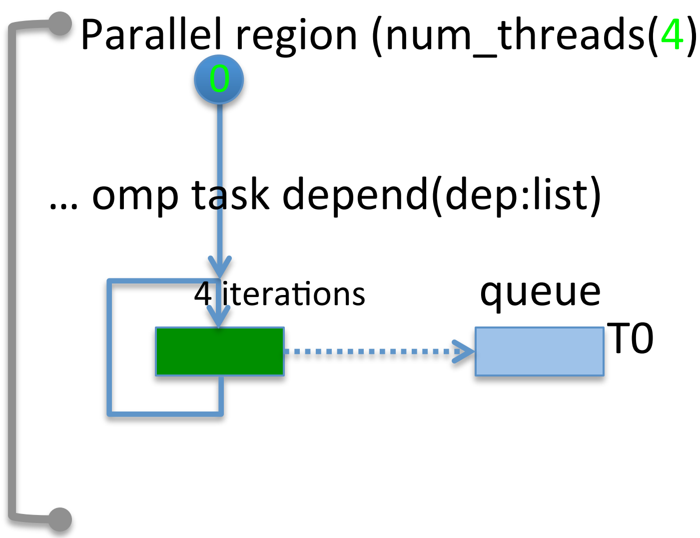
Syntax: … `task depend(` *dependence-type* `:` *list* `)`

*dependence-type:* == what the task needs to do

| | |
|---|---|
| **in** | think of as a read |
| **out** | think of as a write |
| **inout** | think of as a read and then a write |
| | |
| *list* | Item it needs to "Read" or "Write" |

- Dependences

**Task 1 execution dependence**

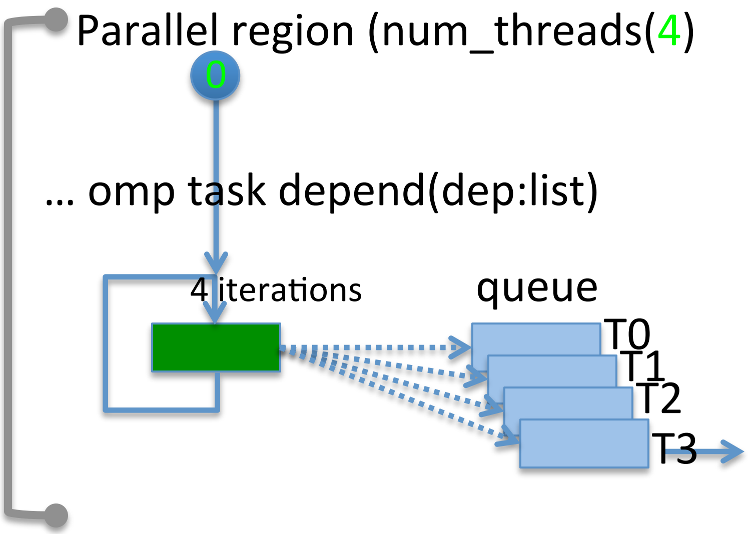



When runtime executes T0, it checks previously generated tasks for identical list times.

There are no previous tasks– so this task has NO dependence. EVEN if it has an IN (read) dependence type!


40

- Dependences




**Task 3 execution dependence**

T3 checks previously generated tasks for identical list times.

If an identical list item exists in previously generated tasks,T3 adheres to the *dependence-type*.

# Task depend clause

- Flow Control (RaW, Read after Write)

```
    x = 1;
#pragma omp parallel
#pragma omp single
{
   #pragma omp task shared(x) depend(out: x)
      x = 2;
   #pragma omp task shared(x) depend( in: x)
      printf("x = %d\n", x);
}
...
```

T1 is put on queue, sees no previously queued tasks with x identifier→ No dependences.

T2 is put on queue, sees previously queued task with identifier→ Has RaW dependence.

Print value is always 2.

# Task depend clause

- Anti-dependence (WaR, write after read)

```
   x = 1;
#pragma omp parallel
#pragma omp single
{
   #pragma omp task shared(x) depend( in: x)
      printf("x = %d\n", x);
   #pragma omp task shared(x) depend(out: x)
      x = 2;
}
...
```

T1 is put on queue, sees no previously queued tasks with x identifier→ No dependences.

T2 is put on queue, sees previously queued task with identifier→ has WaR dependence.

Print value is always 1.

# Task depend clause

- Output Dependence (WaW, Write after Write)

```
    x = 1;
#pragma omp parallel
#pragma omp single
{
   #pragma omp task shared(x) depend(out: x)
      printf("x = %d\n", x);
   #pragma omp task shared(x) depend(out: x)
      x = 2;
}
```

T1 is put on queue, sees no previously queued tasks with x identifier→ No dependences.

T2 is put on queue, sees previously queued task with identifier→ has WaW dependence.

Print value is always 1.

# Task depend clause

- (RaR, no dependence)

```
    x = 1;
#pragma omp parallel
#pragma omp single
{
   #pragma omp task shared(x) depend(in: x)
      printf("x = %d\n", x);
   #pragma omp task shared(x) depend(in: x)
      x = 2;
}
```

T1 is put on queue, sees no previously queued tasks with x identifier→ No dependences.

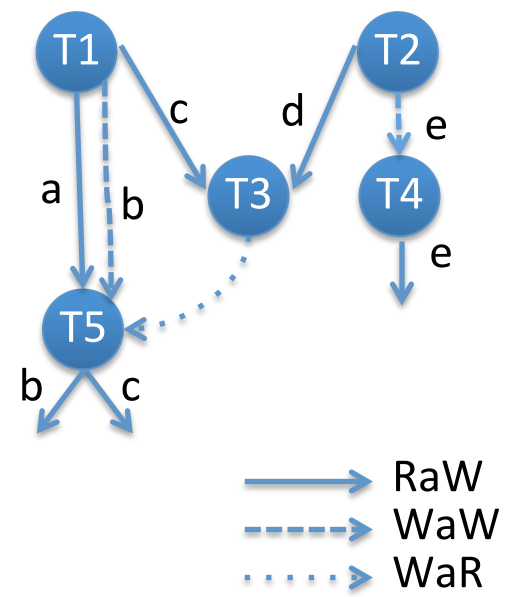
T2 is put on queue, sees previously queued task with x identifier→ has NO ordering (because it is RAR)

Print value is 1 or 2.

# Task depend clause

- Following a graph

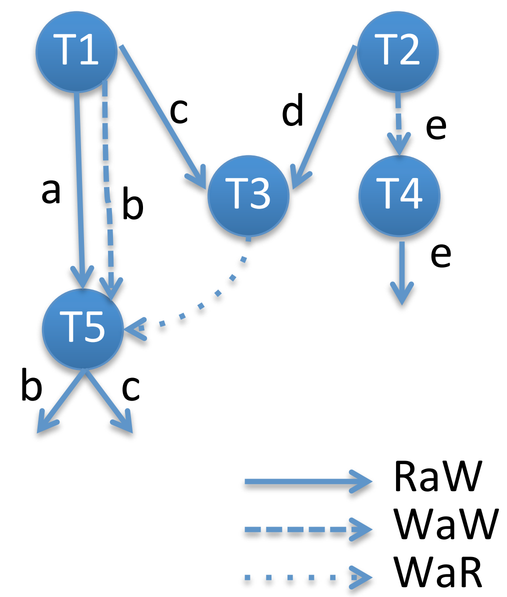
```
  #pragma omp parallel
  #pragma omp single
  {
T1  #pragma omp task depend(out:a,b,c)
      f1(&a, &b, &c);
T2  #pragma omp task depend(out:d,e)
      f2(&d, &e);
T3  #pragma omp task depend(in:c,d)
      f3(c,d);
T4  #pragma omp task depend(out,e)
      f4(&e);
T5  #pragma omp task depend(in:a) depend(out:b,c)
      f5(a,&b,&c)
  }
```



# Task Depend Clause

- Following non-computed variables-- works, too.

```
 #pragma omp parallel
 #pragma omp single
 {
T1 #pragma omp task depend(out:t1,t2,t3)
      f1(&a, &b, &c);
T2 #pragma omp task depend(out:t4,t5)
      f2(&d, &e);
T3 #pragma omp task depend(in:t3,t4)
      f3(c,d);
T4 #pragma omp task depend(out,t5)
      f4(&e);
T5 #pragma omp task depend(in:t1)depend(out:t2,t3)
      f5(a,&b,&c)
 }
```



Review Parallel and Worksharing Concepts

Forking, data scoping -- for/do construct, implicit barriers

Basic Task Syntax and Operations

Task Synchronization

Running Tasks in Parallel

Data-sharing and firstprivate Default for Tasks

Common Use Cases for Tasks

Task Dependences

Taskloop

# taskloop

- Iterations of loops are executed as tasks (of a taskgroup)
  - <u>Single generator needed</u>
  - All team members are not necessary
  - Implied taskgroup for the construct

Syntax: `... omp taskloop [clauses]`

Some clauses:

| | | |
|---|---|---|
| grainsize | | number of iterations assigned to a task. |
| numtasks | or | create this number of tasks |
| | default: | number of tasks & iterations/task implementation defined |
| untied | | tasks need not be continued by initial thread of task |
| nogroups | | don't create a task group |
| priority | | for each task (default 0) |

# Taskloop

- ptr points to a structure in C/C++, defined type in F90

```
void parallel_work(void) {  // execute by single in parallel
   int i, j;
   #pragma omp taskgroup
   {
     #pragma omp task
     long_running(); // can execute concurrently

     #pragma omp taskloop private(j) grainsize(500) nogroup
     for (i = 0; i < 10000; i++)   //can execute concurrently
       for (j = 0; j < i; j++)
          loop_body(i, j);

   }  // end taskgroup
}
```

# Summary

- Tasks are used mainly in irregular computing.
- Tasks are often generated by a single thread.
- Task generation can be recursive.
- Depend clause can prescribe dependence.
- Priority provides hint for execution order.
- First private is default data-sharing attribute, shared variable remain shared.
- Untied generator task can assure generation progress.