Tasking

Review Worksharing and Limitations

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

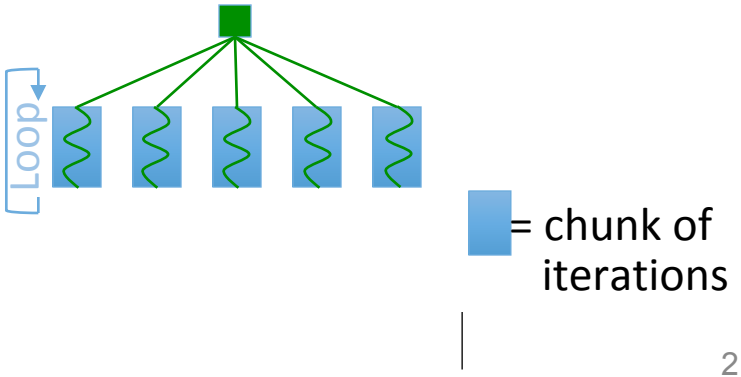
Worksharing

- After forking a TEAM OF THREADS does work
- LOOP ITERATION PARTITIONING (chunks of independent work)
- DATA ENVIRONMENT altered by clauses & implicits (e.g. private).
- IMPLIED BARRIER forces threads to wait at end.

```
omp parallel
{
}
```

→

```
omp parallel
{
   <Worksharing
     Loops>
}
```



Worksharing Limitation

- Requires Loop Count.
- Dynamic Scheduling– Only FIFO queue.
- Worksharing is for “data parallel”.
  Tasking is for “task parallel”.

Learning Objective

-- Tasking --

- Review Worksharing and Limitations
- 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

Tasking

- After forking a TEAM OF THREADS does work
- ONE THREAD OF TEAM GENERATES TASKS of independent work, others execute the task.
- DATA ENVIRONMENT altered by clauses & implicits (e.g. firstprivate).
- EXPLICIT WAIT required for synchronization.

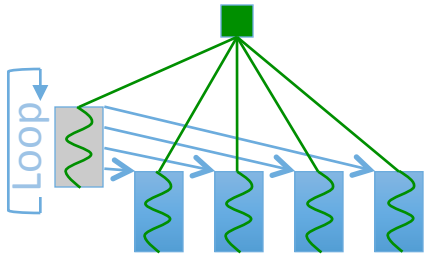
```
omp parallel
{



}
```

→

```
omp parallel
{
   omp single
   <Loop Generates
     TASKS>
}
```



= chunk of iterations

Generating A task (in a serial region)

task directive forms a task from a block of code.

- A task can be executed immediately or later (it is "deferrable").
- Deferred tasks are queued.

These Tasks do Independent Work.

// Master Thread

C/C++

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

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

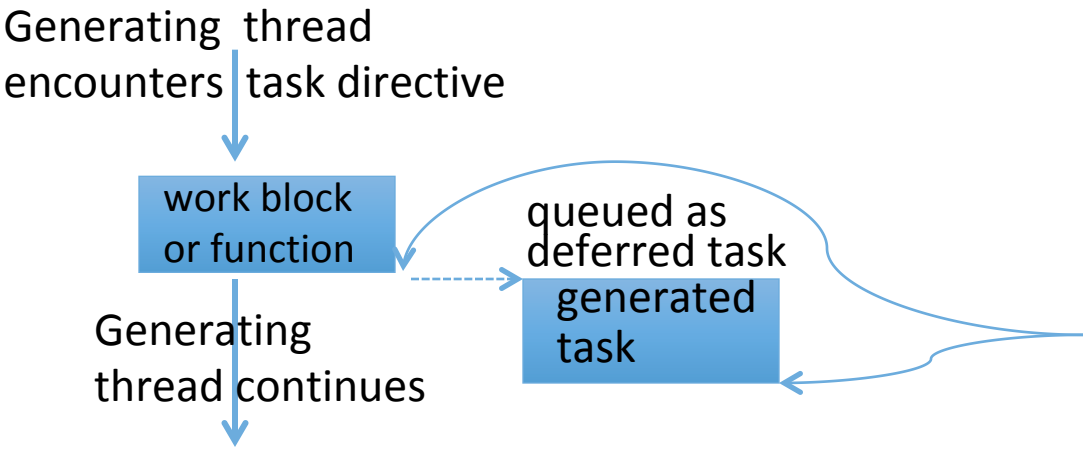
!! Master Thread

F90

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

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

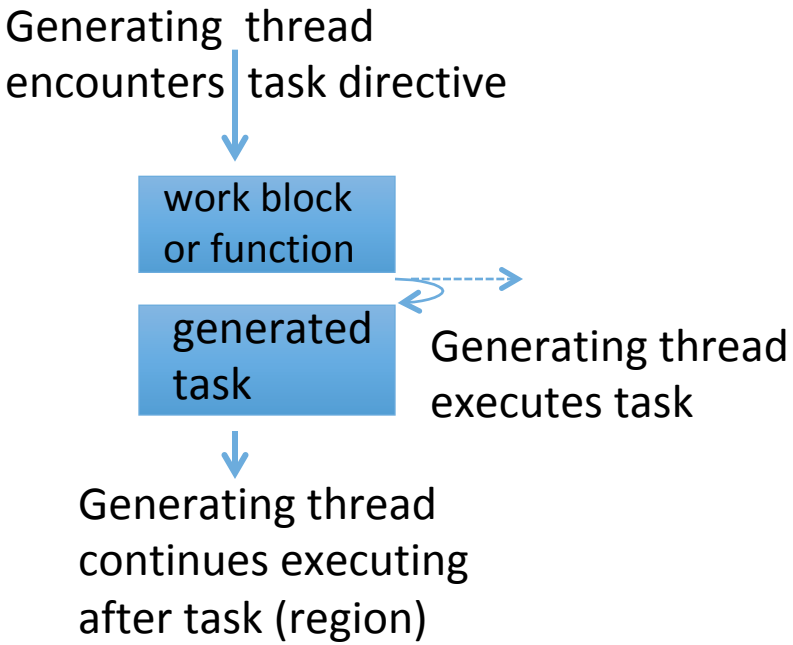
Deferred Task (usual case)



This thread
or another thread can
execute the queued task
(at a thread scheduling point,
includes barriers)

Deferred task

Immediate Task



Immediate task

Learning Objective

-- Tasking --

- Review Worksharing and Limitations
- 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

Use taskwait construct to wait for completion of generator's child tasks.

C/C++

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


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

#pragma omp taskwait
```

F90

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

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

Synchronizing tasks

Use taskgroup construct to wait for all child and descendants.

Nested 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
```

Tasking

Review Worksharing and Limitations

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

Generating Concurrent Tasks—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 num_threads(4)
{
  #pragma omp single
  {
    //generate multiple tasks

  }
}
```

F90

```
!$omp parallel num_threads(4)

  !$omp single

    !generate multiple tasks

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

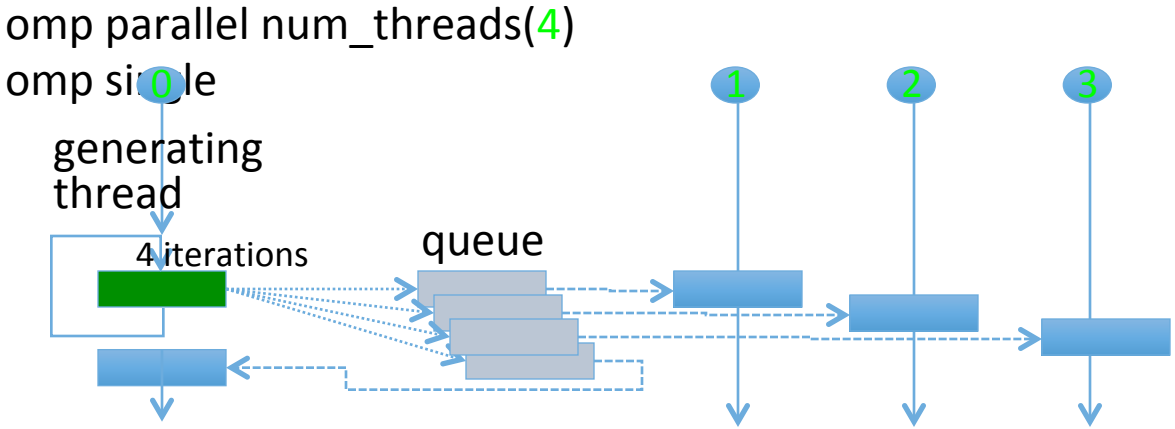
Generate multiple tasks: with loop (while/do/for) or recursion

Tasks in parallel region

Threads of Paralle Team will dequeue & execute tasks

Shared variables of parallel region are also shared by tasks

Tasks obey explicit and implicit barriers, also taskwait & taskgroup



Tasking

- Review Worksharing and Limitations
- 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

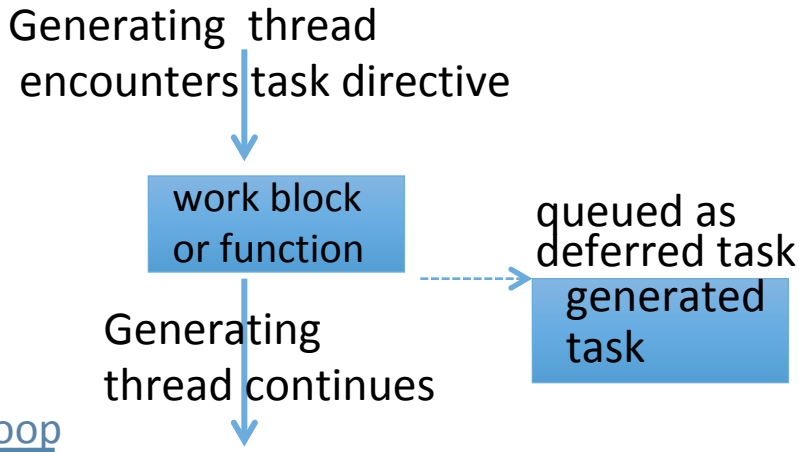
Tasks: Basic Data-sharing Attributes

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

The Point: The index passed to a task needs to be firstprivate.

Deferred Task

Basic Concept:
The argument value at generation time is needed– not later when it is run.



loop

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

j++;
```

loop

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

j = j+1
```

In a parallel region if j is shared, it needs to be declared firstprivate.

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 still be present.

The priority clause is a user-defined way to solve imbalance. (Larger # = higher priority.)

```
for (i=N-1;i<0; i++) {
    #pragma omp task priority(i+1)
    compute_array(array[i], N);
}
```

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

Tasking

- Review Worksharing and Limitations
- 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

What is Tasking for?

Irregular Computing of independent work chunks:

- While loop, execute independent iterations in parallel
- Follow pointers in list until a NULL pointer is reached,
  performing independent work for each pointer position.
  - Note: the pointer chase is inherently serial but if work at each pointer position is independent, then work can be executed in parallel.
- Follow nodes in tree graph & perform independent work at nodes
- Ordered executions that have task (work) with dependences

While loop

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

while(cntr>0){

 #pragma omp task firstprivate(cntr)
 {
  printf("cntr=%d\n",cntr);
  work_long_time(cntr);
 }
 cntr--;
}
```

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

do while(cntr>0)

  !$omp task firstprivate(cntr)
    print*,"cntr= ",cntr
    call work_long_time(cntr)
  !$omp end task

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

firstprivate clause required since *cntr* is shared and value must be captured for work.

Exploiting tasks within while loop

The generating 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)
    work_long_time(cntr);

  cntr--;
}
```

Serial – generation

Parallel – Tasking

Serial – incrementation

Pointer Chasing

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

```
   struct node *ptr;
…//initialize pointer
#pragma omp parallel
#pragma omp single

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


  ptr = ptr->next;
}
```

```
 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
```

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 true → business as usual.

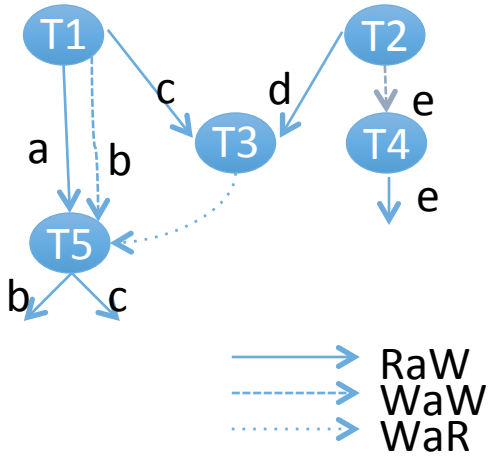
If the ***if*** argument is false task is undeferred (exec time (usec) is less than 0.01).

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

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)
  }
```



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.
- Firstprivate is default data-sharing attribute, shared variables remain shared.
- Untied generator task can assure generation progress.

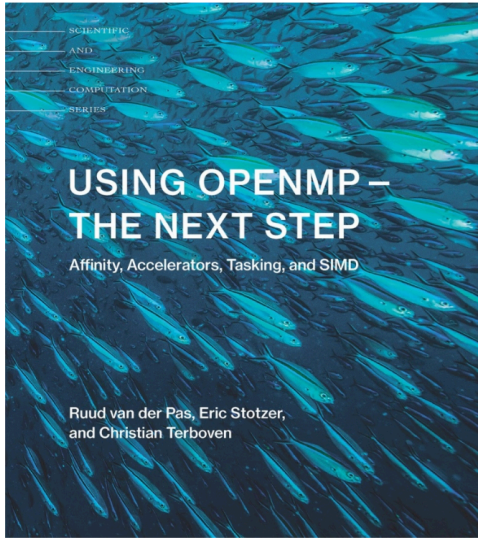
Not discussed here.

--The END--

Questions?

References:

OpenMP Programming: The Next Step



More Steps? We can cover Task Dependences, time permitting.

Tasking

Review Worksharing and Limitations

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

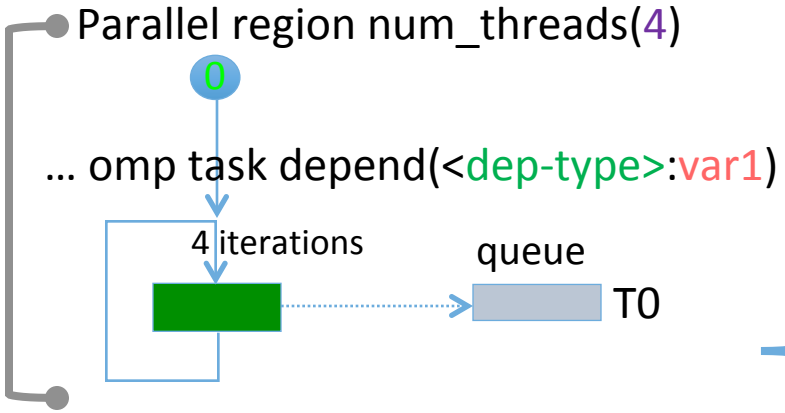
Syntax: … `task depend(` *dependence-type*`:` *list* `)`

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

dependence-type: == what the task needs to do with list item

in	think of as a read
out	think of as a write
inout	think of as a read and then a write
list	item (variable) needing to “Read” or “Write”

Dependences

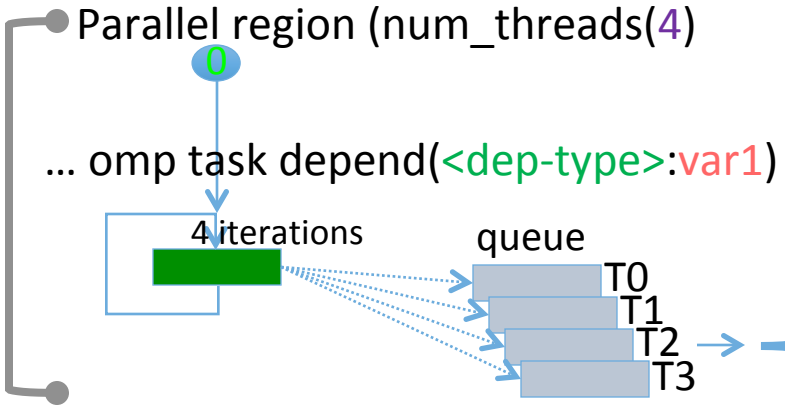


Task 1 execution dependence

When runtime executes T0, it checks previously generated tasks for identical list items (var1).

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

Dependences



Task 3 execution dependence

T3 checks previously generated tasks for identical list times (var1).

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

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.

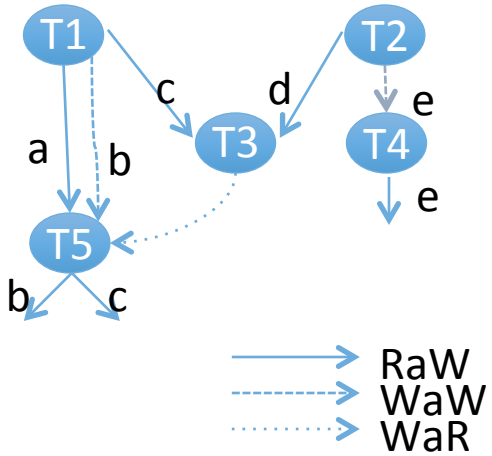
T2 is put on queue, sees previously queued task with x identifier→ has NO ordering (because it is RAR)

Print value is 1 or 2.

Task depend clause

Following a graph

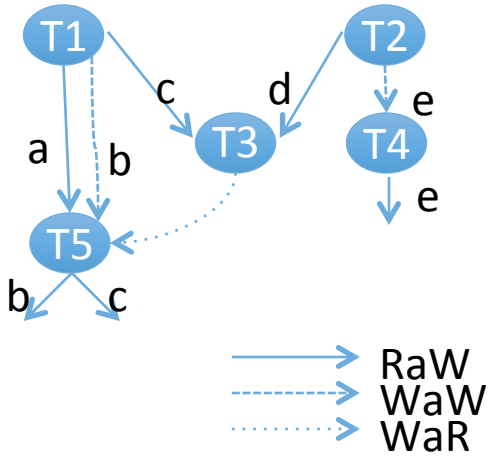
```
  #pragma omp parallel
  #pragma omp single
  {
T1  #pragma omp task depend(out:a,b,c)
      f1(&a, &b, &c);
T2  #pragma omp task depend(out:d,e)
      f2(&d, &e);
T3  #pragma omp task depend(in:c,d)
      f3(c,d);
T4  #pragma omp task depend(out,e)
      f4(&e);
T5  #pragma omp task depend(in:a)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



Tasking

Review Worksharing and Limitations

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)

Single generator needed

All team members are not necessary

Implied taskgroup for the construct

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

- some clauses:

grainsize numtasks	or	number of iterations assigned to a task (See spec 4 details.) number of tasks to be executed concurrently (See spec 4 details.)
	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

```
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 variables remain shared.
- Untied generator task can assure generation progress.