OpenMP Tasking Concepts

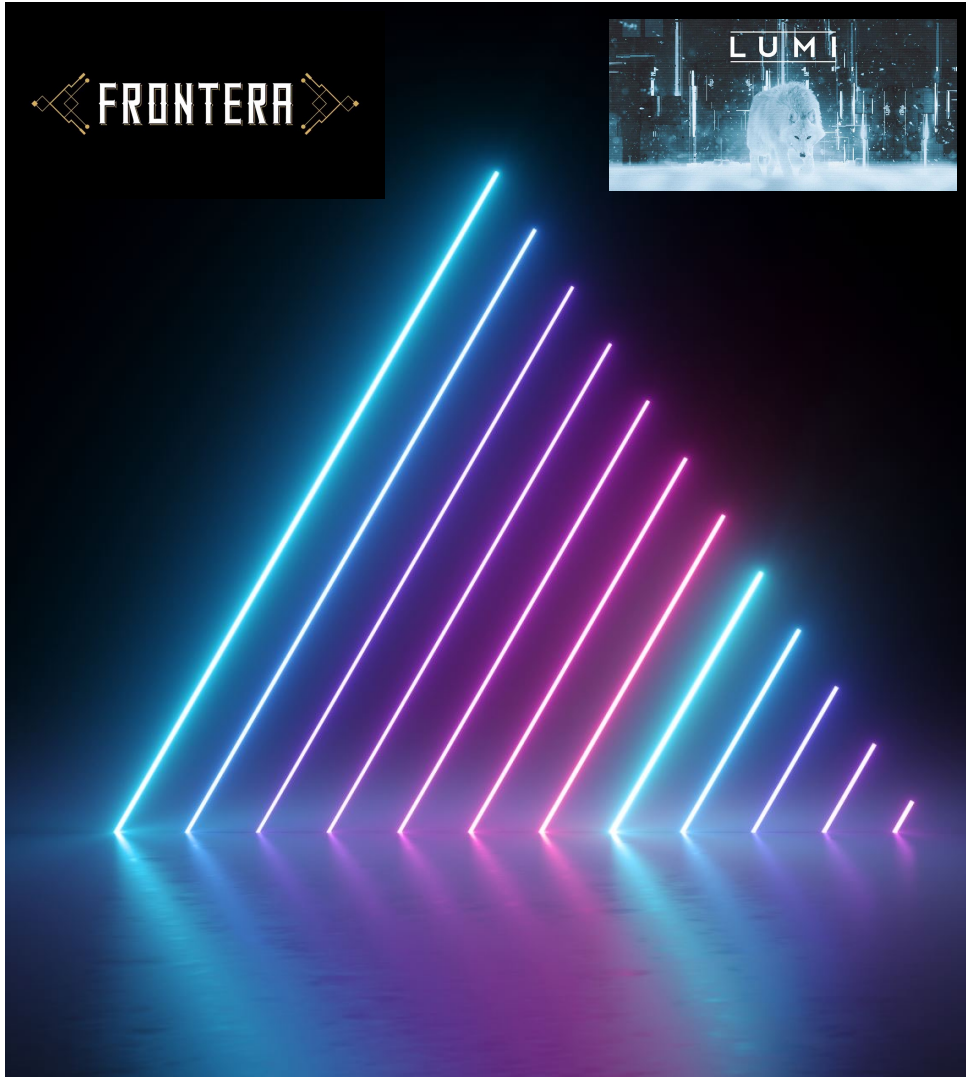
CSC Summer Institute

October 9-11, 2023

Kent Milfeld (TACC)

Emanuele Vitali (CSC)

Lecture and Lab slides available at:



Learning Objective -- will learn

-- Tasking --

worksharing (WS) Limitations after a WS review

difference between implicit/explicit tasks

understand how potential race conditions lead to task dependences

how to create an explicit OpenMP task (*task* construct)

how tasks are queued and executed

how to synchronize tasks

run tasks in parallel

data-sharing attributes and *firstprivate* default for tasks

common use cases for tasks

how to order dependent task with *depend* clause

how to use *taskloops* constructs

the future with *free-agent* threads (in OpenMP v6.0)

Session

Review Worksharing and Limitations

Basic Task Operations and Syntax

Task Synchronization

Running Tasks in Parallel

Data-sharing and firstprivate Default for Tasks

Common Use Cases for Tasks

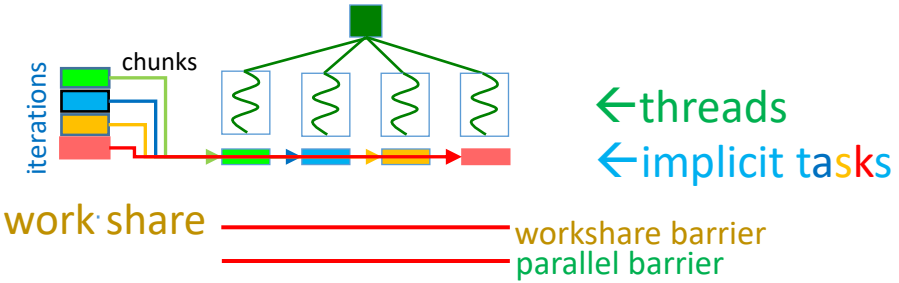
Task Dependences

Taskloop

Worksharing

- Fork a TEAM OF THREADS, encounter workshare directive (loop...)
- LOOP ITERATION ARE PARTITIONED (chunks of independent work)
- DATA ENVIRONMENT: ..., index of workshare loop becomes private
- IMPLIED BARRIER forces threads to wait at end.
- Threads in team are assigned to work = implicit tasks


```
omp parallel
{
  omp for | do
    <workshare Loop>
}
```



Worksharing Limitations

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

  (Tasking is for “task parallel”, and more.)

Terminology

- The executable work assigned to a thread of a parallel team is an implicit task.
- Often it is reasonable to think of a thread and task as the same “thing”.

C/C++

```
#pragma omp parallel for schedule(static)
  for(i=0;i<n;i++) printf("%d\n",i);
```

iteration chunks are implicit task of parallel region

F90

```
!$omp parallel do schedule(static)
  do i=1,n; print*,i; enddo
!$omp end parallel do
```

Terminology

- An OpenMP implicit task has "always" been defined as:

  "A task generated by the implicit parallel region or **generated when a `parallel` construct is encountered during execution**." [v3.0 (2009) – present]

- An OpenMP explicit task has been defined as:

  "A task generated when **a `task` construct** is encountered during execution." [v3.0– v4.5]

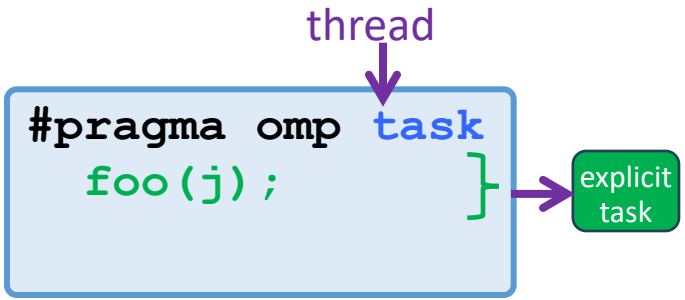
  "A task that is not an implicit task." [v5.0 – present] --what is expected here?

These are the TASKs of interest in this course,
but these have various "flavors".

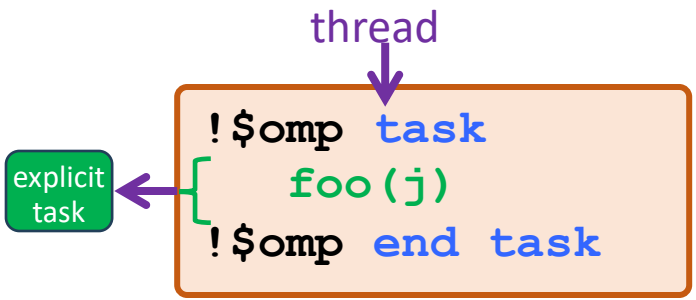
explicit task

- An explicit task is created by a **task** construct for its structure block (of code), when a thread encounters the **task** construct.

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




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

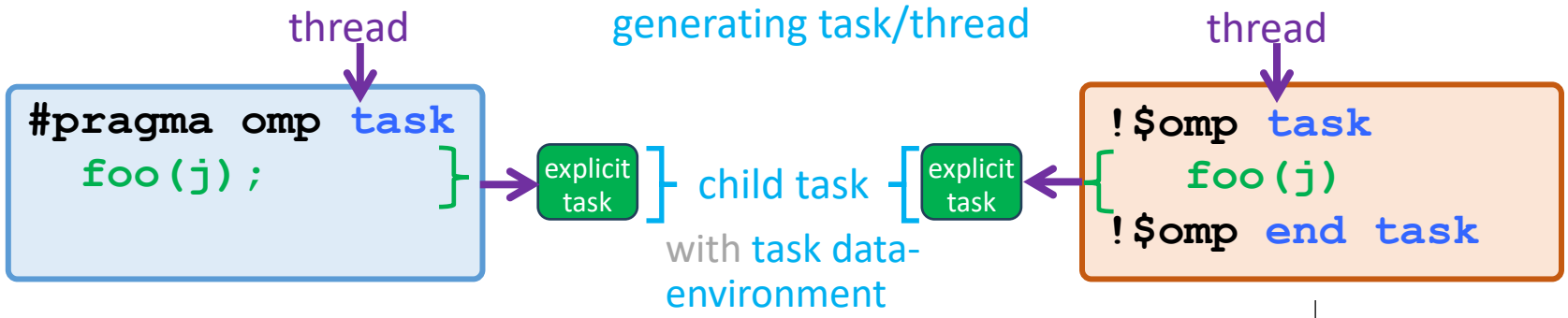


task clauses and terminology

- *child* task has its own data-environment apart from *generating* task
- clauses:

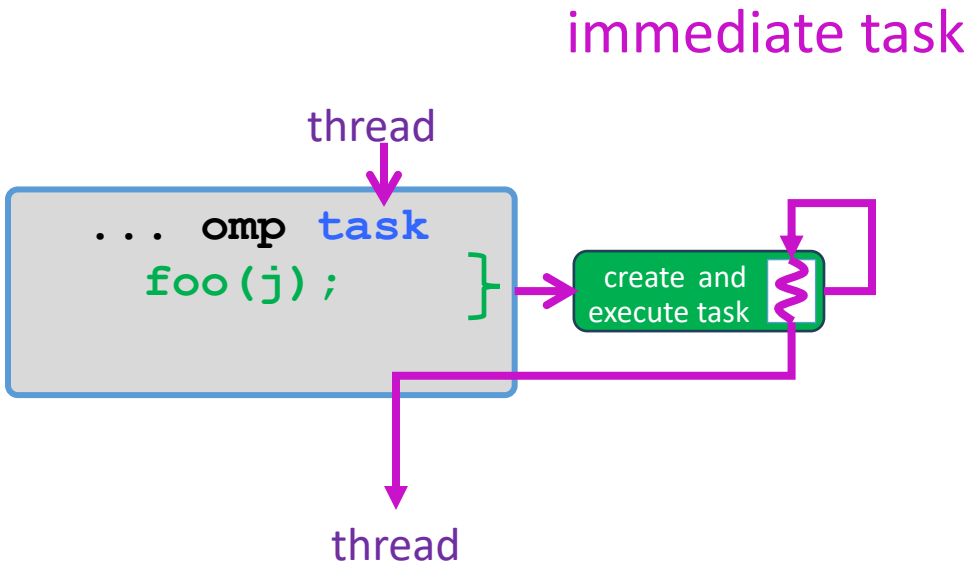
data attributes	scheduling	generation	+ more data attributes
private shared firstprivate in_reduction	depend priority detach if	final mergeable untied	affinity allocate default

- tasks can be nested



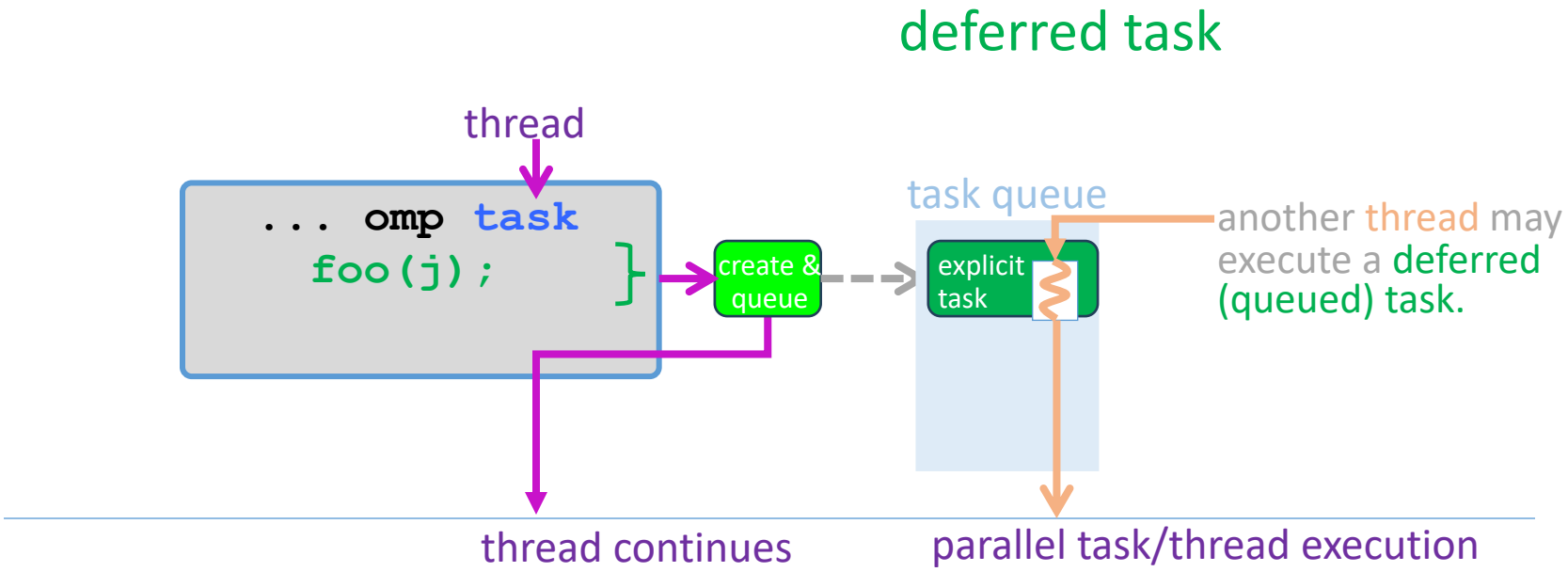
immediate task

- The encountering thread **may immediately execute** the task, or defer its execution.



deferred task

- The encountering thread may immediately execute the task, or **defer its execution** for parallel execution by another thread.



Tasking

Dependence

Review Worksharing and Limitations

Basic Task Operations and Syntax

Task Synchronization

Running Tasks in Parallel

Data-sharing and firstprivate for Tasks

Common Use Cases for Tasks

Task Dependences

Taskloop

Parallel Tasks

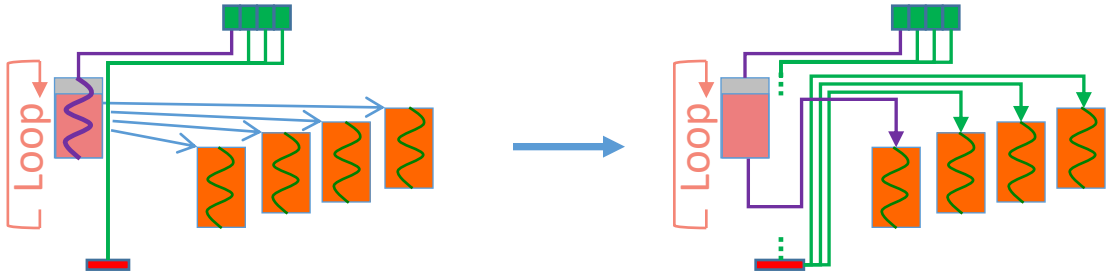
- Need multiple threads to execute queued tasks.
- But only need one thread/task to generate explicit tasks.

- get threads from a parallel construct
- use master/single for generation.

Tasking

- Fork a TEAM OF THREADS
- ONE THREAD OF TEAM GENERATES TASKS of independent work,
  OTHERS EXECUTE THE TASKS. (Other mechanism: TASKS can be in recursions.)
- DATA ENVIRONMENT: ..., region private variables become firstprivate.
- NO IMPLICIT BARRIER, INSERT EXPLICIT WAIT or use BARRIER for syncing.

```
omp parallel
{
   omp master
   {<loop over omp tasks>
                work block=
   }
}
```



Threads waiting at barrier can execute tasks.

A task w.o. a loop generator

(in a serial region)

Task consists of a function or block of code.

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

Task 1

Task 2

Task 1 & 2 do independent work.

// Master Thread

C/C++

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

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

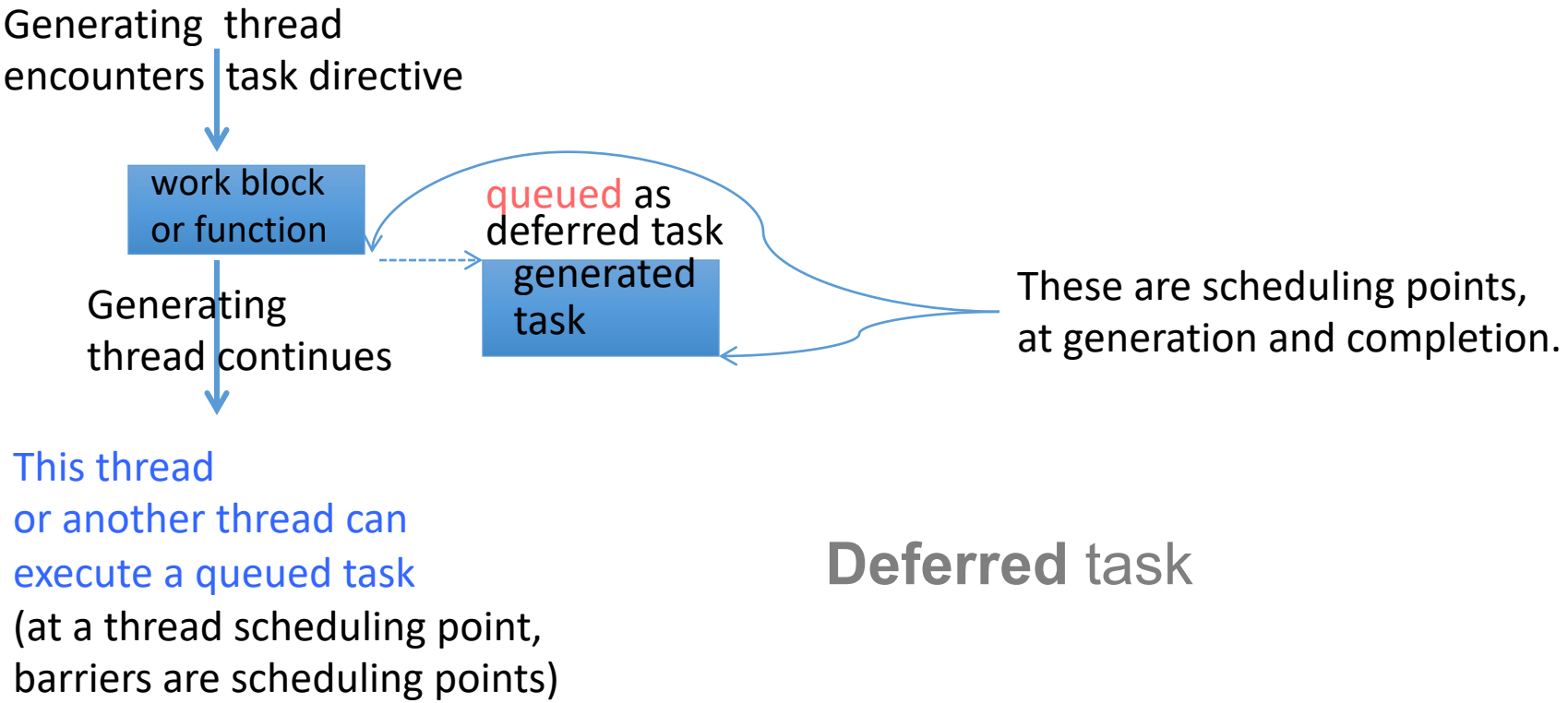
!! Master Thread

F90

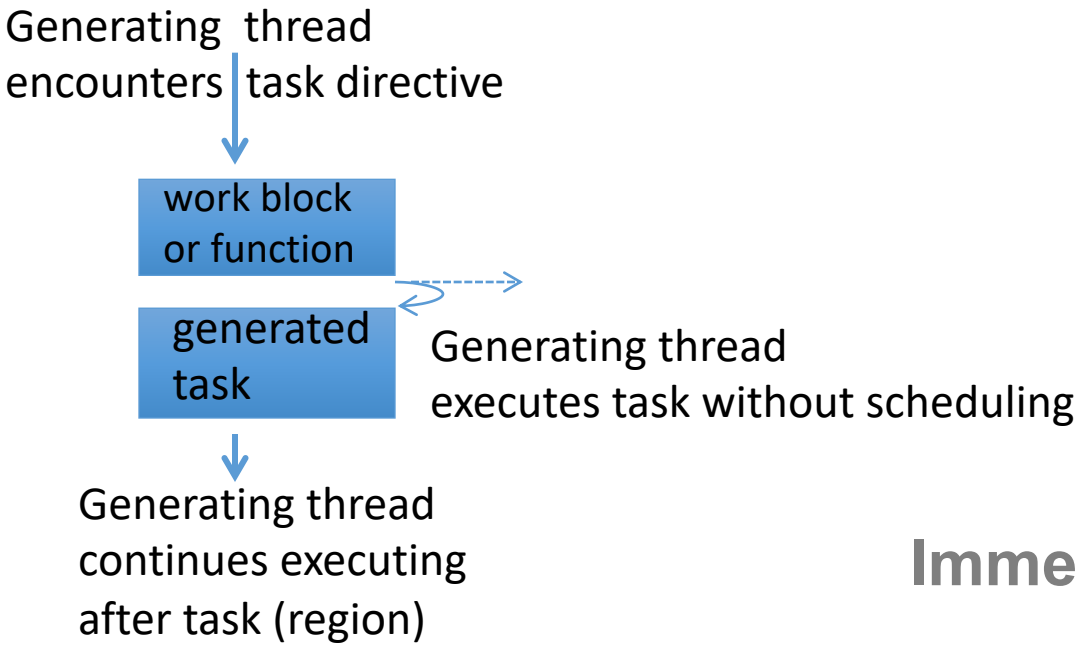
```
!$omp task
   foo(j)
!$omp end task

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

Deferred Task (usual case)



Immediate Task



Immediate task

Parallel Tasks

- Just as with worksharing (WS) chunks (of loops), the work of tasks (usually generated by a loop) must be free of race conditions (dependences) with other task executions.
- Critical and atomic directives provide mutually excusive operations on variables for avoiding race conditions within WS loops and Tasks.
- Unlike WS chunks, some types of tasks have dependences which involve large data sets (usually arrays) and cannot be efficiently handled by microscale atomics or critical regions. In these cases, dependence ordering is prescribed by depend clauses-- Course Grained Ordering.

Serial Tasks:

add1 tasks are sequential (-- in a prescribed order)

C/C++

```
void add1(int *a) { *a=*a+1;}

int main(){
  int a = 1;
  add1(&a); printf("a=%d\n",a);
  add1(&a); printf("a=%d\n",a);
}
```

F90

```
subroutine add1(a)
   integer :: a
   a=a+1
end subroutine

program main
   integer :: a=1;
   call add1(a); print*,a
   call add1(a); print*,a
end program
```

OUTPUT:

```
a=2
a=3
```

Parallel Tasks:

Run parallel tasks– is this correct? … NO

C/C++

```
void add1(int *a) { *a=*a+1;}
int main(){
int a = 1;
#pragma omp parallel num_threads(2)
 {
  if(omp_get_thread_num()==0) \
     add1(&a);
  if(omp_get_thread_num()==1)\
     add1(&a);
 }
 printf("a=%d\n",a);
}
```

F90

```
subroutine add1(a)
   integer :: a; a=a+1
end subroutine

program main
 use omp_lib; integer :: a=1;
 !$omp parallel num_threads(2)
   if(omp_get_thread_num()==0) &
      call add1(a)
   if(omp_get_thread_num()==1) &
      call add1(a)
 !$omp end parallel
 print*, a
end program
```

OUTPUT: a=?2|3

Race Condition

C/C++

```
void add1(int *a) { //slow down race, to expose problem
    if(omp_get_thread_num() == 1) sleep(1);
    int a_captured = *a;
    if(omp_get_thread_num() == 0) sleep(2);
    a_captured++;
    *a = a_captured;
}

int main(){ // Expose effect of race condition
   int a = 1;
    #pragma omp parallel num_threads(2)
    {
      if(omp_get_thread_num() == 0) add1(&a);
      if(omp_get_thread_num() == 1) add1(&a);
    }
    printf(" *** Final value of a = %d\n",a);
}
```

Most likely: a=2

Race Condition

F90

```
subroutine add1(a)
   use omp_lib; integer :: a, a_captured
    if(omp_get_thread_num() == 1) call sleep(1)
    a_captured = a
    if(omp_get_thread_num() == 0) call sleep(2)
    a_captured=a_captured+1
    a = a_captured;
end subroutine

program main
 use omp_lib; integer :: a=1
 !$omp parallel num_threads(2)
   if(omp_get_thread_num()==0) call add1(a)
   if(omp_get_thread_num()==1) call add1(a)
 !$omp end parallel
 print*, a
end program
```

Most likely: a=2

Learning Objective

-- Tasking --

Review Worksharing and Limitations

Basic Task Operations and Syntax

Task Synchronization

Running Tasks in Parallel

Data-sharing and firstprivate Default for Tasks

Common Use Cases for Tasks

Task Dependences

Taskloop

Synchronizing with child 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 with descendant tasks

Use taskgroup construct to wait for all children and their 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 Operations and Syntax

Task Synchronization

Running Tasks in Parallel

Data-sharing and firstprivate for Tasks

Common Use Cases for Tasks

Task Dependences

Taskloop

Generating Parallel Tasks—in a parallel region

First, create a team of threads, to work on tasks.

Use a one thread (via master/single) to generate tasks.

C/C++

```
#pragma omp parallel num_threads(4)
{
  #pragma omp master
  {
    //generate multiple tasks

  }
}
```

```
while(…){
  #pragma omp task
  {…}
}
```

F90

```
!$omp parallel num_threads(4)

  !$omp master

    !generate multiple tasks

  !omp end master
!$omp end parallel
```

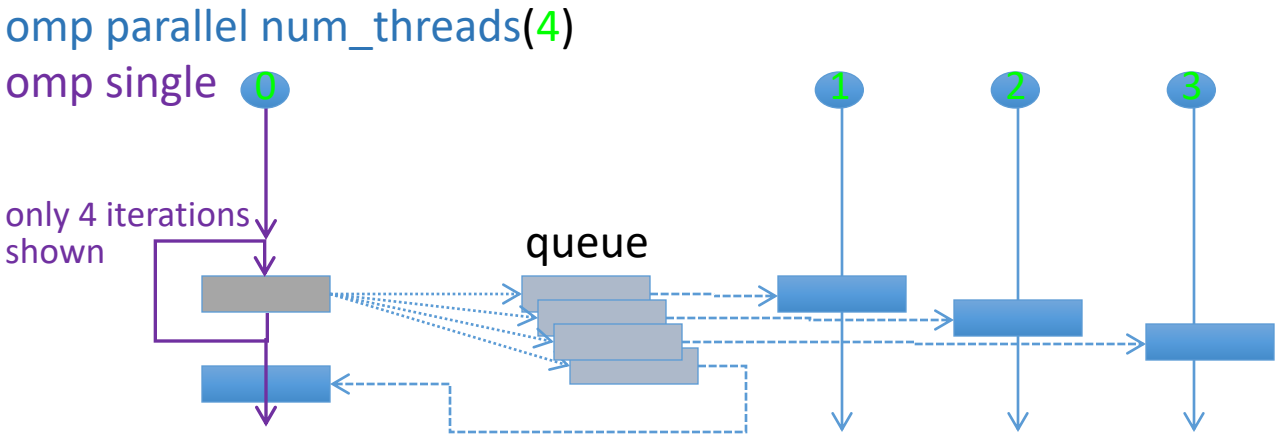
Generate multiple tasks: with loop (while/do/for), or recursively

Tasks in parallel region

Threads of Parallel Team will dequeue & execute tasks

Shared variables of parallel region are also shared by tasks

Tasks are often synchronized by taskwait & taskgroup (although, all tasks must obey barriers)



Tasking

Review Worksharing and Limitations

Basic Task Operations and Syntax

Task Synchronization

Running Tasks in Parallel

Data-sharing and implicit firstprivate for Tasks

Common Use Cases for Tasks

Task Dependences

Taskloop

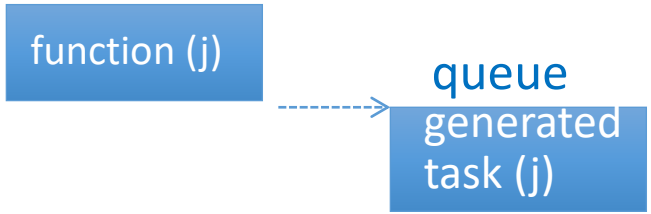
Tasks: Data Environment

- 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 of the enclosing parallel construct become firstprivate for the task region.

The “work index” passed to a task needs to be firstprivate. Why?

Deferred Task

Generating thread encounters task construct



Basic Concept:

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

#omp parallel master
loop

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

j++;
```

!$omp parallel master
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

The priority clause* is a user-defined way to solve imbalance. (Larger # = higher priority.) General Rule: "largest" tasks first.

```
for (int i=N-1;i=<0; i++) {
     n_units=amt_of_work(A[i])
     #pragma omp task priority(n_units)
      work(i, A, N);
}
```

```
do i=N,1
    n_unit=amt_of_work(A(i))
    !$omp task firstprivate(i) priority(n_units)
       call work(i, A, N)
    !$omp end task
enddo
```

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.

* implementations are not obliged to adhere to priority

Tasking

Review Worksharing and Limitations

Basic Task Operations and Syntax

Task Synchronization

Running Tasks in Parallel

Data-sharing and firstprivate for Tasks

Common Use Cases for Tasks

Task Dependences

Taskloop

What is Tasking for?

Irregular Computing of independent work blocks:

- 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 tasks (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 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
```

Immediate Task Execution 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.

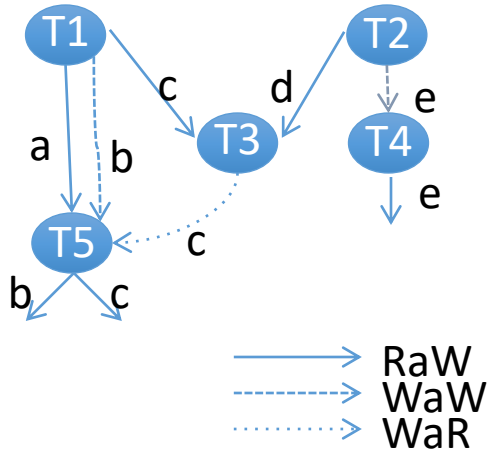
If the ***if*** argument is false, task is undeferred (exec time (usec) is less than 0.01).

- 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 ...
      f1(&a, &b, &c);
T2  #pragma omp task ...
      f2(&d, &e);
T3  #pragma omp task ...
      f3(c,d);
T4  #pragma omp task ...
      f4(&e);
T5  #pragma omp task ...}
      f5(a,&b,&c);
```



dependence

out == W
in == R

Summary

- Tasks are used mainly in irregular computing (async, too).
- Tasks are often generated by a single thread, or
- Task generation can be recursive.
- Depend clause can prescribe dependence.
- Priority provides hint for execution order.
- Firstprivate becomes the data-sharing attribute for private variables, shared variables remain shared
- Untied generator task can assure generation progress.

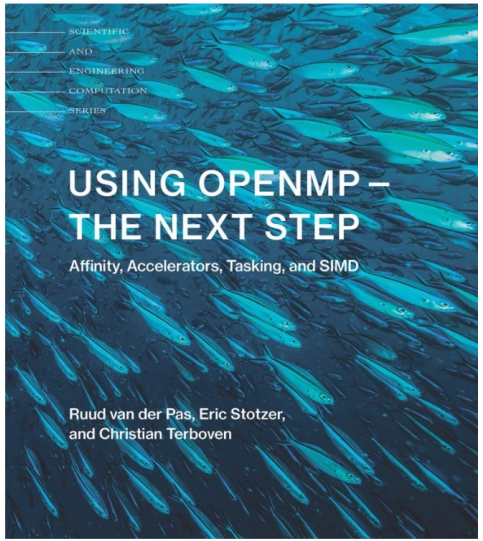
Not discussed here.

--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 Operations and Syntax

Task Synchronization

Running Tasks in Parallel

Data-sharing and firstprivate for Tasks

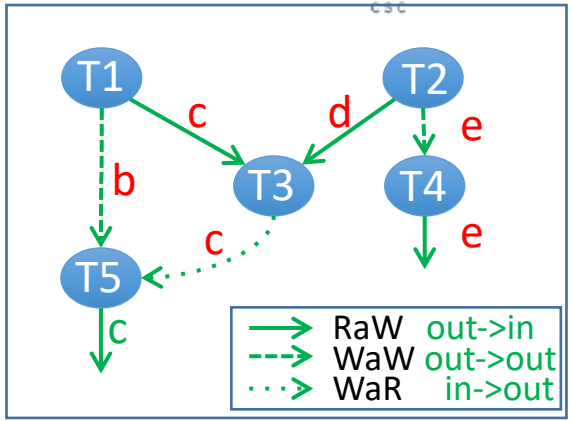
Common Use Cases for Tasks

Task Dependences

Taskloop

Depend clause

depend(*dependence-type* **:** *list* **)**

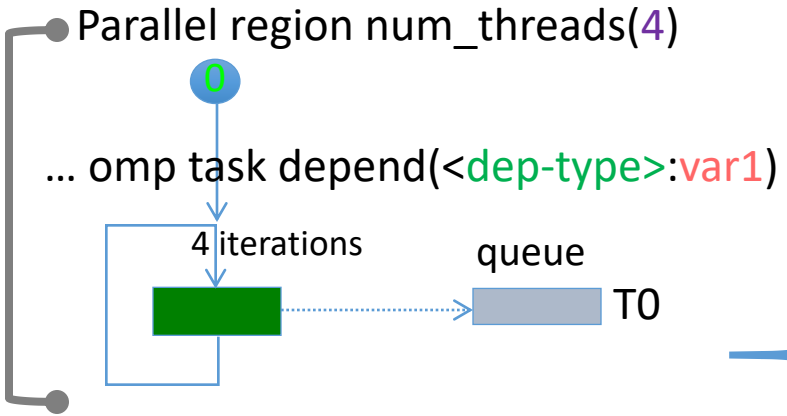


dependence-type: == what the task needs to do with list *items*

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 Rule: wait for RaW, WaW or WaR to finish.

Dependences

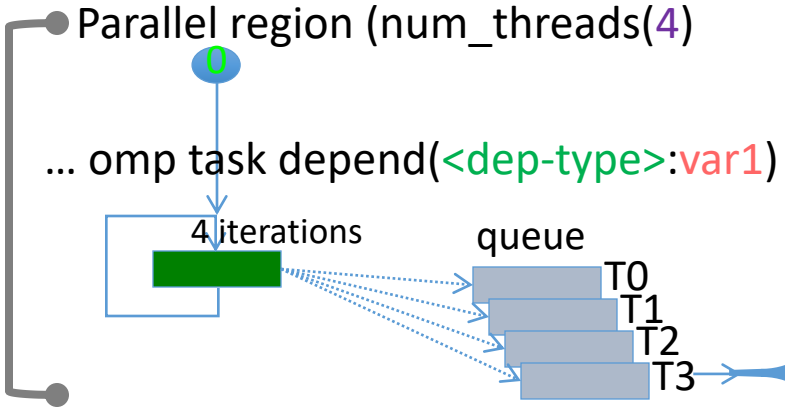


Task 1 execution dependence

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 and uncompleted 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.

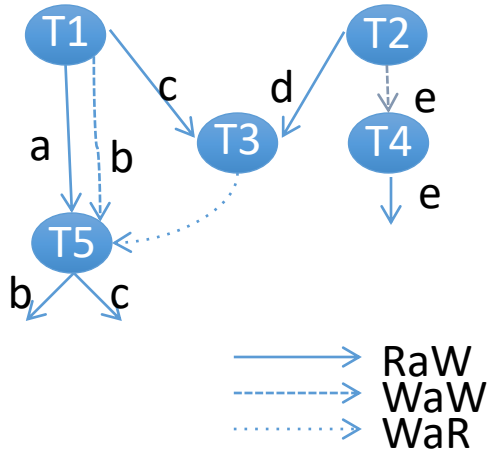
T2 is put on queue, sees previously queued task with x identifier→ has NO ordering (because it is RAR)

Print value is 1 or 2.

Task depend clause

Following a graph

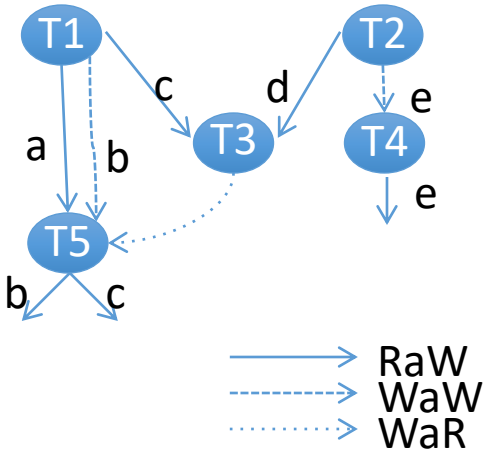
```
  #pragma omp parallel
  #pragma omp single
  {
T1  #pragma omp task depend(out:a,b,c)
      f1(&a, &b, &c);
T2  #pragma omp task depend(out:d,e)
      f2(&d, &e);
T3  #pragma omp task depend(in:c,d)
      f3(c,d);
T4  #pragma omp task depend(out,e)
      f4(&e);
T5  #pragma omp task depend(in:a)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

Task Synchronization

Running Tasks in Parallel

Data-sharing and firstprivate for Tasks

Common Use Cases for Tasks

Task Dependences

Taskloop

taskloop

Allows iterations of loops can be executed as tasks (of a taskgroup)

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

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

- important clauses:

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

Single generator needed
All team members are not necessary
Implied taskgroup for synchronization

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

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); }

     #pragma omp task
     other_work(); // can execute concurrently

   }  // 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.
- Firstprivate becomes the data-sharing attribute for private variables, shared variables remain shared.
- Untied generator task can ensure generation progress.

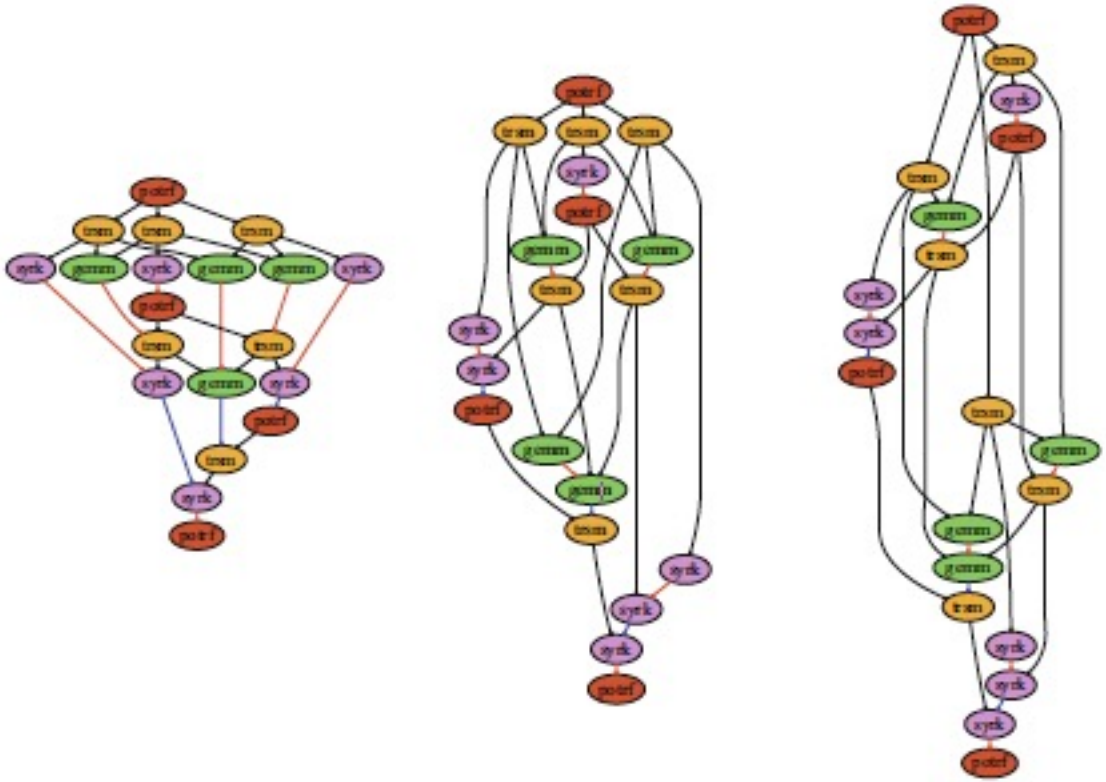


Fig. 4. DAGs for 3 variations of tiled Cholesky decomposition (from left to right): right-looking, left-looking, and top-looking. These show how the order in which tasks are presented to the scheduler affect the available parallelization (green=GEMM, red=POTRF, orange=TRSM, and purple=SYRK). (Color figure online)

Jack dongarra, et al,
Task-Based Cholesky Decomposition on Knights Corner Using OpenMP
DOI:
10.1007/978-3-319-46079-6_37

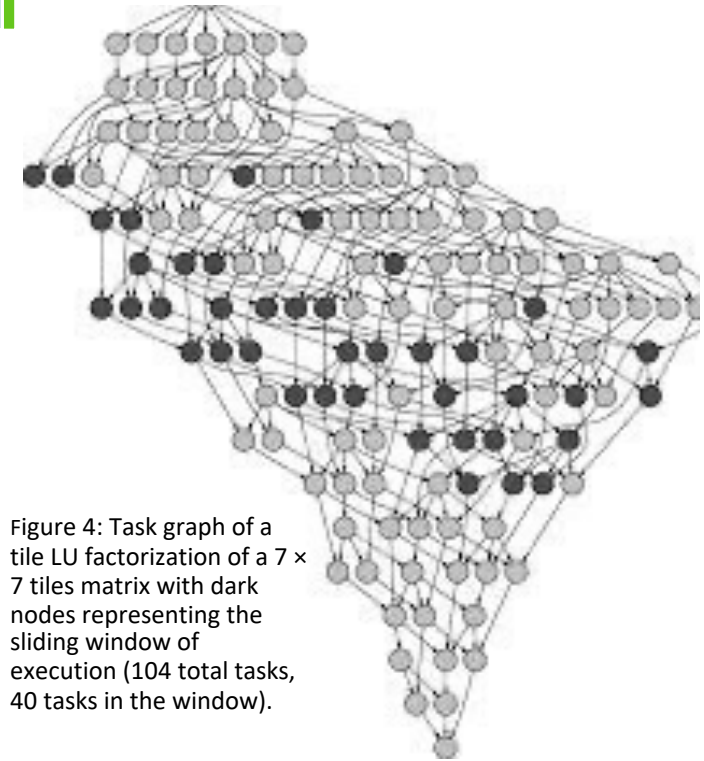
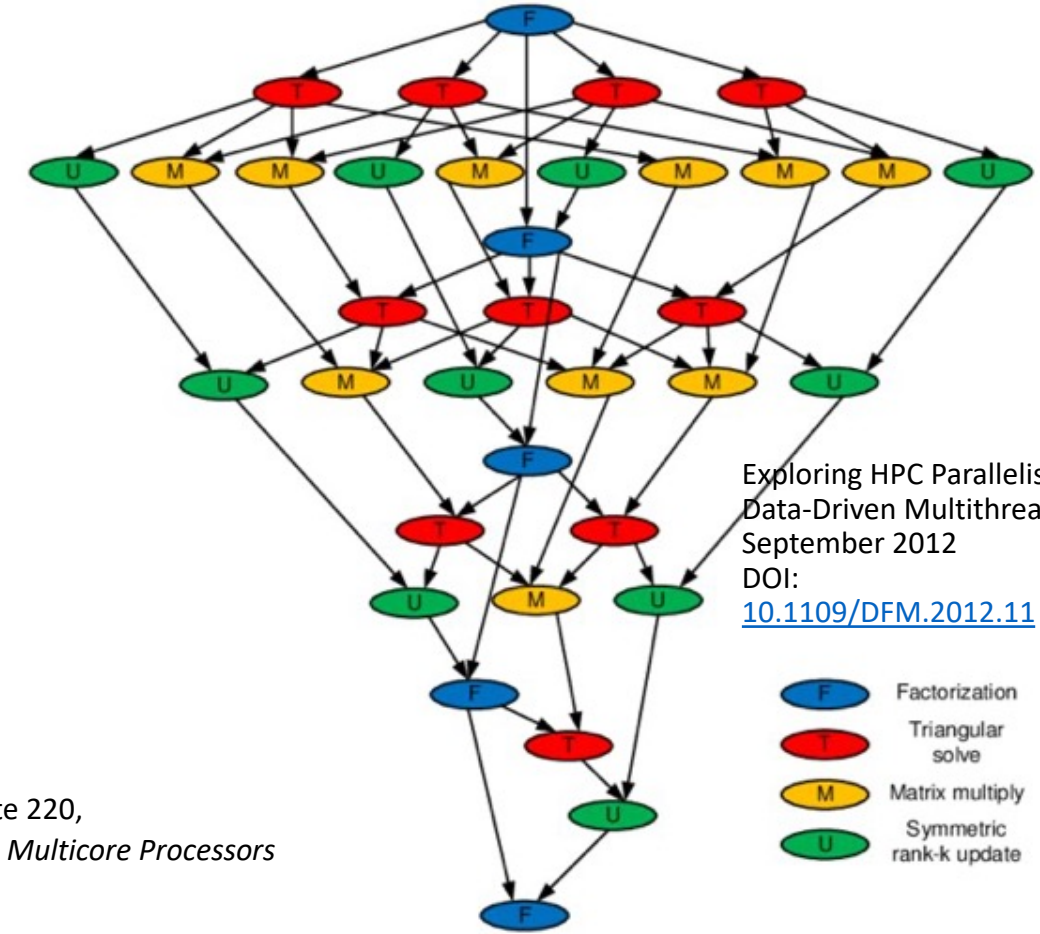


Figure 4: Task graph of a tile LU factorization of a 7 × 7 tiles matrix with dark nodes representing the sliding window of execution (104 total tasks, 40 tasks in the window).

Jakub Kurzak and Jack Dongarra, LAPACK Working Note 220,
Fully Dynamic Scheduler for Numerical Computing on Multicore Processors



Exploring HPC Parallelism with Data-Driven Multithreading
September 2012
DOI:
10.1109/DFM.2012.11

https://pm.bsc.es/ftp/ompss/doc/examples/01-examples/cholesky/README.html