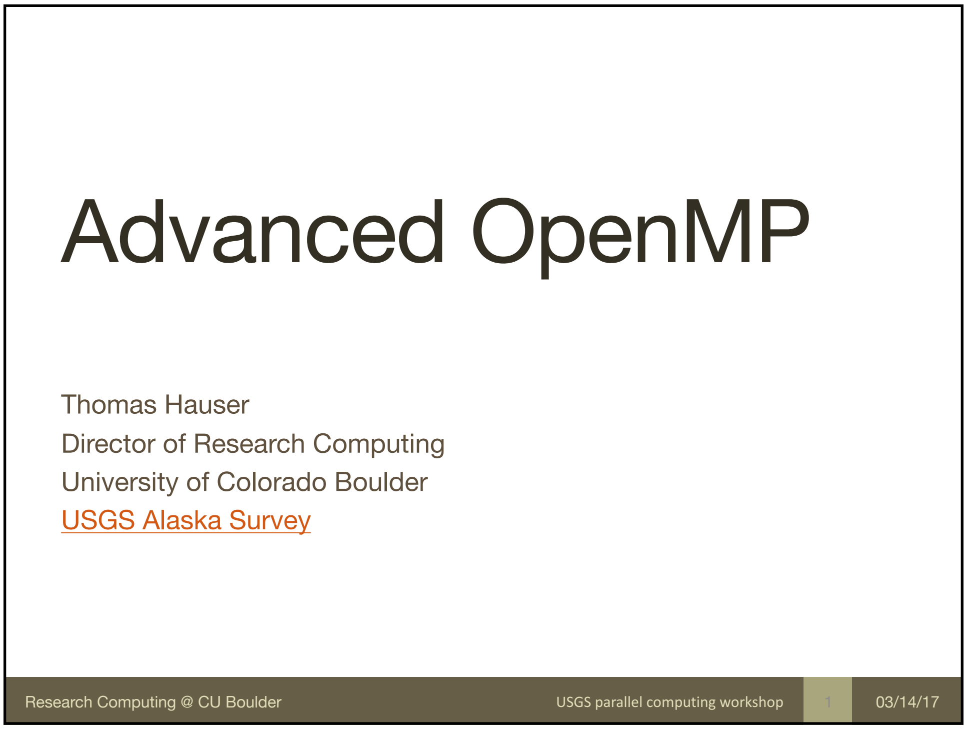
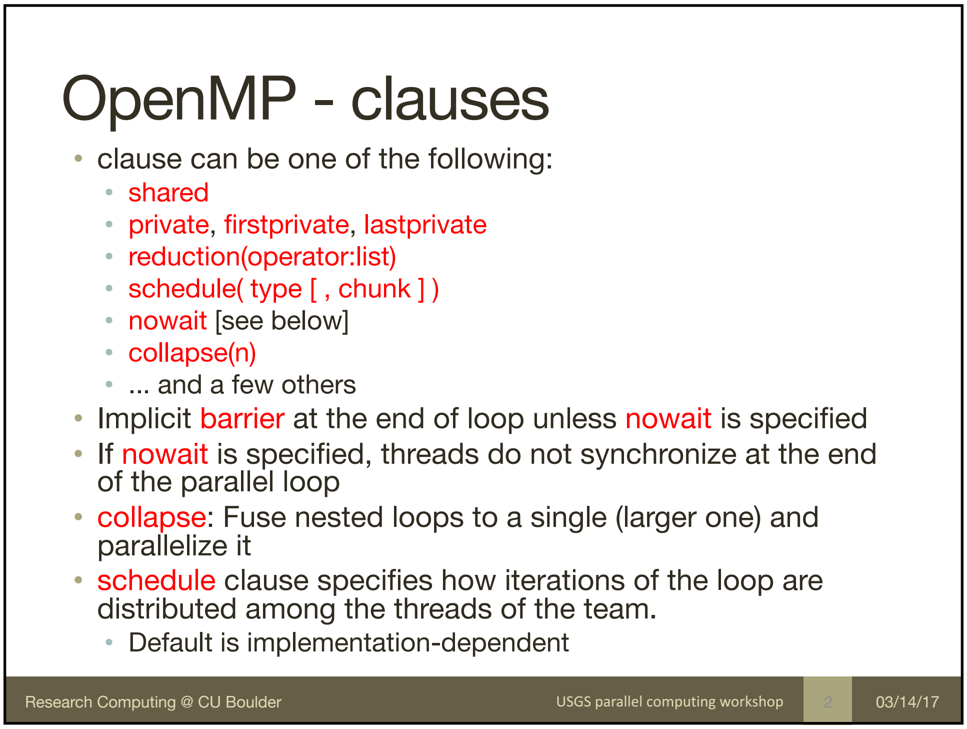


Advanced OpenMP

Thomas Hauser
Director of Research Computing
University of Colorado Boulder
USGS Alaska Survey



OpenMP - clauses

- clause can be one of the following:
  - shared
  - private, firstprivate, lastprivate
  - reduction(operator:list)
  - schedule(type [, chunk])
  - nowait [see below]
  - collapse(n)
  - ... and a few others
- Implicit barrier at the end of loop unless nowait is specified
- If nowait is specified, threads do not synchronize at the end of the parallel loop
- collapse: Fuse nested loops to a single (larger one) and parallelize it
- schedule clause specifies how iterations of the loop are distributed among the threads of the team.
  - Default is implementation-dependent

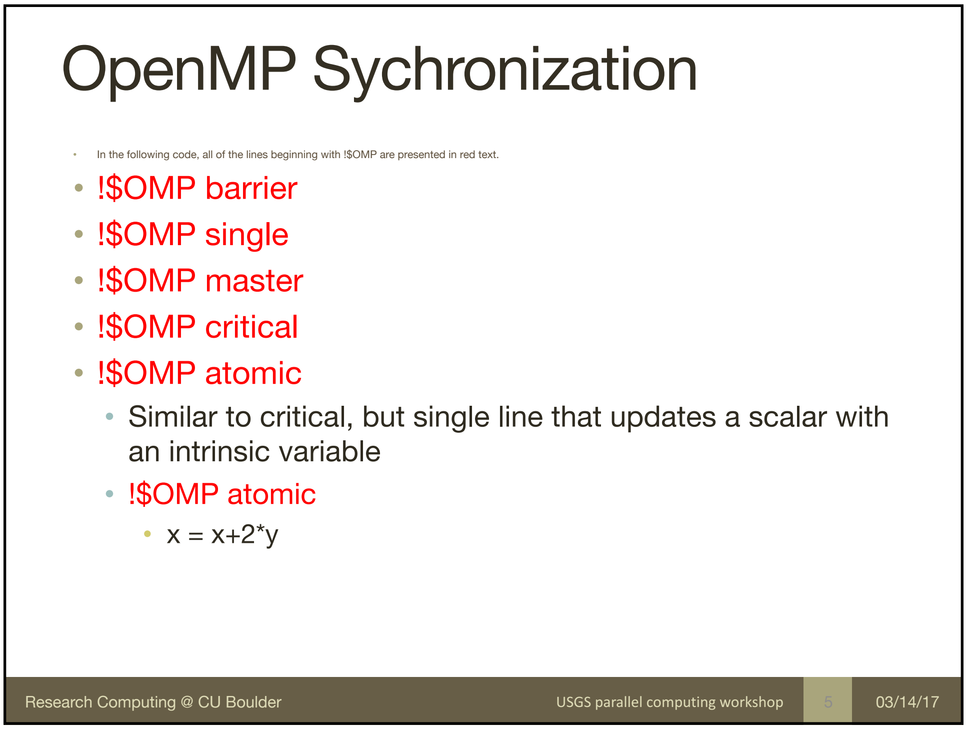
nowait Clause

- Compiler puts a barrier synchronization at end of every parallel do statement
- In our example, this is necessary: if a thread leaves loop and changes **low** or **high**, it may affect behavior of another thread
- If we make these private variables, then it would be okay to let threads move ahead, which could reduce execution time

Use of nowait Clause

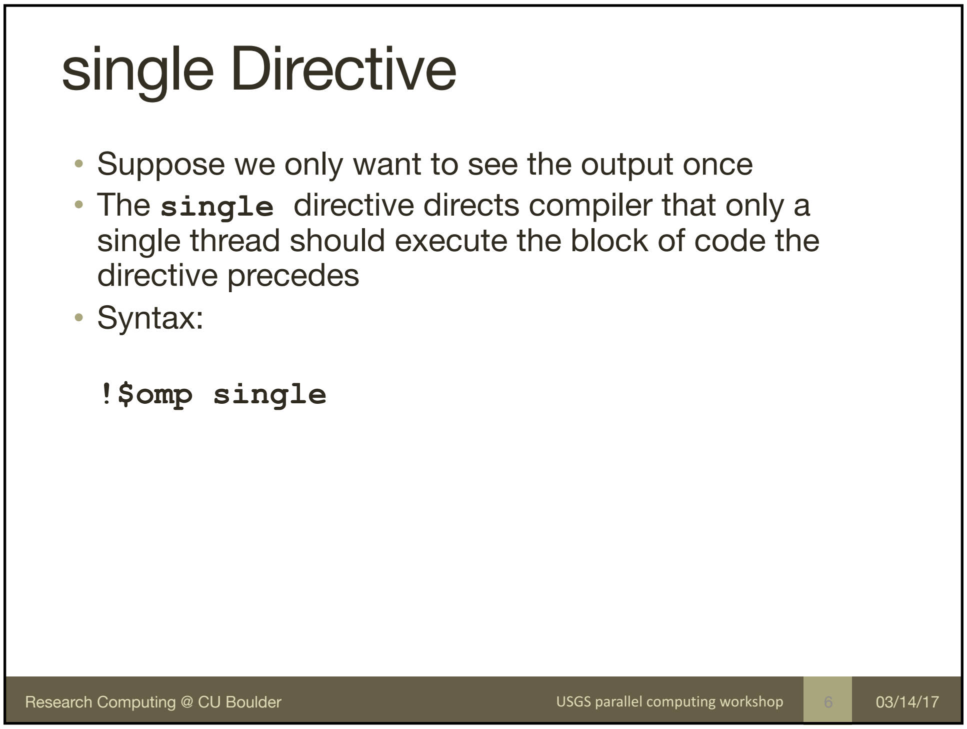
In the following code, the lines beginning with $!OMP are presented in red text.

```
!$omp parallel private(i,j)
Do i = 1, m
   low = a(i)
   high = b(i)
   if (low > high) then
      !$OMP single
      write(*,*) "Exiting”, I
      !$OMP single
      exit
   endif
End do
!$omp do
do j = low, high
  c(j) = (c(j) - a(i))/b(i)
End do nowait
!$omp end do
!$omp end parallel
```



OpenMP Sychronization

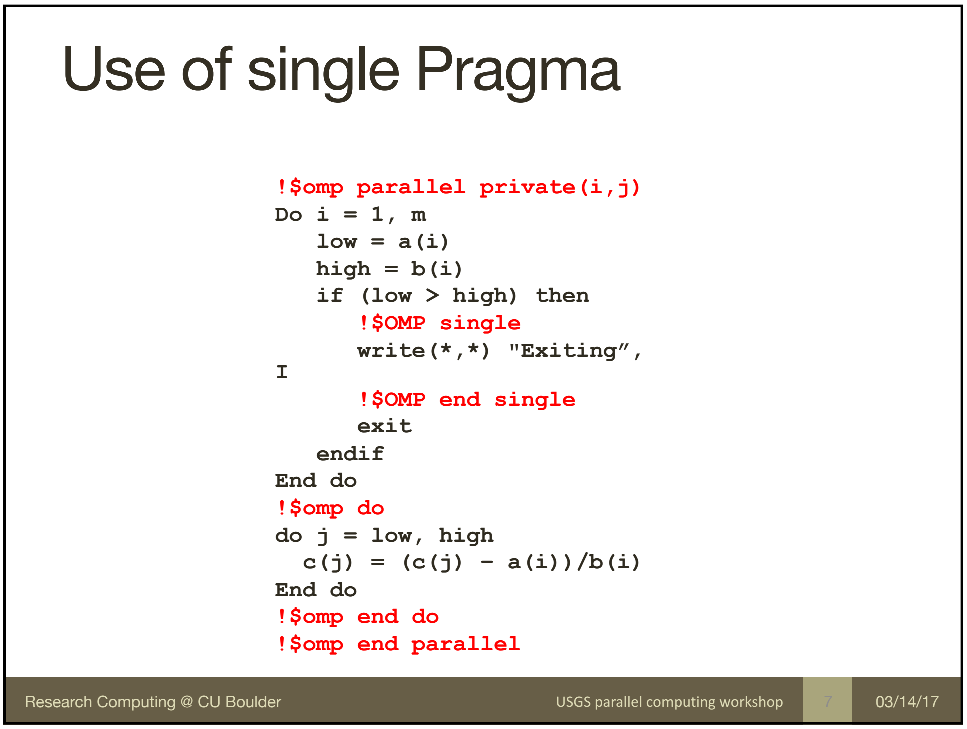
- In the following code, all of the lines beginning with !$OMP are presented in red text.
- !$OMP barrier
- !$OMP single
- !$OMP master
- !$OMP critical
- !$OMP atomic
  - Similar to critical, but single line that updates a scalar with an intrinsic variable
  - !$OMP atomic
    - x = x+2*y



single Directive

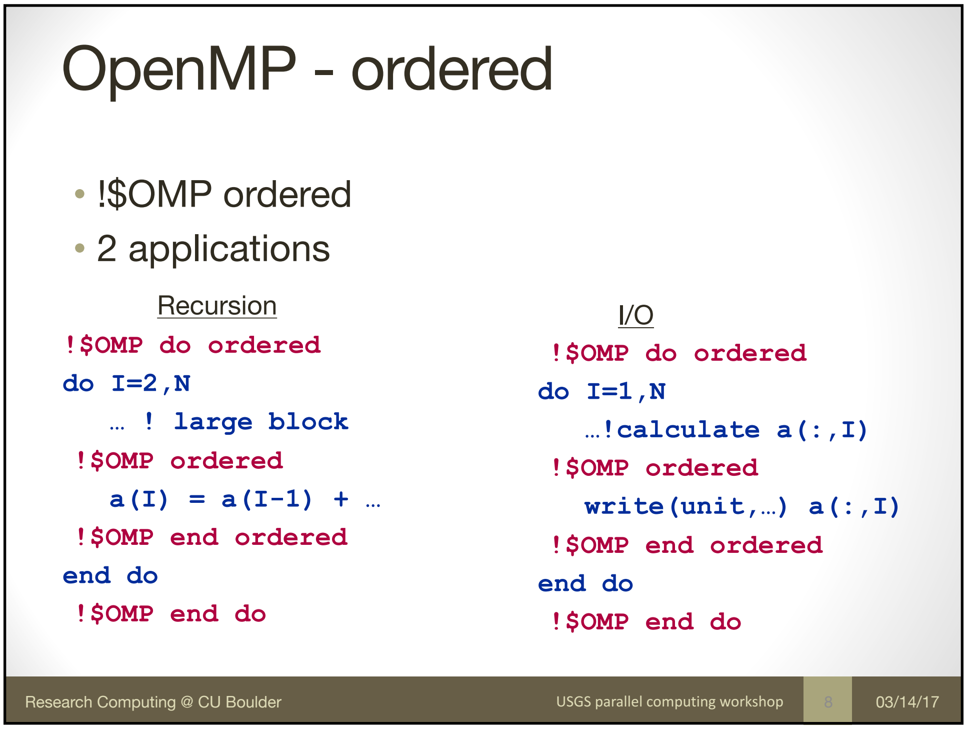
- Suppose we only want to see the output once
- The **`single`** directive directs compiler that only a single thread should execute the block of code the directive precedes
- Syntax:

```
!$omp single
```



Use of single Pragma

```
!$omp parallel private(i,j)
Do i = 1, m
   low = a(i)
   high = b(i)
   if (low > high) then
      !$OMP single
      write(*,*) "Exiting”,
I
      !$OMP end single
      exit
   endif
End do
!$omp do
do j = low, high
  c(j) = (c(j) – a(i))/b(i)
End do
!$omp end do
!$omp end parallel
```



OpenMP - ordered

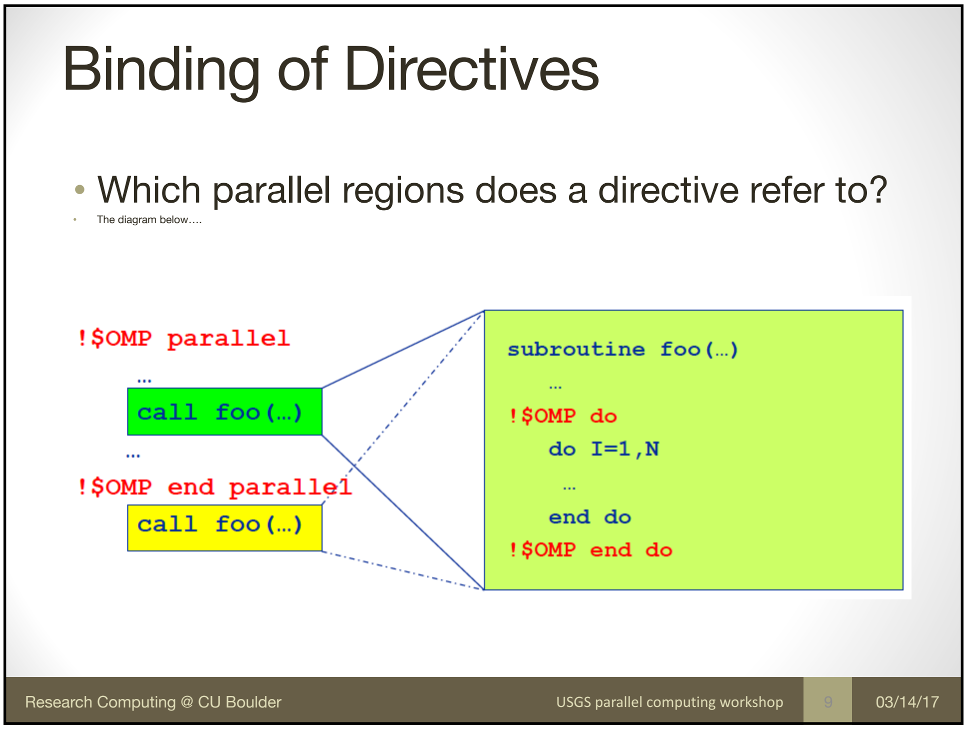
- !$OMP ordered
- 2 applications

Recursion

```
!$OMP do ordered
do I=2,N
   … ! large block
 !$OMP ordered
   a(I) = a(I-1) + …
 !$OMP end ordered
end do
 !$OMP end do
```

I/O

```
 !$OMP do ordered
do I=1,N
   …!calculate a(:,I)
 !$OMP ordered
   write(unit,…) a(:,I)
 !$OMP end ordered
end do
 !$OMP end do
```

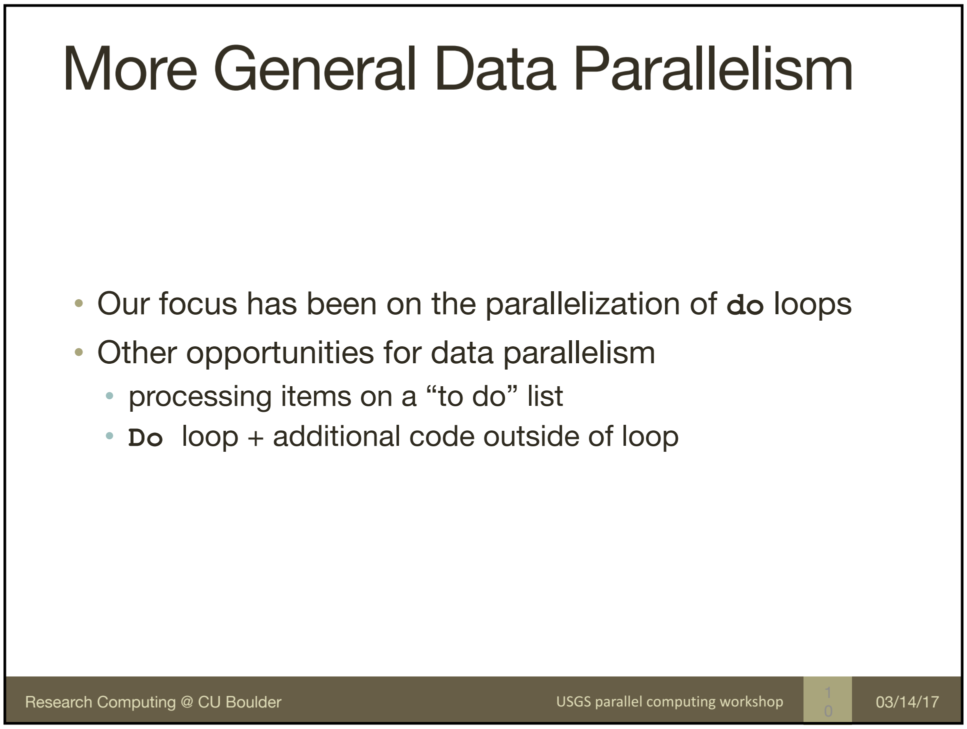


Binding of Directives

- Which parallel regions does a directive refer to?
  - The diagram below….

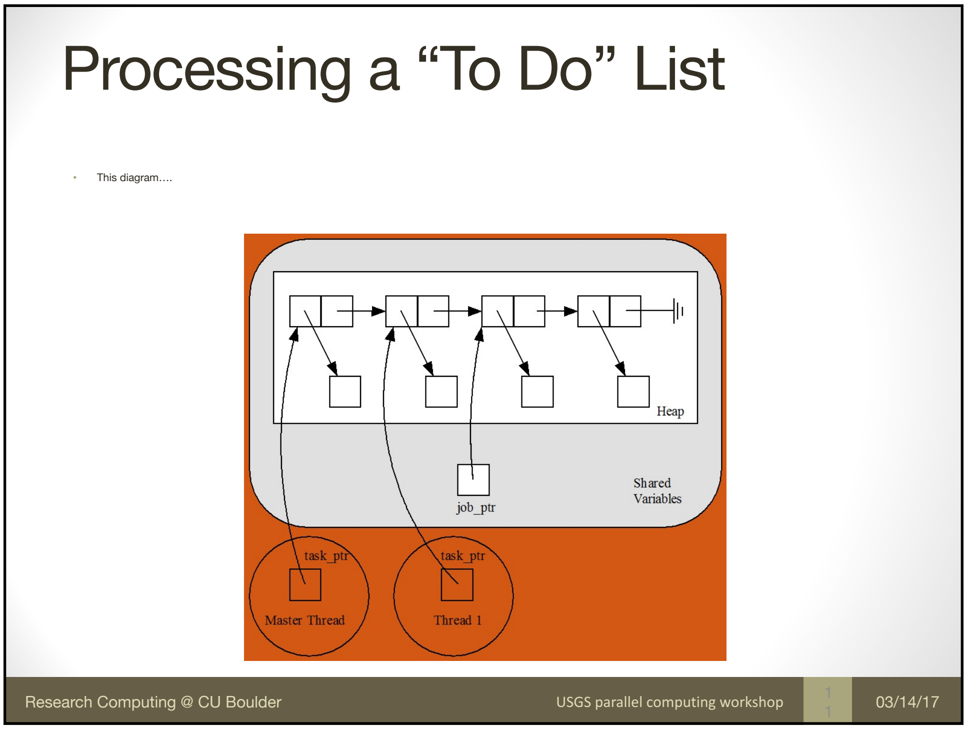
```
!$OMP parallel
    …
    call foo(…)
   …
!$OMP end parallel
    call foo(…)
```

```
subroutine foo(…)
   …
!$OMP do
   do I=1,N
    …
   end do
!$OMP end do
```



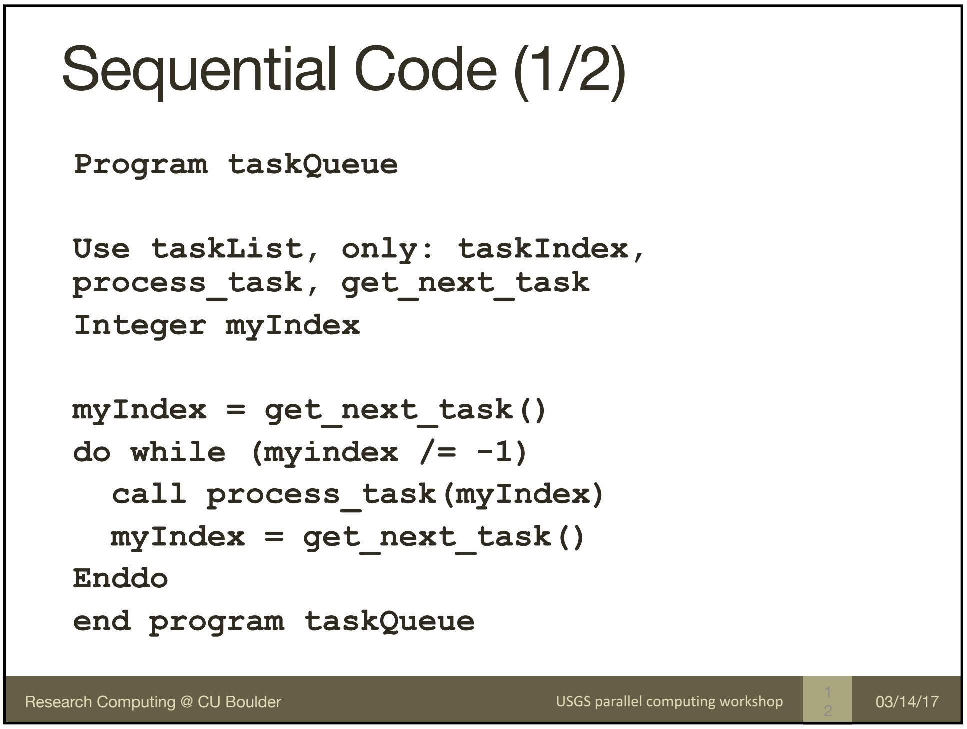
More General Data Parallelism

- Our focus has been on the parallelization of **`do`** loops
- Other opportunities for data parallelism
  - processing items on a “to do” list
  - **`Do`** loop + additional code outside of loop



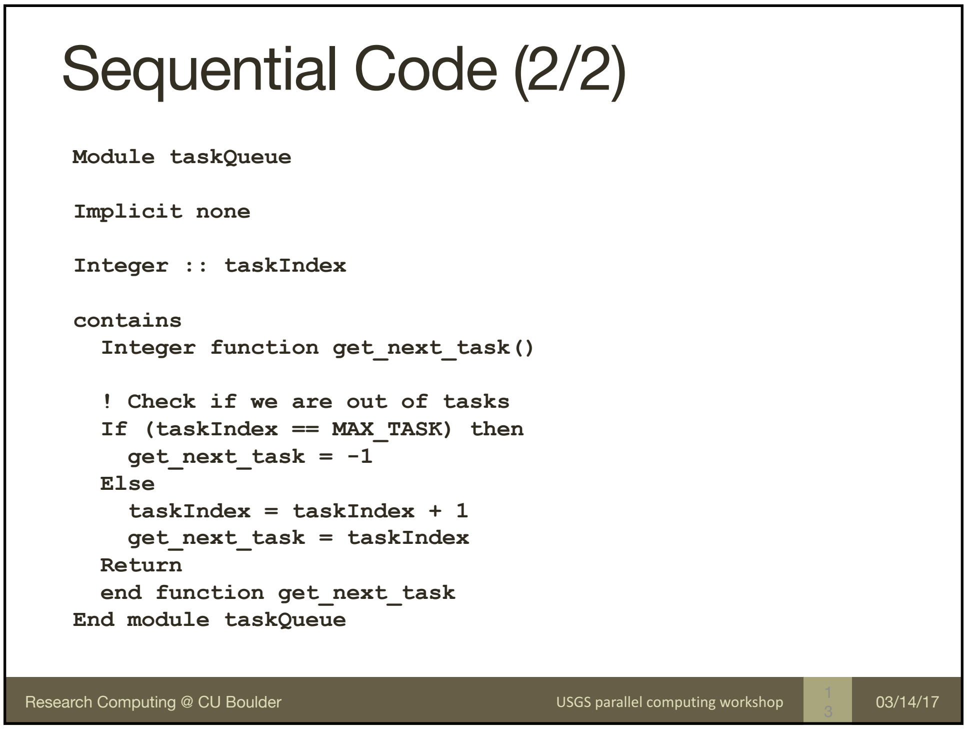
Processing a “To Do” List

- This diagram....



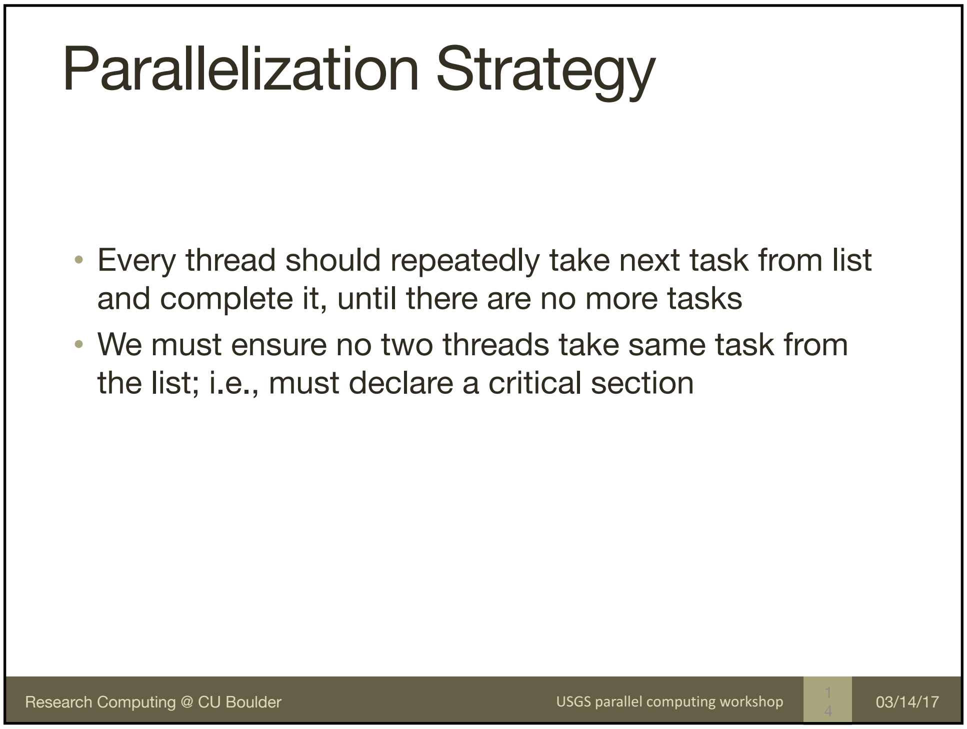
Sequential Code (1/2)

```
Program taskQueue

Use taskList, only: taskIndex,
process_task, get_next_task
Integer myIndex

myIndex = get_next_task()
do while (myindex /= -1)
  call process_task(myIndex)
  myIndex = get_next_task()
Enddo
end program taskQueue
```



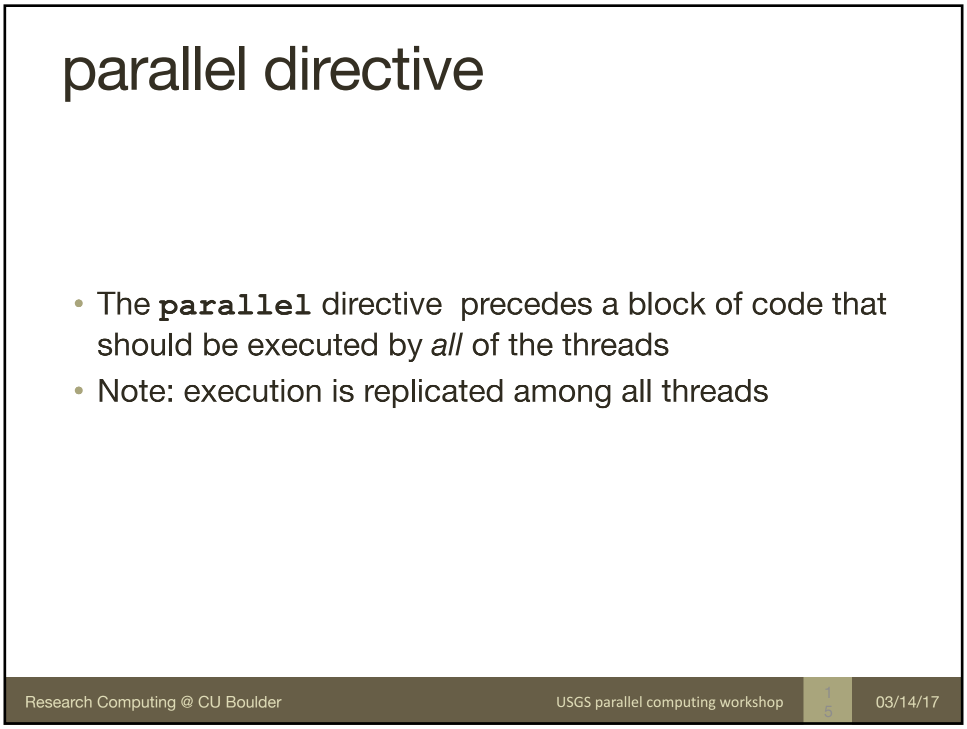
Sequential Code (2/2)

```
Module taskQueue

Implicit none

Integer :: taskIndex

contains
  Integer function get_next_task()

  ! Check if we are out of tasks
  If (taskIndex == MAX_TASK) then
    get_next_task = -1
  Else
    taskIndex = taskIndex + 1
    get_next_task = taskIndex
  Return
  end function get_next_task
End module taskQueue
```



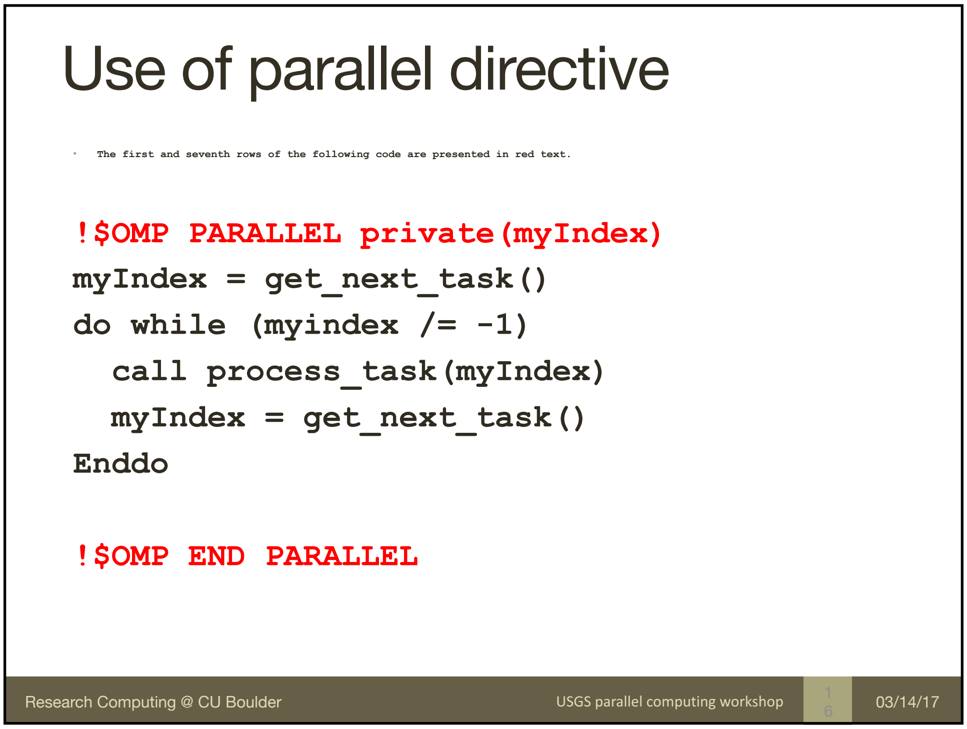
Parallelization Strategy

- Every thread should repeatedly take next task from list and complete it, until there are no more tasks
- We must ensure no two threads take same task from the list; i.e., must declare a critical section



parallel directive

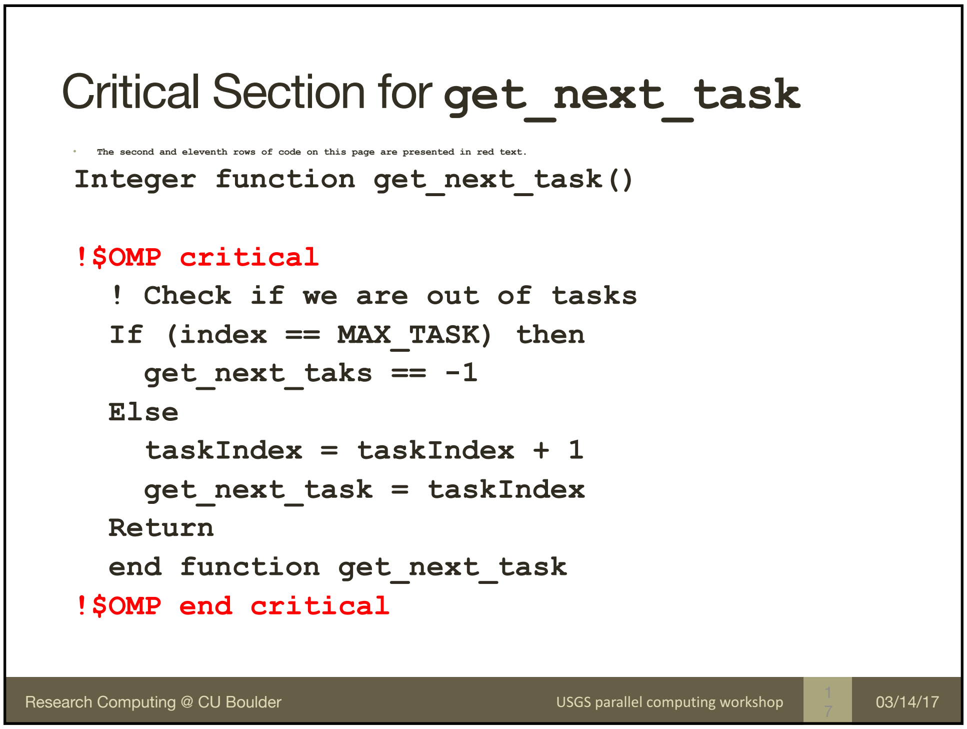
- The **`parallel`** directive precedes a block of code that should be executed by *all* of the threads
- Note: execution is replicated among all threads



Use of parallel directive

- The first and seventh rows of the following code are presented in red text.

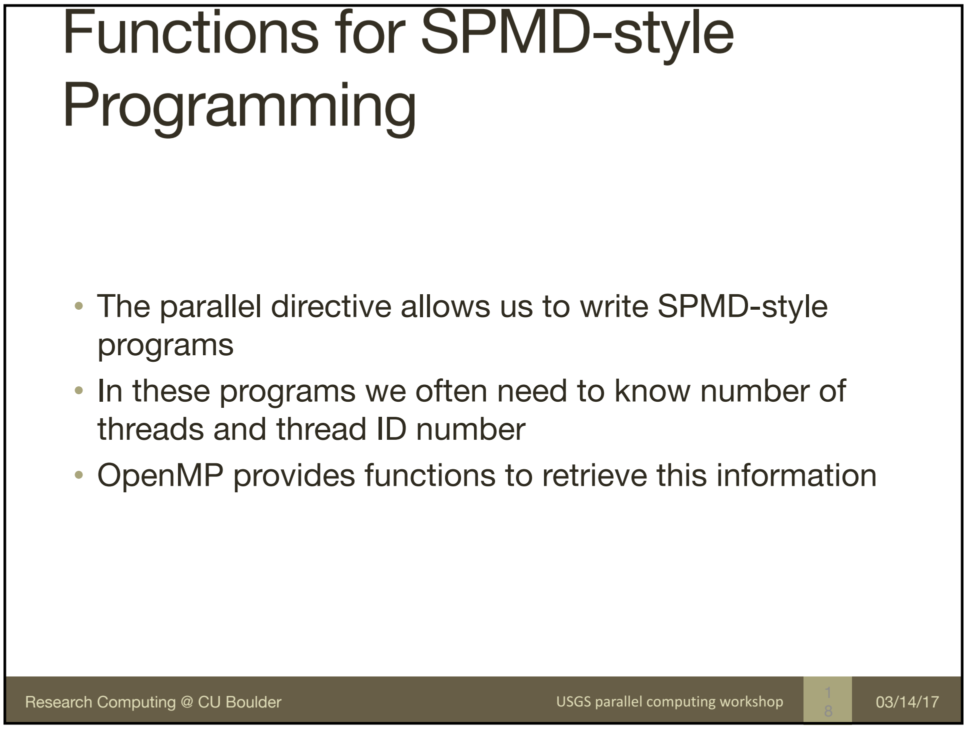
```
!$OMP PARALLEL private(myIndex)
myIndex = get_next_task()
do while (myindex /= -1)
  call process_task(myIndex)
  myIndex = get_next_task()
Enddo

!$OMP END PARALLEL
```



Critical Section for `get_next_task`

- The second and eleventh rows of code on this page are presented in red text.

```
Integer function get_next_task()

!$OMP critical
  ! Check if we are out of tasks
  If (index == MAX_TASK) then
    get_next_taks == -1
  Else
    taskIndex = taskIndex + 1
    get_next_task = taskIndex
  Return
  end function get_next_task
!$OMP end critical
```



Functions for SPMD-style Programming

- The parallel directive allows us to write SPMD-style programs
- In these programs we often need to know number of threads and thread ID number
- OpenMP provides functions to retrieve this information

Functional Parallelism

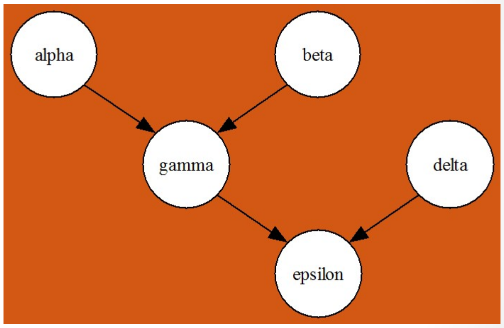
- To this point all of our focus has been on exploiting data parallelism
- OpenMP allows us to assign different threads to different portions of code (functional parallelism)

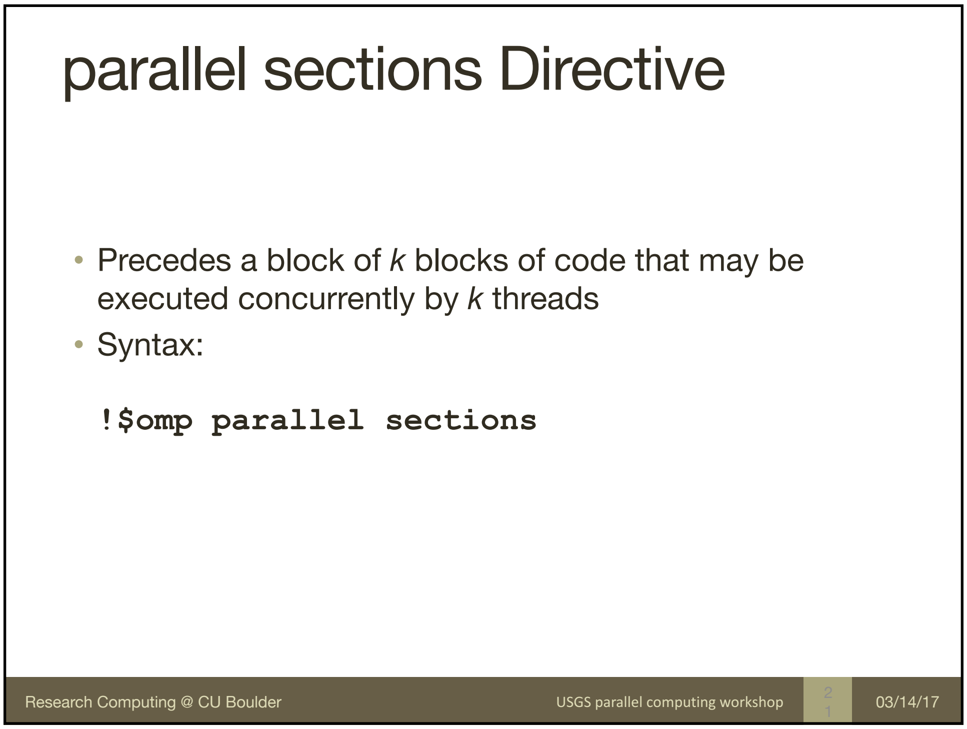
Functional Parallelism Example

```
v = alpha()
   w = beta()
   x = gamma(v, w)
   y = delta()
   write(*,*)
epsilon(x,y)
```

- May execute alpha, beta, and delta in parallel

The following flow diagram shows….

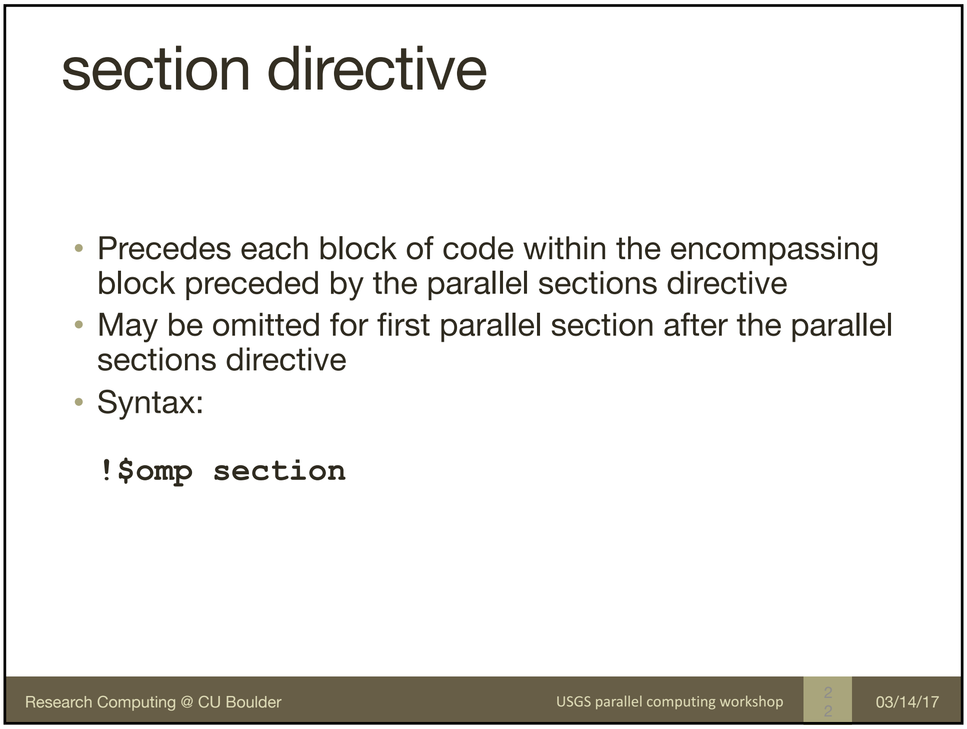




parallel sections Directive

- Precedes a block of *k* blocks of code that may be executed concurrently by *k* threads
- Syntax:

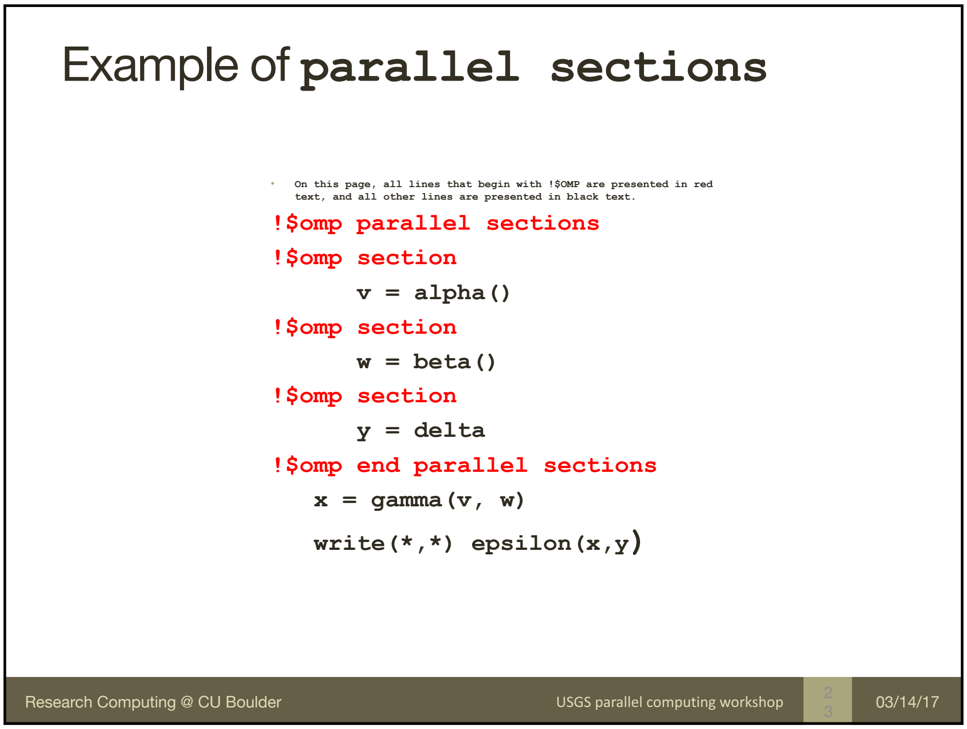
```
!$omp parallel sections
```



section directive

- Precedes each block of code within the encompassing block preceded by the parallel sections directive
- May be omitted for first parallel section after the parallel sections directive
- Syntax:

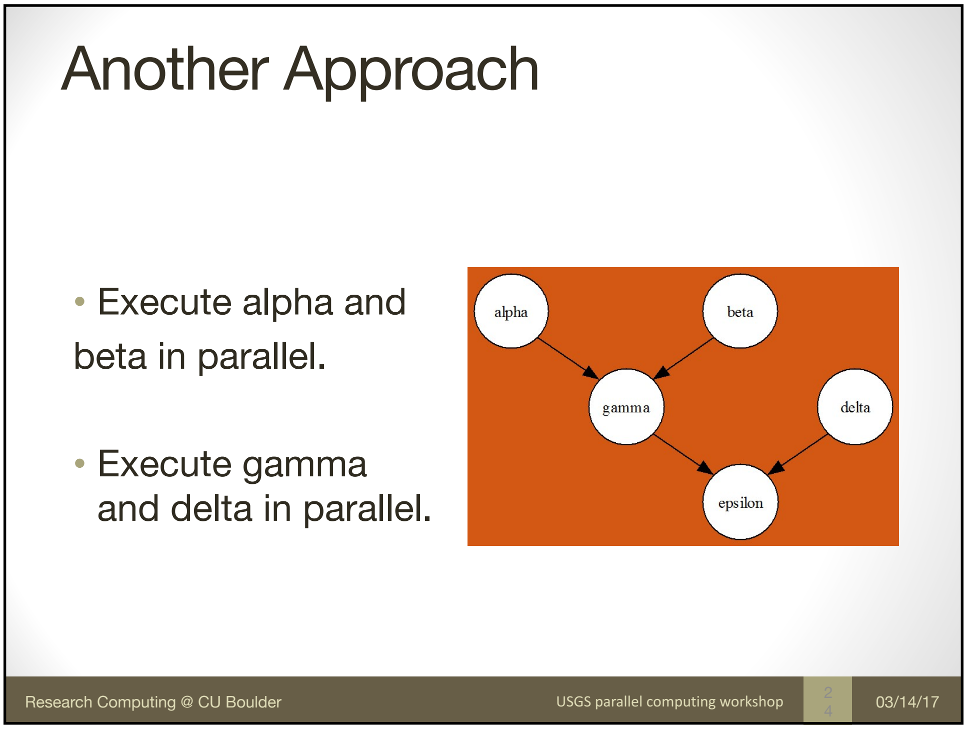
```
!$omp section
```



Example of **parallel sections**

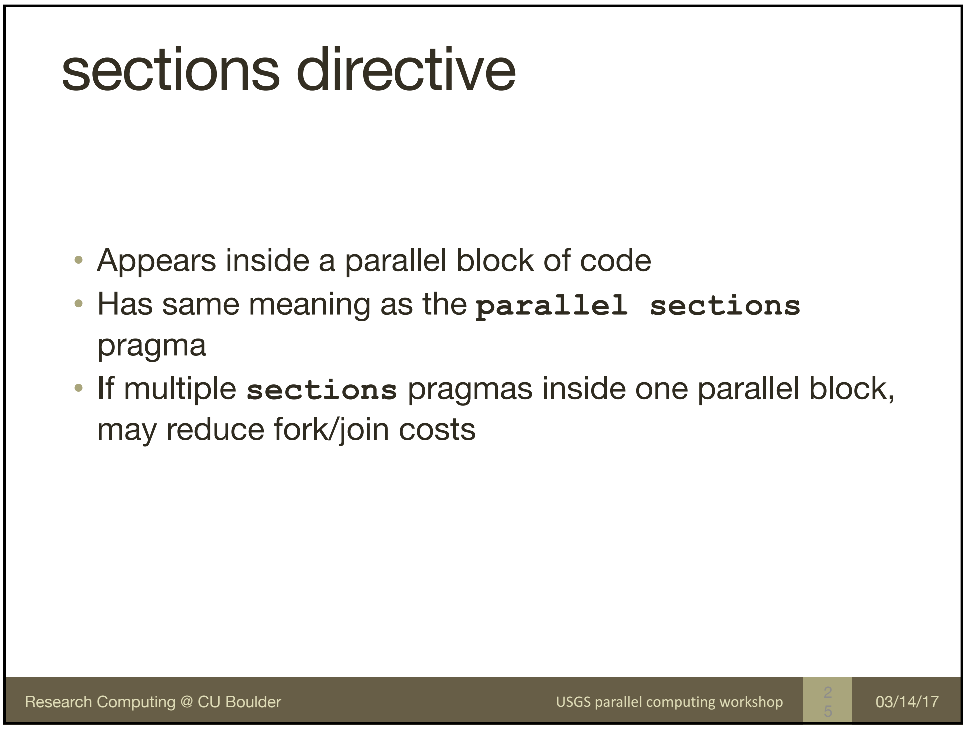
- On this page, all lines that begin with !$OMP are presented in red text, and all other lines are presented in black text.

```
!$omp parallel sections
!$omp section
      v = alpha()
!$omp section
      w = beta()
!$omp section
      y = delta
!$omp end parallel sections
   x = gamma(v, w)
   write(*,*) epsilon(x,y)
```



Another Approach

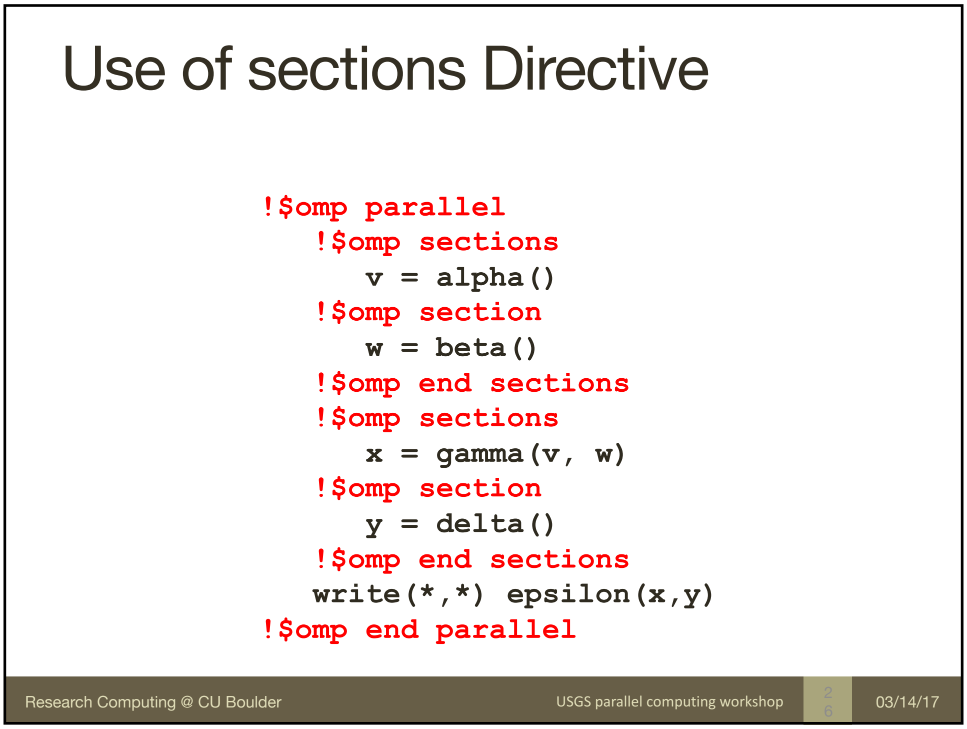
- Execute alpha and beta in parallel.
- Execute gamma and delta in parallel.



sections directive

- Appears inside a parallel block of code
- Has same meaning as the **`parallel sections`** pragma
- If multiple **`sections`** pragmas inside one parallel block, may reduce fork/join costs

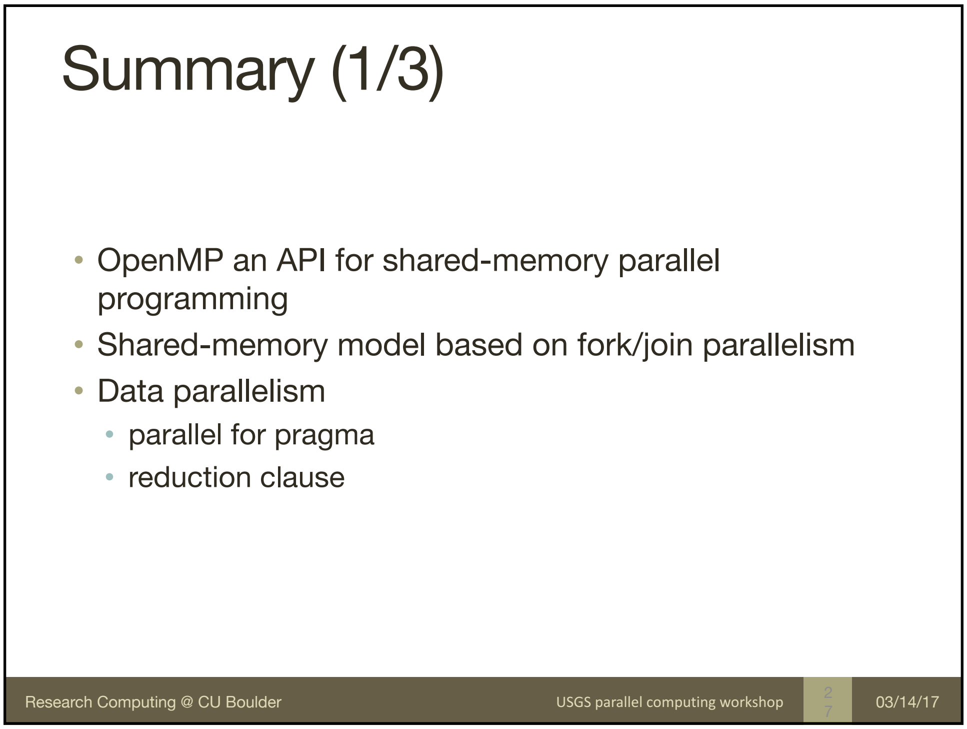
Research Computing @ CU Boulder USGS parallel computing workshop 25 03/14/17



Use of sections Directive

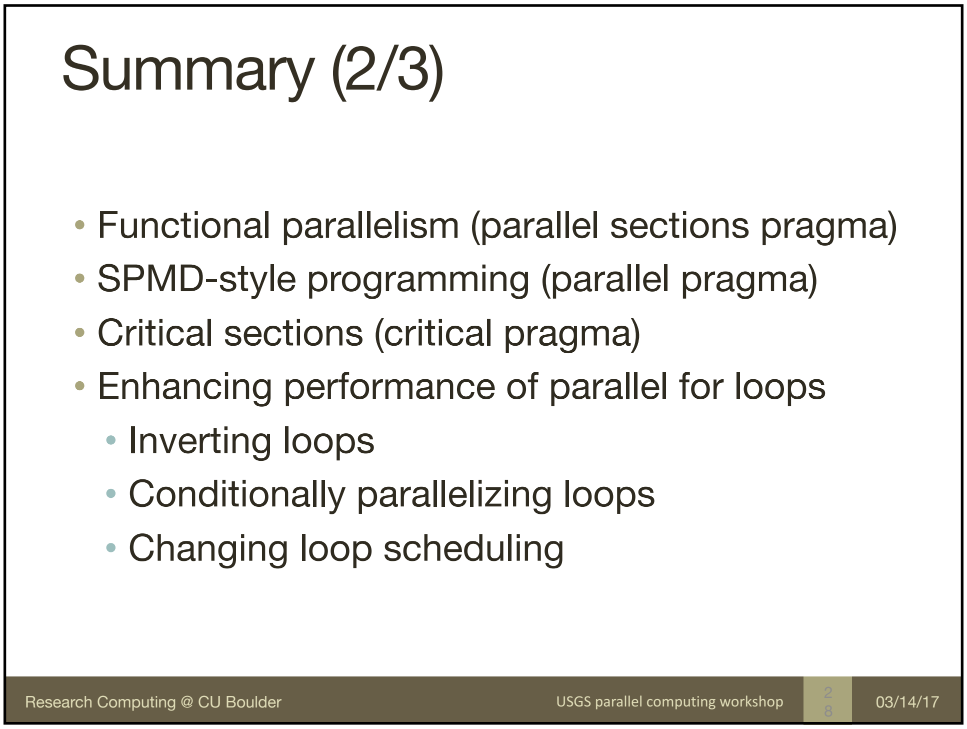
```
!$omp parallel
   !$omp sections
      v = alpha()
   !$omp section
      w = beta()
   !$omp end sections
   !$omp sections
      x = gamma(v, w)
   !$omp section
      y = delta()
   !$omp end sections
   write(*,*) epsilon(x,y)
!$omp end parallel
```

Research Computing @ CU Boulder USGS parallel computing workshop 26 03/14/17



Summary (1/3)

- OpenMP an API for shared-memory parallel programming
- Shared-memory model based on fork/join parallelism
- Data parallelism
  - parallel for pragma
  - reduction clause



Summary (2/3)

- Functional parallelism (parallel sections pragma)
- SPMD-style programming (parallel pragma)
- Critical sections (critical pragma)
- Enhancing performance of parallel for loops
  - Inverting loops
  - Conditionally parallelizing loops
  - Changing loop scheduling