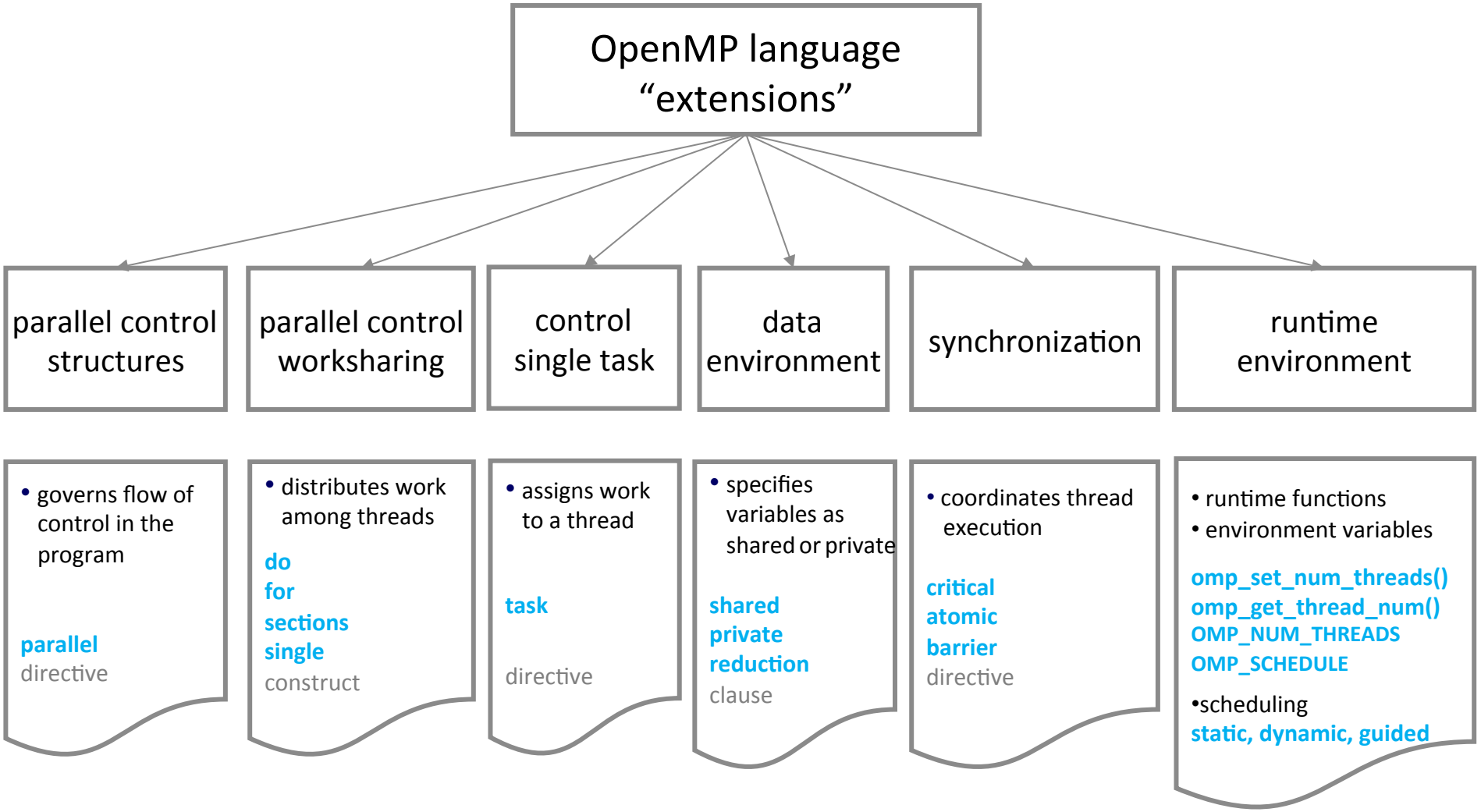
Parallel Computing for Science & Engineering CS395T

02/06/2018

Instructors:
Lars Koesterke, TACC
Charlie Dey, TACC

OpenMP Constructs



OpenMP Worksharing -- Single

- SINGLE (or MASTER)
  - Block of code is executed only once by a single thread (or the master thread)
  - Implied barrier (ONLY single)
- Only one thread is executing the block at all!

```
!$OMP single

  glob_count = glob_count + 1
  print *, glob_count

!$OMP end single
```

```
#pragma single
{
  glob_count++;
  printf("%d\n", glob_count);

}
```

Mutual exclusion – atomic and critical directives

When each thread must execute a section/statement of code serially (only one thread at a time can execute it) the region must be marked with CRITICAL/ATOMIC directives.

Use the ATOMIC directive if executing only one operation.

All threads are executing the block one after another

```
!$omp parallel shared(sum,x,y)
 ...
!$omp critical
   call update(x)
   call update(y)
   sum=sum+1
!$omp end critical
...
!$omp end parallel
```

```
 !$omp parallel
  ...
 !$omp atomic
       sum=sum+1
...
 !$omp end parallel
```

Syntax:

$pragma omp critical [*name*]

!$omp critical [*name*]

!$omp end critical [*name*]

TACC

Mutual exclusion – atomic and critical directives

When each thread must execute a section/statement of code serially (only one thread at a time can execute it) the region must be marked with CRITICAL/ATOMIC directives.

Use the ATOMIC directive if executing only one operation.

```
#pragma parallel shared(sum,x,y){
  ...
#pragma omp critical{
    update(x);
    update(y);
    sum=sum+1;
}
...
}
```

```
#pragma omp parallel{
  ...
#pragma omp atomic
        sum=sum+1;
...
}
```

Syntax:

$pragma omp critical [*name*]

!$omp critical [*name*]

!$omp end critical [*name*]

TACC

OpenMP Synchronization -- NOWAIT

Where a work-sharing region ends, a barrier is implied - all threads must reach the barrier before any can proceed. By using the NOWAIT clause at the end of each loop inside the parallel region, an unnecessary synchronization of threads can be avoided.

```
!$omp parallel
…

!$omp do
      do i=1,n
         call work(i)
      enddo
!$omp end do nowait

!$omp do schedule(dynamic,m)
      do i=1,n
         x(i)=y(i)+z(i)
      enddo

!$omp end parallel
```

```
#pragma omp parallel
{…

#pragma omp for nowait
      for(i=0; i<n; i++)
         work(i);



#pragma omp schedule(dynamic,m)
    for(i=0; i<n; i++)
       x(i)=y(i)+z(i);


}
```


Dynamic scheduling is used to allow early threads to do more work in second loop."

Runtime Library API

Functions	Operation
`omp_get_max_threads`	from OMP_NUM_THREADS
`omp_get_num_threads()`	Number of Threads in team,N.
`omp_get_thread_num()`	Thread ID. {0 -> N-1}
`omp_get_num_procs()`	Number of machine hardware threads
`omp_in_parallel()`	True if in parallel region & multiple thread executing
`omp_set_num_threads(#)`	Changes Number of Threads for parallel region.

API Example: Pseudo code

Assume: 'export OMP_NUM_THREADS=3'

```
print omp_num_procs()
print omp_max_threads()

omp parallel
  print omp_num_threads()
  print omp_get_thread_num()
omp end parallel

omp_set_num_threads(6)
print omp_num_procs()
print omp_max_threads()


omp parallel num_threads(2)
  print omp_num_threads()
  print omp_get_thread_num()
omp end parallel
```

```
16
3


3 printed 3 times
0, 1, or 2 each printed once



?
?



?, printed how often?
?
```

TACC

API example

Or can use OPENMP pragma (Fortran: compile with -fpp)

```
#include <omp.h>

#pragma omp parallel private(i,n)
{
  i  = omp_get_thread_num( ) ;
  n = omp_get_num_threads( );

}
```

```
use omp_lib

!$omp parallel private(i,n)

      i = omp_get_thread_num( )
     n = omp_get_num_threads( )

!$omp end parallel
```

What happens if you don't compile with –qopenmp?

OpenMP Conditional Compilation

Fortran triggers: !$

```
i=0; n=1
!$omp parallel private(i,n)
!$      i = omp_get_thread_num( )
!$      n = omp_get_num_threads( )
        call sub(i,n)
!$omp end parallel
```

Or can use OPENMP pragma (Fortran: compile with -fpp)

```
#include <omp.h>
i=0; n=1;
#pragma omp parallel private(i,n)
{
  #ifdef  OPENMP
      i = omp_get_thread_num( ) ;
     n = omp_get_num_threads( );
  #endif
      sub(i,n);
}
```

```
use omp_lib
i=0; n=1
!$omp parallel private(i,n)

  #ifdef  OPENMP
      i = omp_get_thread_num( )
     n = omp_get_num_threads( )
  #endif
     call sub(i,n)
!$omp end parallel
```

Runtime Library API

API Environment Variables

OMP_NUM_THREADS	Set to No. of Threads
OMP_SCHEDULE	Schedule for the run-time clause

Example

```
export OMP_NUM_THREADS=4
export OMP_SCHEDULE='static,4'
./a.out
```

References

- Some material identical to:
  www.chpc.utah.edu/attachments/20110112.05/IntroOpenMP06.pdf
- This one is a real tutorial and even has test modules:
  http://www.citutor.org/login.php
- The sites
  www.llnl.gov/computing/tutorials/openMP/
  www.nersc.gov/assets/Uploads/XE62011OpenMP.pdf
  have good reference/tutorial pages for OpenMP.

Examples

```
export OMP_NUM_THREADS=2
./a.out

x and y : arrays with 2 elements
x(1) = 2.; x(2) = 4.
j = 0

!$omp parallel private(i)
!$omp do
do i=1 , 2
  j = j + 1
  y(i) = x(j)
enddo
!$omp end parallel

What values in x(1) and x(2)?
```

Splitting dependences out of loop

Loop fission

```
for(i=1; i<n; i++){
  b[i  ] = a[i  ]*r[i  ]
  a[i-1] = t[i-1]*s[i-1]
}
```

```
for(i=1; i<n; i++)
  b[i  ] = a[i  ]*r[i  ]

for(i=1; i<n; i++)
  a[i-1] = t[i-1]*s[i-1]
```

C/C++

```
do i=2,n
  b(i  ) = a(i  )*r(i  )
  a(i-1) = s(t-1)*t(t-1)
enddo
```

```
do i=2,n
  b(i  ) = a(i  )*r(i  )
enddo
do i=2,n
  a(i-1) = s(t-1)*t(t-1)
enddo
```

F90

Examples

```
export OMP_NUM_THREADS=2
./a.out

x : array with 2 elements
x(1) = 2.; x(2) = 4.
sum = 0

!$omp parallel
!$omp do
do i=1 , 2
  sum = sum + x(i)
enddo
!$omp end parallel
```

Examples

```
export OMP_NUM_THREADS=4
./a.out

x : array with 8 elements
'all elements set to 1'
sum = 0

!$omp parallel reduction(+:sum)
!$omp do
do i=1 , 8
  sum = sum + x(i)
enddo
!$omp end parallel
```

Examples

```
export OMP_NUM_THREADS=4
./a.out

x : array with 8 elements
'all elements set to 1'
sum = 0

!$omp parallel reduction(+:sum)
!$omp do
do i=1 , 8
  sum = sum + x(i) * y(i) + fct(…)
enddo
!$omp end parallel
```

Examples

```
export OMP_NUM_THREADS=4
./a.out

x(4,4) : array with 4x4 elements

!$omp parallel
!$omp do schedule(static,1)
do i=1, 4
  do j=1, i
    x(i,j) = x(i,j) + 1.
  enddo
enddo nowait

!$omp do schedule(static,1)
do i=1, 4
   …
enddo
```