Parallel Computing for Science & Engineering CS395T

2/08/07

Instructors:

Dr. Bill Barth, Research Associate, TACC

Dr. Kent Milfeld, Research Associate, TACC

OpenMP Data Scoping

1
```
sum = 10.0
!$omp parallel do reduction(+:sum)
      do i = 1, 10
        sum = sum + a(i)
      end do
```

2
```
!$omp parallel do lastprivate(tmp)
      do i = 1, 100
        tmp = a(i)
      end do
      print *, ‘a(100) = = ', tmp
```

3
```
logical :: torf=.true.
!$omp parallel firstprivate(torf)
  do while(torf)
     torf  = do_work()
 end do
!$omp end parallel
```

1
```
sum = 10.0;
#pragma omp parallel for reduction(+:sum)
      for (i=0; i<10; i++)
        sum = sum + a(i);
```

2
```
#pragma omp parallel for lastprivate(tmp)
      for(i=0; i<100; i++){
         tmp = a(i);
       }
       print *, ‘a(100) = = ', tmp
```

3
```
int torf = 1;
#pragma omp parallel firstprivate(torf)
{
  while(torf)
     torf  = do_work();
}
```

1.) Each thread's copy of sum is added to original sum at end of loop
2.) tmp is equal to a(100) at end of loop
3.) Each thread repeats (picks up) work until work function returns false.

OpenMP Data Scoping

An operation that “combines” multiple elements to form a single result, such as a summation, is called a reduction operation. A variable that accumulates the result is called a reduction variable. In parallel loops reduction operators and variables must be declared.

```
real*8 asum, aprod
      ...
!$omp  parallel do   &
!$omp reduction(+:asum )
!$omp reduction(*:aprod)
      do i=1,n
         asum  = asum  + a(i)
         aprod = aprod * a(i)
      enddo
print*, asum, aprod
```

```
double asum, aprod
      ...
#pragma omp parallel for \
      reduction(+:asum)  \
      reduction(*:aprod)
      for(i=0;i<n; i++){
         asum  = asum  + a(i);
         aprod = aprod * a(i);
      }
printf(“%f %f\n”, asum,aprod);
```

Each thread has a private ASUM and APROD, initialized to the operator’s identity, 0 & 1, respectively. After the loop execution, the master thread collects the private values of each thread and finishes the (global) reduction.

OpenMP Synchronization

CRITICAL

- All threads execute the block of code
- But, only one thread can be executing block at any time. Not required to be in Parallel Scope

ATOMIC

- Only applies to a single assignment statement that updates a scalar variable. Designed to be implemented with machine instructions that perform “read, modify, and write” operations on memory atomically. Has form: x = *intrinsic*(x,expr)
- Not required to be in Parallel Scope

Barrier

- Each thread of the team waits for all others to arrive at the barrier (classical synchronization).
- Cannot be executed in a work-sharing construct.

OpenMP Synchronization -- NOWAIT

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

Use the ATOMIC directive if executing only one operation.

```
!$omp parallel shared(sum,x,y)
 ...
!$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
```

Mutual exclusion – atomic and critical directives

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

Use the ATOMIC directive if executing only one operation.

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

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

OpenMP Synchronization

- BARRIER
  - Threads in a team wait until entire team reaches the barrier

```
!$omp parallel
…
!$omp do reduction(+:s)
   do i = 1, 100
      s = s + f(i)
    end do
!$omp atomic
    s = s + extra
!$omp barrier
    print*, s
!$omp end parallel
```

```
#pragma omp parallel
{ …
#pragma omp for reduction(+:s)
   for(i=0; i<100; i++)
      s = s + f(i);

#pragma omp atomic
    s = s + extra;
#pragma omp barrier
    printf("%f\n", s);
}
```

OpenMP Synchronization -- NOWAIT

When a work-sharing region is exited, 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);


}
```

TACC

Merging Parallel Regions

The PARALLEL directive declares an entire region as parallel. Merging work-sharing constructs into a single parallel region eliminates the overhead of separate team formations.

```
!$omp parallel

  !$omp do
      do i=1,n
         a(i)=b(i)+c(i)
      enddo
  !$omp end do

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

!$omp end parallel
```

```
!$omp parallel do

      do i=1,n
         a(i)=b(i)+c(i)
      enddo
!$omp end parallel do

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

!$omp end parallel do
```

Runtime Library API

Functions	Operation
`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 CPUs.
`omp_in_parallel()`	True if in parallel region & multiple thread executing
`omp_set_num_threads(#)`	Changes Number of Threads for parallel region.

For C, use include file: #include <omp.h>

Runtime Library API

API Dynamic Scheduling

omp_get_dynamic()	True if dynamic threading is on.
omp_set_dynamic()	Set state of dynamic threading (true/false)

API Environment Variables

OMP_NUM_THREADS	Set to No. of Threads
OMP_DYNAMIC	TRUE/FALSE for enable/disable dynamic threading

Runtime Library API

Mutual exclusion- lock routines

When each thread must execute a section of code serially (only one thread at a time can execute it) , locks provide a more flexible way of ensuring serial access than CRITICAL and ATOMIC directives. Locks are not tied to blocks of code– can be executed anywhere. Used for recursion. ("Nested" form eliminates deadlock to same thread setting lock again. Also, see omp_test_lock.)

```
call omp_init_lock(maxlock)
!$omp parallel shared(x,y)
...
call omp_set_lock(maxlock)
call update(x)
call omp_unset_lock(maxlock)
...
!$omp end parallel
call omp_destroy_lock(maxlock)
```

```
   omp_init_lock(&maxlock);
#pragma omp parallel shared(x)
{...
   omp_set_lock(&maxlock);
   update(x);
   omp_unset_lock(&maxlock);
...
}
   omp_destroy_lock(&maxlock);
```

OpenMP Conditional Compilation

FORTRAN with a !$, C$ or *$ trigger

```
i=1; 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 macro in cpp (Fortran or C)

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

#idef _OPENMP
      i = omp_get_thread_num( ) ;
     n = omp_get_num_threads( ;)
#endif
      call sub(i,n);
!$omp end parallel
```

```
i=1; n=1;
#pragma omp parallel private(i,n)
{
#idef _OPENMP
     i = omp_get_thread_num( ) ;
    n = omp_get_num_threads( ;)
#endif
      sub(i,n);
}
```

Variable Scoping, Fortran

scope

```
program main
integer, parameter :: nmax=100
common /vars/ y(nmax)
real*8  :: x(n,n)
integer :: n, j
...
n=nmax; y=0.0
!$omp parallel do
   do j=1,n
      call adder(x,n,j)
   end do
...
end program main
```

lexical extent

```
subroutine adder(a,m,col)
integer, parameter :: nmax=100
common /vars/ y(nmax)
real*8  :: a(m,m)
integer :: m,col
save array_sum = 0.0
...
do i=1,m
   y(col)=y(col)+a(i,col)
end do
array_sum=array_sum+y(col)
...
end subroutine adder
```

dynamic extent

TACC

Default variable scoping in Fortran

Variable	Scope	Is use safe?	Reason for scope
n	shared	yes	declared outside parallel construct
j	private	yes	parallel loop index variable
x	shared	yes	declared outside parallel construct
y	shared	yes	common block
i	private	yes	parallel loop index variable
m	shared	yes	actual variable *n* is shared
a	shared	yes	actual variable *x* is shared
col	private	yes	actual variable *j* is private
array_sum	shared	no	declared with SAVE attribute

Variable Scoping, C

scope

```
#define NMAX
double y[NMAX][NMAX];
main (int argc,char*argv[]){
int n, j;
double x[NMAX]
...
n=NMAX;
for(j=0;j<n;j++)y[i]=0.0;
#pragma omp parallel for
   for(j=0;j<n;j++){
      adder(x,n,j)
   }
...
}
```

lexical extent

```
int adder(double* a,
        int nsub,int jsub){
double sum = 0.0;
int i;

 for(i=0; i<nsub; i++){
  a(jsub)=a(jsub)+y(i,jsub);
 }
 sum=sum+x(jsub);

}
```

dynamic extent

Default variable scoping in Fortran

Variable	Scope	Is use safe?	Reason for scope
n	shared	yes	declared outside parallel construct
j	private	yes	parallel loop index variable
x	shared	yes	declared outside parallel construct
y	shared	yes	global
i	private	yes	parallel loop index variable
nsub	private	yes	actual variable *n* is shared
a	shared	yes	actual variable *x* is shared
jsub	private	yes	actual variable *j* is private
sum	private	No/yes	probably want global sum

Introduction : OpenMP “Hello”

```
program hello
integer :: omp_get_thread_num

   print*, "hello, main“

!$omp  parallel
   print*, "thrd=",&
       omp_get_thread_num()
!$omp  end parallel

end program
```

```
#include <omp.h>
int main(int argc, char* argv[ ]){

   printf("hello, main\n");

#pragma omp parallel
   {
     printf("thr=%d\n",
         omp_get_thread_num());
    }
}
```

champion

```
xlf90_r –O3 -qsource -qsmp=omp:noauto hello.f90
 xlc_r  –O3 -qsource -qsmp=omp:noauto hello.c
```

lonestar

```
ifort -O3 -openmp hello.f90
icc   -O3 -openmp hello.c
```

TACC

Introduction : OpenMP “Hello”

- Set OMP_NUM_THREADS to 4
- Within OpenMP directives three additional copies of the code are started (what does this mean?)
- Each copy is called a thread or thread of execution
- The OpenMP routine omp_get_thread_num() reports unique thread# between 0 and OMP_NUM_THREADS-1

Introduction : OpenMP “Hello, World”

- Output after running on 4 threads :

  hello, main

  thrd=1

  thrd=0

  thrd=3

  thrd=2

- Analysis of OpenMP output :
  - Threads are working completely independently
  - Threads may have to cooperate to produce correct results, requiring synchronization

References

- Some material identical to: http://www.ascc.net/compsrv/sysmgnt/euler/workshop/IBM-Compilers.pdf
- This one is a real tutorial and even has test modules:http://webct.ncsa.uiuc.edu:8900/public/OPENMP/
- The sites
  http://www.llnl.gov/computing/tutorials/openMP/
  http://www.nersc.gov/nusers/help/tutorials/openmp
  have good reference/tutorial pages for OpenMP.