Version 2013-1
as of 31 May 2013

Parallel Computing for Science and Engineering

OpenMP Part 2

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.

Fortran

```
real*8 asum, aprod
      ...
asum=0.0; aprod=1.0;
do i=1,n
   asum  = asum  + a(i)
   aprod = aprod * a(i)
enddo
print*, asum, aprod
```

C/C++

```
double asum, aprod
      ...
asum=0.0; aprod=1.0;
for(i=0;i<n; i++){
   asum  = asum  + a[i];
   aprod = aprod * a[i];
}
printf("%f %f\n", asum,aprod);
```

OpenMP Data Scoping

Fortran

```
asum=0.0; aprod=1.0

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

C/C++

```
asum=0.0; aprod=1.0;

#pragma omp parallel
{     ...
#pragma omp 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 Data Scoping

Fortran

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
```

C/C++

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 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 Region

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)
- Uses hardware support
- Not required to be in Parallel Region
- Intrinsic function or simple expression
- x = min(x,a+b); x = x + 1; x += 1

Barrier

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

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`** 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
```

Syntax:

```
$pragma omp critical [name]
!$omp critical [name]
!$omp end critical [name]
```

OpenMP Synchronization

- **barrier**
  - Threads in team wait until entire team reaches barrier

Fortran

```
!$omp parallel
…
!$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
```

C/C++

```
#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 worksharing 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.

Fortran

```
!$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
```

C/C++

```
#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."

Merging Parallel Regions

- The **`parallel`** directive declares an entire region as parallel. Merging worksharing constructs into a single parallel region eliminates the overhead of separate team formations.

```
!$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
```

```
!$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
```

OpenMP Conditional Compilation

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

```
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** definition in cpp (Fortran or C)

```
i=0; n=1;        ! include "omp_lib.h"  above
!$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
```

Fortran

```
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);
}
```

C/C++

Variable Scoping in 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,icol)
integer,parameter :: nmax=100
common /vars/ y(nmax)
real*8  :: a(m,m)
integer :: m,icol
save sum = 0.0
...
do i=1,m
   y(icol)=y(icol)+a(i,icol)
end do

sum=sum+y(icol)
...
end subroutine adder
```

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	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
`icol`	private	yes	actual variable j is private
`array_sum`	shared	no	declared with SAVE attribute

Variable Scoping in C

Scope

```
#define NMAX 100
double y[NMAX][NMAX], sum=0.0;
main (){
int n, j;
double x[NMAX];
...
n=NMAX;
for(j=0;j<n;j++)x[j]=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){
int i;

 for(i=0; i<nsub; i++){
  a[jsub]=a[jsub]+y[i][jsub];
 }

 sum=sum+a[jsub];

}
```

dynamic extent

Default Variable Scoping in C

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 everywhere
`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

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>`

For Fortran, use include file: `include "omp_lib.h"`

For F90, use module file: `use omp_lib`

Runtime Library API

- API Environment Variables

`OMP_NUM_THREADS`	Set to Number of Threads
`OMP_DYNAMIC`	TRUE/FALSE for enable/disable dynamic threading

- API Dynamic Scheduling

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

Runtime Library API

!$ (integer parameter openmp_version)	Conditional for using library with/ without OpenMP Support

Example:	`integer :: it=0`
	…
	`!$omp parallel private(it)`
	`!$ it = omp_get_thread_num()`

_OPENMP	Defined as yyyymm of release

Example:	`int it=0;`
	…
	`#pragma omp parallel private(it)`
	`ifdef _OPENMP`
	`  it=omp_get_thread_num();`
	`#endif`

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.

We gratefully acknowledge the sponsorship of Chevron Corporation, whose generous support of TACC has made possible this Scientific Computing Curriculum and other student-focused initiatives.

License