Introduction to High Performance Scientific Computing

Autumn, 2016

Lecture 12

Imperial College London

Prasun Ray
17 November 2016

Today

Introduction to OpenMP

Getting started

Parallel regions

Parallel loops

Imperial College London

Overview

- OpenMP provides a fairly easy approach to parallelizing c/c++ or fortran code
- Add *directives* indicating how/where the code should run in parallel
- Parallel regions have multiple threads, each of which should be assigned computational tasks
  - OpenMP is for *shared-memory* parallel programming
  - Each thread has access to all variables that existed before parallel region was created
  - This can cause problems if multiple threads try to change the same variable!
- Particularly useful for parallelizing loops
- When compiling, add *–fopenmp* flag

Imperial College London

Overview

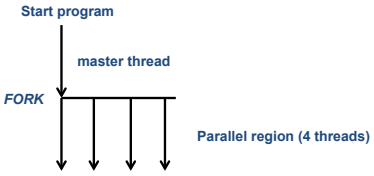
Program starts with single *master thread*

Overview

Program starts with single *master thread*

Then, launch parallel region with multiple threads.

Each thread has access to all variables introduced previously



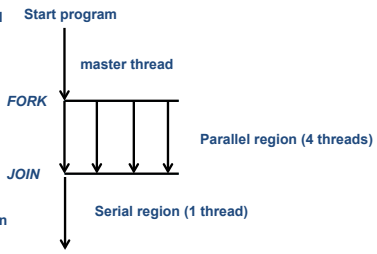
Overview

Program starts with single *master thread*

Then, launch parallel region with multiple threads.

Each thread has access to all variables introduced previously

Can end parallel region if/when desired and launch parallel regions again in future as needed



Simple OpenMP example

- **Launch parallel region, get info on threads (see *firstomp_v0.f90*)**
- **Must use openMP module, *omp_lib***
  - **This makes functions like *omp_get_num_threads* available**

```
!Getting started with OpenMP

program firstomp
    use omp_lib !makes OpenMP routines, variables available
    implicit none
    integer :: NumThreads,threadID
```


Simple OpenMP example

- **Launch parallel region, get info on threads (see *firstomp_v0.f90*)**
- **Must use openMP module, *omp_lib***
  - **This makes functions like *omp_get_num_threads* available**

```
!Getting started with OpenMP

program firstomp
    use omp_lib !makes OpenMP routines, variables available
    implicit none
    integer :: NumThreads,threadID
!$OMP PARALLEL
    NumThreads = omp_get_num_threads()
    threadID = omp_get_thread_num()
    print *, 'this is thread',threadID, ' of ', NumThreads
!$OMP END PARALLEL
```

- !$OMP **starts an OpenMP directive (#pragma omp in c)**
- !$OMP PARALLEL **starts a parallel region (forks a number of threads)**
  - omp_get_num_threads **tells us how many threads are forked**
  - omp_get_thread_num **tells us which thread is being used**

Simple OpenMP example

- **Let's compile and run this:**

```
$ gfortran -fopenmp -o testv0.exe firstomp_v0.f90
$ ./testv0.exe
 this is thread        1  of        4
 this is thread        1  of        4
 this is thread        1  of        4
 this is thread        1  of        4
```

- **Total number of threads is correct, but problem getting the thread id.**

Simple OpenMP example

- Let's compile and run this:

```
$ gfortran -fopenmp -o testv0.exe firstomp_v0.f90
$ ./testv0.exe
 this is thread        1  of        4
 this is thread        1  of        4
 this is thread        1  of        4
 this is thread        1  of        4
```

- Total number of threads is correct, but problem getting the thread id.
- Remember: *threadID* is a ***shared*** variable.
  - Each thread is writing to the same variable with it's id, so only the "last" thread has its ID displayed

Simple OpenMP example

- Let's compile and run this:

```
$ gfortran -fopenmp -o testv0.exe firstomp_v0.f90
$ ./testv0.exe
 this is thread        1  of        4
 this is thread        1  of        4
 this is thread        1  of        4
 this is thread        1  of        4
```

- Total number of threads is correct, but problem getting the thread id.
- Remember: *threadID* is a ***shared*** variable.
  - Each thread is writing to the same variable with it's id, so only the "last" thread has its ID displayed
- How can we fix this? First approach: define a *critical* region…

Simple OpenMP example

- A critical region, defined with !$OMP CRITICAL, runs in serial
  - The threads carry out their tasks sequentially (***firstomp_v1.f90***)

```
!$OMP PARALLEL
    NumThreads = omp_get_num_threads()
    !$OMP CRITICAL
        threadID = omp_get_thread_num()
        print *, 'this is thread',threadID, ' of ', NumThreads
    !$OMP END CRITICAL
!$OMP END PARALLEL
```

Simple OpenMP example

- A critical region, defined with !$OMP CRITICAL, runs in serial
  - The threads carry out their tasks sequentially (*firstomp_v1.f90*)

```
!$OMP PARALLEL
   NumThreads = omp_get_num_threads()
   !$OMP CRITICAL
      threadID = omp_get_thread_num()
      print *, 'this is thread',threadID, ' of ', NumThreads
   !$OMP END CRITICAL
!$OMP END PARALLEL
```

So now, if we compile and run:

```
$ ./testv1.exe
 this is thread           0  of           4
 this is thread           2  of           4
 this is thread           1  of           4
 this is thread           3  of           4
```


Simple OpenMP example

- Critical regions useful for: data I/O, displaying results
- But forcing serial execution in a parallel region, not generally desirable
- Better approach: set threadID to be a *private* variable (*firstomp.f90)*
  - Then, each thread will have their own private copy of the variable

Simple OpenMP example

- Critical regions useful for: data I/O, displaying results
- But forcing serial execution in a parallel region, not generally desirable
- Better approach: set threadID to be a *private* variable (*firstomp.f90)*
  - Then, each thread will have their own private copy of the variable

```
!$OMP PARALLEL PRIVATE(threadID)
   NumThreads = omp_get_num_threads()
   threadID = omp_get_thread_num()
   print *, 'this is thread',threadID, ' of ', NumThreads
!$OMP END PARALLEL
```

```
$ ./test.exe
 this is thread           0  of           4
 this is thread           2  of           4
 this is thread           1  of           4
 this is thread           3  of           4
```

Simple parallel calculation

- Can use *threadID* to assign tasks to threads:

```
!$OMP PARALLEL PRIVATE(threadID)
   NumThreads = omp_get_num_threads()
   threadID = omp_get_thread_num()

    if (threadID==0) then
        call subroutine1(in1,out1)
    elseif (threadID==1) then
        call subroutine1(in2,out2)
    end if

!$OMP END PARALLEL
```

- Important to distribute work evenly across threads (load balancing)

Overview

- OpenMP (primarily) consists of *directives* and *routines*
- Directives are denoted with !$OMP
  - !$OMP parallel, !$OMP critical, …
  - Directives are recognized when –fopenmp compile-flag is used
  - Otherwise, they are interpreted as comments
    - What happens if you use:
      !$ print *, "compiled with –fopenmp"
- Routines are available via the use omp_lib command
  - e.g. omp_get_thread_num and omp_get_num_threads

Parallel loops

- Loops form the backbone of most scientific codes
- They should be parallelized whenever possible
- They can be parallelized if the calculations of each iterations are independent of each other (no data dependencies)

```
do i1 = 1,n
    x(i1) = y(i1) + z(i1)
end do
```

Ok to parallelize

```
do i1 = 1,n
    norm = norm + abs(x(i1))
end do
```

Can't parallelize easily: each thread updating, *norm*

Parallel loops

- OpenMP makes it very easy to parallelize loops

```
!$OMP parallel do
do i1 = 1,n
    x(i1) = y(i1) + z(i1)
end do
!$OMP end parallel do
```

- OpenMP automatically distributes iterations across threads
  - If NumThreads=2 and n=10, iterations 1,…,5 would be given to thread 0 and iterations 6,…,10 would be done by thread 1 (or vice versa)
  - The iterated variable, *i1*, is automatically set to *private*. Each thread has its own copy.

Parallel loops

- Simple example (*loop_omp1.f90*):

```
!$OMP parallel do private(threadID)
do i1 = 1,size(x)
    x(i1) = y(i1) + z(i1)
    threadID = omp_get_thread_num()
    print *, 'iteration ',i1,' assigned to thread ',threadID
end do
!$OMP end parallel do

print *, 'test:', maxval(abs(x-y-z))
```

- Note: *threadID* again set to *private*
- Compile and run…

Parallel loops

- Simple example (*loop_omp1.f90*):

```
!$OMP parallel do private(threadID)
do i1 = 1,size(x)
    x(i1) = y(i1) + z(i1)
    threadID = omp_get_thread_num()
    print *, 'iteration ',i1,' assigned to thread ',threadID
end do
!$OMP end parallel do

print *, 'test:', maxval(abs(x-y-z))
```

```
$ gfortran -fopenmp -o testl1.exe loop_omp1.f90
$ ./testl1.exe
 interation            1  assigned to thread           0
 interation            3  assigned to thread           2
 interation            2  assigned to thread           1
 interation            4  assigned to thread           3
 test:   2.2204460492503131E-016
```

Parallel loops

- Can easily "embed" parallel loop in parallel region:

```
!$OMP parallel
!$OMP do
do i1 = 1,n
    x(i1) = y(i1) + z(i1)
end do
!$OMP end do

!Other parallel calculations

!$OMP end parallel
```


Parallel loops

- Now, return to "unparallelizable" example:

```
do i1 = 1,n
    norm = norm + abs(x(i1))
end do
```

- Next lecture: use *reduction* to parallelize
- Now:
  1. Let each thread have it's own copy of norm
  2. Sum each thread's partial sum in *critical* region

Parallel loops

1. Let each thread have it's own copy of norm

```
norm=0.d0
partial_norm=0.d0
!$OMP parallel firstprivate(partial_norm),private(threadID)
!$OMP do
do i1 = 1,size(x)
    partial_norm = partial_norm + abs(x(i1))
end do
!$OMP end do
```

- The partial sum, *partial_norm*, is a private variable which must be initialized
- *firstprivate* initializes each thread's value to the value set before the parallel region

Parallel loops

1. Let each thread have it's own copy of norm
2. Sum each thread's partial sum in *critical* region (see *norm_omp1.f90)*

```
norm=0.d0
partial_norm=0.d0
!$OMP parallel firstprivate(partial_norm),private(threadID)
!$OMP do
do i1 = 1,size(x)
    partial_norm = partial_norm + abs(x(i1))
end do
!$OMP end do

!$OMP critical
threadID = omp_get_thread_num()
print *, 'Thread number:',threadID, 'partial norm=',partial_norm
norm = norm + partial_norm
!$OMP end critical
```


Parallel loops

Compile and run, testing code with:

```
print *, 'test:',norm-sum(abs(x))
```

```
$ gfortran -fopenmp -o testn1.exe norm_omp1.f90
$ ./testn1.exe
 Thread number:           2 partial norm=   3.1411200080598674
 Thread number:           3 partial norm=   3.2431975046920716
 Thread number:           1 partial norm=   2.9092974268256819
 Thread number:           0 partial norm=   1.8414709848078965
 test:   0.0000000000000000
```


Parallel loops: nested loops

- Often work with nested loops:

```
do j1 = 1,N
    do i1 = 1,M
        x(i1,j1) = y(i1,j1) + z(i1,j1)
    end do
end do
```

- Should we parallelize the inner or outer loop? (assuming M ~ N)

```
!$OMP parallel do private(i1)
do j1 = 1,N
    do i1 = 1,M
        x(i1,j1) = y(i1,j1) + z(i1,j1)
    end do
end do
!$OMP end parallel do
```

outer

Parallel loops: nested loops

- Often work with nested loops:

```
do j1 = 1,N
    do i1 = 1,M
        x(i1,j1) = y(i1,j1) + z(i1,j1)
    end do
end do
```

- Should we parallelize the inner or outer loop? (assuming M ~ N)

```
do j1 = 1,N
   !$OMP parallel do private
    do i1 = 1,M
        x(i1,j1) = y(i1,j1) + z(i1,j1)
    end do
   !$OMP end parallel do
end do
```

inner

Parallel loops: nested loops

- Should we parallelize the inner or outer loop? (assuming M ~ N)

```
!$OMP parallel do private(i1)
do j1 = 1,N
    do i1 = 1,M
        x(i1,j1) = y(i1,j1) + z(i1,j1)
    end do
end do
!$OMP end parallel do
```

Better to parallelize outer thread (setting the inner variable, i1, to private)

If inner loop is parallelized: forking/joining of threads is repeated with each outer loop: this is inefficient!

Parallel loops: nested loops

Must always be sure loop(s) can be parallelized

Example:

```
!$OMP parallel do private(i1)
do j1 = 2,N
    do i1 = 1,M
        x(i1,j1) = x(i1,j1-1)
    end do
end do
!$OMP end parallel do
```

Incorrect

- Different j1's are assigned to different threads
- x(i1,j1-1) may not have been computed at the time that it is needed

Parallel loops: nested loops

Must always be sure loop(s) can be parallelized

Example:

```
!$OMP parallel do private(j1)
do i1 = 1,M
    do j1 = 2,N
        x(i1,j1) = x(i1,j1-1)
    end do
end do
!$OMP end parallel do
```

Correct

- Solution: swap inner and outer loops
- Now, computation of x is "safe."
  - The "i1 loop" is parallelized, and calculations of x do not depend on the order in which i1 is iterated.

Imperial College London