Introduction to High Performance Scientific Computing

Autumn, 2016

Lecture 13

Imperial College London

Prasun Ray
21 November 2016

Today

More on OpenMP

Reductions

Setting number of threads

A few useful OpenMP commands

Imperial College London

Last time: Parallel 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
```

Is the order of the iterations important? (data dependency)

Do different iterations assign values to same variable? (race condition)

Imperial College London

Last time: Parallel 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
```

Imperial College
London

Example: computing a norm

Last time: developed simple code for computing norm: sum(|x|)

Serial version:

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

Imperial College
London

Example: computing a norm

Last time: developed simple code for computing norm: sum(|x|)

Serial version:

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

Parallel version:

```
!$OMP parallel firstprivate(partial_norm)
!$OMP do
do i1 = 1,size(x)
    partial_norm = partial_norm + abs(x(i1))
end do
!$OMP end do

!$OMP critical
norm = norm + partial_norm
!$OMP end critical
!$OMP end parallel
```

Example: computing a norm

- Typically want to avoid using critical regions
- *reduction* provides a simpler approach:

```
!$OMP parallel do reduction(+:norm)
do i1 = 1,size(x)
    norm = norm + abs(x(i1))
end do
!$OMP end parallel do
```

Imperial College London

Example: computing a norm

- Typically want to avoid using critical regions
- *reduction* provides a simpler approach (*omp_norm2.f90*):

```
!$OMP parallel do reduction(+:norm)
do i1 = 1,size(x)
    norm = norm + abs(x(i1))
end do
!$OMP end parallel do
```

- Generally, *reduction* "reduces" an array of numbers distributed across multiple threads to a single number
- Several operations are available, a few common operators are: +,-,*,max,min,.and,.or.
- Not specific to OpenMP! In MPI, we will use *MPI_REDUCE.*
- Due to ease-of-use and usefulness, one of the most important tools in parallel computing!

Imperial College London

Example: reduction with *min*

- Here, computation of x is parallelized
- Reduction is used to find min(|x|)

```
!$OMP parallel do reduction(min:xmin)
do i1=1,size(x)
    x(i1) = z(i1)+y(i1)
    xmin = min(abs(x(i1)),xmin)
end do
!$OMP end parallel do
```

Imperial College London

Setting number of threads

- By default, OpenMP "detects" the number of threads on computer and uses all of them
- Can also set threads in two ways:

  1. Within code with *omp_set_num_threads*, e.g.:
  ```
  !$ call omp_set_num_threads(2)
  ```
  (the "!$" ensures this is only called if *–fopenmp* flag is used when compiling*)*

  2. From Unix terminal before program execution:
  ```
  $  export OMP_NUM_THREADS=2
  ```

Imperial College London

Other useful OpenMP directives

- Consider a parallel region of code:

```
!$OMP parallel

!code run by *each* thread

!$OMP end parallel
```

- There are a number of directives which we can use in the parallel region

Imperial College London

Other useful OpenMP directives

- do-loops

```
!$OMP parallel private(i1)

do i1=1,N
    !some operations
end do

!$OMP end parallel
```

- In the example above, the full do-loop is run by each thread

Imperial College London

Other useful OpenMP directives

- do-loops

```
!$OMP parallel private(i1)

do i1=1,N
    !some operations
end do

!$OMP end parallel
```

- In the example above, the full do-loop is run by each thread

```
!$OMP parallel private(i1)
!$OMP do
do i1=1,N
    !some operations
end do
!$OMP end do
!$OMP end parallel
```

- Now, this is the same as a parallel do loop

Imperial College London

Other useful OpenMP directives

- Sections

```
!$OMP parallel

!$OMP sections

!$OMP section
   !code run by one thread

!$OMP section
   !code run by second thread

!$OMP section
   !code run by another thread

!$OMP end sections


!$OMP end parallel
```

- Manually assign tasks to threads
- For example, invert four matrices (of the same size)
- Could have four “sections”, one for each matrix inversion

Imperial College London

Last lecture: 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)

Imperial College London

Simple parallel calculation

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

```
!$OMP PARALLEL PRIVATE(threadID)
   NumThreads = omp_get_num_threads()
   threadID = omp_get_thread_num()
   !$OMP sections
      !$OMP section
         call subroutine1(in1,out1)
      !$OMP section
         call subroutine1(in2,out2)
   !$OMP end sections
!$OMP END PARALLEL
```

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

Imperial College London

Other useful OpenMP directives

- Single

```
!$OMP parallel
!$OMP single
   !code run by only one thread
!$OMP end single
!$OMP end parallel
```

- Used to run commands only once within parallel region
- Useful for: print statements, data input/output

Imperial College London

Other useful OpenMP directives

- Single

```
!$OMP parallel
!$OMP single
   !code run by only one thread
!$OMP end single nowait
!$OMP end parallel
```

- Used to run commands only once within parallel region
- Useful for: print statements, data input/output
- Add *nowait* tag to allow other threads to continue while one thread is in *single* region

Imperial College London

Synchronization

- Some threads may be given more work than others
- One thread may complete its tasks quickly and move very far ahead of the other threads
- *Barriers* keep the threads synchronized:

```
!$OMP parallel
!Some code
!$OMP barrier
!$OMP end parallel
```

- Threads will not continue past the barrier until all threads reach the barrier

Imperial College London

Synchronization

- Some threads may be given more work than others
- One thread may complete its tasks quickly and move very far ahead of the other threads
- *Barriers* keep the threads synchronized:

```
!$OMP parallel
!Some code
!$OMP barrier
!$OMP end parallel
```

- Threads will not continue past the barrier until all threads reach the barrier
- There are *implicit* barriers at end of !$OMP do and !$OMP single blocks

Imperial College London

Thread-safe routines

- What happens when you call sub-program from within parallel region?
- Each thread will call it's own "copy" of sub-program
  - All "local" variables declared within sub-program are private to thread

```
!$OMP parallel
call sub1(in1,in2,out1,out2)
!$OMP end parallel
!---------------------------------
subroutine sub1(in1,in2,out1,out2)
    use mod1
    implicit none
    real(kind=8) intent(in) :: in1,in2
    real(kind=8) intent(out) :: out1,out2
    real(kind=8) :: local1

    !should not modify mod1 variables
    !out1,out2 should (usually) be
    !private in the calling parallel region

end subroutine sub1
```

Basic questions:

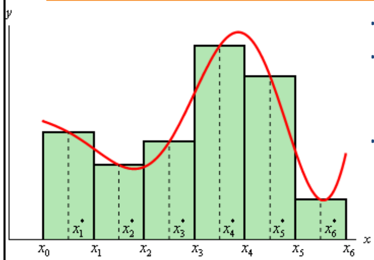
1. Does code give same answer independent of the total number of threads?
2. Is it independent of the *order* in which threads call the subroutine

If yes, the subroutine is *thread-safe*

Should not include OMP directives in subroutine called from within parallel region

Imperial College London

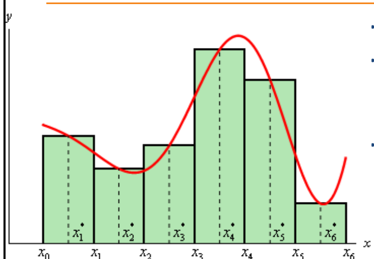
Example: computing an integral



- How to parallelize?
- With three processors, can compute areas of two rectangles on each processor
- Not practical for small calculations, but could split 1e7 rectangles across, say, 10 processors

Imperial College London

Example: computing an integral



- How to parallelize?
- With three processors, can compute areas of two rectangles on each processor
- Not practical for small calculations, but could split 1e7 rectangles across, say, 10 processors
- This is a simple *reduction* problem

Imperial College London

Example: computing an integral

- Serial version:

```
!loop over intervals computing each interval's contribution to integral
    do i1 = 1,N
        xm = dx*(dble(i1)-0.5d0) !midpoint of interval i1
        call integrand(xm,f)
        sum_i = dx*f
        sum = sum + sum_i !add contribution from interval to total
    end do
```

- Parallel version (*see midpoint_omp.f90)*:

```
    !$OMP parallel do private(xm,f,sum_i),reduction(+:sum)
    do i1 = 1,N
         xm = dx*(dble(i1)-0.5d0) !midpoint of interval i1
        call integrand(xm,f)
        sum_i = dx*f
        sum = sum + sum_i !add contribution from interval to total
    end do
    !$OMP end parallel do
```

Imperial College London

Example: computing an integral

- Is there any actual performance gain?
  - Use system_clock and omp_set_num_threads *(see midpoint_time_omp.f90)*

- N=1000

numThreads = 1 `wall time=   2.30000005E-04`

numThreads = 2 `wall time=   6.97000010E-04`

numThreads = 4 `wall time=   1.09699997E-03`

- Here, parallelization slows down the calculation! Why?
- Recall Amdahl's law, here s > p
- s/p will change as N increases…

Imperial College London

Example: computing an integral

- Is there any actual performance gain?
  - Use system_clock and omp_set_num_threads *(see midpoint_time_omp.f90)*

- N=1e7

numThreads = 1 `wall time=   1.11312997`

numThreads = 2 `wall time=  0.6055430174`

numThreads = 4 `wall time=  0.565499008`

- Now, we see improved performance
- Speedup from two threads = 1.8
- No meaningful gain from four threads – laptop only has two cores

Imperial College London