Chapter 33
OpenMP

The best way to have a good idea is to have a lot of ideas.

Linus Pauling

Aim
The aims of this chapter is to provide a short introduction to OpenMP programming in Fortran.

33.1 Introduction

The main OpenMP site is

```
http://www.openmp.org/
```

and this has details about the various specifications

```
http://www.openmp.org/specifications/
```

We recommend downloading the documentation if you are going to do OpenMP programming. You should visit

```
http://www.openmp.org/resources/openmp-compilers/
```

to see an up to date list of what compilers support the OpenMP specification, and at what level.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*,
https://doi.org/10.1007/978-3-319-75502-1_33

The OpenMP site has a range of resources available, check out

```
http://www.openmp.org/resources/
```

for more information.

We've run the examples in this chapter with one or more of the following compilers

- Cray
- gfortran
- Intel
- Nag

33.2 OpenMP Memory Model

OpenMP is a shared memory programming model. It has several features including

- All threads have access to the same shared memory
- Data can be shared or private
- Data transfer is transparent to the programmer
- Synchronization takes place and is generally implicit

We will look at a small number of examples to highlight some of the key features. We provide a brief coverage of some of the OpenMP glossary to provide a basic background to OpenMP.

- Threading Concepts
  - Thread - An execution entity with a stack and associated static memory, called thread private memory.
  - OpenMP thread - A thread that is managed by the OpenMP run time system.
  - Thread-safe routine - A routine that performs the intended function even when executed concurrently (by more than one thread).
- OpenMP language terminology
  - Structured block - For Fortran, a block of executable statements with a single entry at the top and a single exit at the bottom.
  - Loop directive - An OpenMP executable directive whose associated user code must be a loop that is a structured block. For Fortran, only the do directive and the optional end do directive are loop directives.
  - Master thread - The thread that encounters a parallel construct, creates a team, generates a set of tasks, then executes one of those tasks as thread number 0.
  - Work sharing construct - A construct that defines units of work, each of which is executed exactly once by one of the threads in the team executing the construct. For Fortran, work sharing constructs are do, sections, single and work share.

  - Barrier - A point in the execution of a program encountered by a team of threads, beyond which no thread in the team may execute until all threads in the team have reached the barrier and all explicit tasks generated by the team have executed to completion.

- Data Terminology
  - Variable - A named data object, whose value can be defined and redefined during the execution of a program. Only an object that is not part of another object is considered a variable. For example, array elements, structure components, array sections and substrings are not considered variables.
  - Private variable - With respect to a given set of task regions that bind to the same parallel region, a variable whose name provides access to a different block of storage for each task region.
  - Shared variable - With respect to a given set of task regions that bind to the same parallel region, a variable whose name provides access to the same block of storage for each task region.

- Execution Model
  - The OpenMP API uses the fork-join model of parallel execution. Multiple threads of execution perform tasks defined implicitly or explicitly by OpenMP directives. OpenMP is intended to support programs that will execute correctly both as parallel programs (multiple threads of execution and a full OpenMP support library) and as sequential programs (directives ignored and a simple OpenMP stubs library).

The above coverage should be enough to get started with OpenMP and understand the examples that follow.

33.3 Example 1: Hello World

This is the classic hello world program.

```
program ch3301
  use omp_lib
  implicit none
  integer :: nthreads
  integer :: thread_number
  integer :: i

  nthreads = omp_get_max_threads()
  print *, ' Number of threads = ', nthreads
! $omp parallel do
```

```
    do i = 1, nthreads
      print *, ' Hello from thread ', &
        omp_get_thread_num()
    end do
  ! $omp end parallel do
  end program ch3301
```

Let us go through the program one statement at a time.

```
use omp_lib
```

This use statement makes available the OpenMP environment. OpenMP statements are treated as comments without this statement.

```
  nthreads = omp_get_max_threads()
  print *, ' Number of threads = ', nthreads
```

The first statement sets the variable `nthread` to the value returned by the OpenMP function `omp_get_max_threads()`. We then print out this value.

```
!$omp parallel do
```

OpenMP directives in Fortran start with the comment character (!), followed by a $ symbol and the characters omp. We use this form as it is works with both free format and fixed format Fortran source code.

The parallel do words indicate that the code that follows is a parallel region construct. In this case a `do` loop. Here is a small table listing some of the OpenMP directives.

```
               Parallel region construct

!$omp parallel [clause]
structured block
!$omp end parallel

               Work sharing constructs

!$omp do [clause] ...
do loop
!$omp end parallel
!$omp sections [clause] ...
[!$omp section
    structured block ] ...
```

```
!$omp end sections [nowait]
!$omp single [clause]
structured block
!$omp end single [nowait]

               Combined parallel work
               sharing constructs

!$omp parallel do [clause]
structured block
!$omp end parallel do
!$omp parallel sections [clause] ...
[!$omp section
structured block ] ...
!$omp end parallel sections

               Synchronisation constructs

!$omp master
structured block
!$omp end master
!$omp critical [(name)]
structured block
!$omp end critical [(name)]
!$omp barrier
$omp atomic
expression list
!$omp flush
!$omp ordered
structured block
!$omp end ordered

               Data environment

!$omp threadprivate (/c1/,/c2/)
```

We next have the parallel do.

```
  do i = 1, nthreads
    print *, ' Hello from thread ', &
      omp_get_thread_num()
  end do
```

This loop prints out a message from each thread showing the thread number.

```
!$omp end parallel do
```

This marks the end of the OpenMP parallel loop.

So at the start of the loop the OpenMP run time system does a fork and creates multiple threads. At the end of the loop we have a join operation and we are back to one thread of execution.

Here is the output from the Intel compiler on an Intel i7 system.

```
  Number of threads =            8
  Hello from thread            0
  Hello from thread            4
  Hello from thread            2
  Hello from thread            3
  Hello from thread            1
  Hello from thread            7
  Hello from thread            6
  Hello from thread            5
```

These Intel systems have four real cores and each core supports hyper threading in Intel terminology. So the OpenMP system sees eight threads.

Here is the output from the gfortran compiler on the same system.

```
  Number of threads =            8
  Hello from thread            1
  Hello from thread            3
  Hello from thread            2
  Hello from thread            4
  Hello from thread            5
  Hello from thread            6
  Hello from thread            0
  Hello from thread            7
```

The output is very similar, as one would expect.

33.4 Example 2: Hello World Using Default Variable Data Scoping

This is a simple variation on the first example. At first sight it appears to be identical in effect to example one.

```
program ch3302
  use omp_lib
```

```
    implicit none
    integer :: nthreads
    integer :: thread_number
    integer :: i

    nthreads = omp_get_max_threads()
    print *, ' Number of threads = ', nthreads
  !$omp parallel do
    do i = 1, nthreads
      thread_number = omp_get_thread_num()
      print *, ' Hello from thread ', &
        thread_number
    end do
  !$omp end parallel do end program ch3302
```

However we have introduced a variable `thread_number` and are using the OpenMP default data scoping rules, i.e. we have said nothing. Here is the output from the Intel compiler.

```
    Number of threads =              8
    Hello from thread            4
    Hello from thread            5
    Hello from thread            0
    Hello from thread            1
    Hello from thread            2
    Hello from thread            3
    Hello from thread            7
    Hello from thread            6
```

We appear to have a working program. Here is the output from the gfortran compiler.

```
  $ ./a.exe
    Number of threads =              8
    Hello from thread            6
    Hello from thread            7
    Hello from thread            7
    Hello from thread            7
    Hello from thread            7
    Hello from thread            7
    Hello from thread            7
    Hello from thread            7
```

Now something appears to be not quite right! The default variable scoping rules mean that the variable `thread_number` is available to all threads - in OpenMP terminology it is shared. The opposite of shared is private and each thread has their own copy. Example 3 corrects this problem.

33.5 Example 3: Hello World with Private `thread_number` variable

```
program ch3303
  use omp_lib
  implicit none
  integer :: nthreads
  integer :: thread_number
  integer :: i

  nthreads = omp_get_max_threads()
  print *, ' Number of threads = ', nthreads
!$omp parallel do private(thread_number)
  do i = 1, nthreads
    thread_number = omp_get_thread_num()
    print *, ' Hello from thread ', &
      thread_number
  end do
!$omp end parallel do
  end program ch3303
```

Here is the output from the gfortran compiler.

```
$ ./a.exe
  Number of threads =              8
  Hello from thread            2
  Hello from thread            1
  Hello from thread            4
  Hello from thread            3
  Hello from thread            0
  Hello from thread            6
  Hello from thread            5
  Hello from thread            7
```

Care must be taken with variables in OpenMP to ensure they have the correct data scoping state.

33.6 Example 4: Parallel Solution for pi Calculation

This is an OpenMP parallel implementation of the integration problem (Example 3) from the previous chapter. You should compare it with the MPI solution - Example 4 in the last chapter.

```
include 'precision_module.f90'

include 'timing_module.f90'

program ch3304
  use precision_module
  use timing_module
  use omp_lib
  implicit none
  real (dp) :: fortran_internal_pi
  real (dp) :: partial_pi
  real (dp) :: openmp_pi
  real (dp) :: width
  real (dp) :: x
  integer :: nthreads
  integer :: i
  integer :: j
  integer :: k
  integer :: n

  nthreads = omp_get_max_threads()
  fortran_internal_pi = 4.0_dp*atan(1.0_dp)
  print *, ' Maximum number of threads is ', &
    nthreads
  do k = 1, nthreads
    call start_timing()
    n = 100000
    call omp_set_num_threads(k)
    print *, ' Number of threads = ', k
    do j = 1, 5
      width = 1.0_dp/n
      partial_pi = 0.0_dp
!$omp parallel do private(x) &
!$omp shared(width) reduction(+:partial_pi)
      do i = 1, n
        x = width*(real(i,dp)-0.5_dp)
        partial_pi = partial_pi + f(x)
      end do
```

```
!$omp end parallel do
      openmp_pi = width*partial_pi
      print 100, n, time_difference()
      print 110, openmp_pi, abs(openmp_pi- &
        fortran_internal_pi)
      n = n*10
    end do
  end do
100 format (' N intervals = ', i12, ' time =', &
    f8.3)
110 format (' openmp_pi = ', f20.16, /, &
    'difference = ', f20.16)
  call end_timing()
  stop

contains

  real (dp) function f(x)
    implicit none
    real (dp), intent (in) :: x

    f = 4.0_dp/(1.0_dp+x*x)
  end function f

end program ch3304
```

Here is the output from the Intel compiler.

```
 Maximum number of threads is            8
..
  Number of threads =            1
 N intervals =       100000 time =   0.004
 openmp_pi =   3.1415926535981167
difference =   0.0000000000083236
 N intervals =      1000000 time =   0.012
 openmp_pi =   3.1415926535899033
difference =   0.0000000000001101
 N intervals =     10000000 time =   0.051
 openmp_pi =   3.1415926535896861
difference =   0.0000000000001070
 N intervals =    100000000 time =   0.449
 openmp_pi =   3.1415926535902168
difference =   0.0000000000004237
 N intervals =   1000000000 time =   4.398
 openmp_pi =   3.1415926535897682
```

```
difference =   0.0000000000000249
..
  Number of threads =           2
 N intervals =      100000 time =   0.000
 openmp_pi =   3.1415926535981260
difference =   0.0000000000083329
 N intervals =     1000000 time =   0.000
 openmp_pi =   3.1415926535898624
difference =   0.0000000000000693
 N intervals =    10000000 time =   0.020
 openmp_pi =   3.1415926535897829
difference =   0.0000000000000102
 N intervals =   100000000 time =   0.219
 openmp_pi =   3.1415926535898926
difference =   0.0000000000000995
 N intervals =  1000000000 time =   2.195
 openmp_pi =   3.1415926535897380
difference =   0.0000000000000551
..
  Number of threads =           4
 N intervals =      100000 time =   0.004
 openmp_pi =   3.1415926535981287
difference =   0.0000000000083356
 N intervals =     1000000 time =   0.004
 openmp_pi =   3.1415926535898726
difference =   0.0000000000000795
 N intervals =    10000000 time =   0.027
 openmp_pi =   3.1415926535898153
difference =   0.0000000000000222
 N intervals =   100000000 time =   0.137
 openmp_pi =   3.1415926535898038
difference =   0.0000000000000107
 N intervals =  1000000000 time =   1.781
 openmp_pi =   3.1415926535898544
difference =   0.0000000000000613
..
  Number of threads =           8
 N intervals =      100000 time =   0.000
 openmp_pi =   3.1415926535981278
difference =   0.0000000000083347
 N intervals =     1000000 time =   0.004
 openmp_pi =   3.1415926535898784
difference =   0.0000000000000853
 N intervals =    10000000 time =   0.016
 openmp_pi =   3.1415926535897962
```

```
difference =   0.0000000000000031
 N intervals =    100000000 time =   0.113
 openmp_pi =   3.1415926535898162
difference =   0.0000000000000231
 N intervals =   1000000000 time =   1.137
 openmp_pi =   3.1415926535898824
difference =   0.0000000000000893
```

We have similar timing improvements to the MPI solutions.

33.7 Example 5: Comparing the Timing of Whole Array Syntax, Simple Do Loops, Do Concurrent and an OpenMP Solution

The chapter on data structuring introduced the `do concurrent` statement. In the example we solve a summation problem using the following four methods:

- whole array syntax
- simple do loop
- do concurrent loop
- OpenMP parallel loop

Here is the program.

```
include 'timing_module.f90'
include 'precision_module.f90'

program ch3305

use timing_module
use precision_module
use omp_lib

implicit none

integer , parameter :: n=10000000
integer , parameter ::loop_count=10
integer , parameter :: n_types=4
integer :: i
integer :: j
integer :: nthreads

real (dp) , allocatable , dimension(:) :: x
real (dp) , allocatable, dimension(:)  :: y
real (dp) , allocatable , dimension(:) :: z

 real , dimension(n_types,loop_count) :: &
  timing_details = 0.0
real , dimension(n_types) :: t_sum  = 0.0
real , dimension(n_types) :: t_average = 0.0
real :: reset = 0.0

character (15)  , dimension(n_types) :: &
  heading_1 = &
    [ ' Whole array   ' , &
      ' Do loop       ' , &
      ' Do concurrent ' , &
      ' openmp        ' ]

  call start_timing()
  print *,' '

  nthreads = omp_get_max_threads()
  open(unit=20,file='ch3305.dat')
  print 10,nthreads
  10 format(' Nthreads = ',i3)
  allocate (x(n))
  allocate (y(n))
  allocate (z(n))

  call random_number(x)
  call random_number(y)
  z=0.0_dp
  print 20,time_difference()
  20 format(' Initialise time = ',f6.3)
  write(20,30) x(1),y(1),z(1)
  30 format(3(2x,f6.3))
  print  *, ' '

  do j=1,loop_count

    print 40,j
    40 format(' Iteration = ',i3)
!
!  Whole array syntax
!
    z=x+y
    timing_details(1,j) = time_difference()
```

```
      write(20,30) x(1),y(1),z(1)
      z = 0.0_dp
      reset = time_difference()
  !
  ! Simple traditional do loop
  !
      do i=1,n
        z(i)=x(i)+y(i)
      end do
      timing_details(2,j) = time_difference()
      z = 0.0_dp
      reset = time_difference()
  !
  ! do concurrent loop
  !
      do concurrent (i=1:n)
        z(i)=x(i)+y(i)
      end do
      timing_details(3,j) = time_difference()
      write(20,30) x(1),y(1),z(1)
      z = 0.0_dp
      reset = time_difference()
  !
  ! OpenMP parallel loop
  !
    !$omp parallel do
      do i=1,n
        z(i)=x(i)+y(i)
      end do
    !$omp end parallel do
      timing_details(4,j) = time_difference()
      write(20,30) x(1),y(1),z(1)
      z = 0.0_dp
      reset = time_difference()

    end do
    close(20)
    print 50
    50 format(15x,70x,'    Sum      Average')

    do i=1,n_types
      t_sum(i) = &
        sum(timing_details(i,1:loop_count))
      t_average(i) = t_sum(i)/loop_count
      print 60,heading_1(i) , &
```

```
        timing_details(i,1:loop_count),&
        t_sum(i),t_average(i)
      60 format(a,10(1x,f6.3),2(3x,f6.3))
    end do

    print *,' '
    call end_timing()
  end program ch3305
```

Here are some timing details for three compilers on one system under both Linux and Windows.

	gfortran Linux	Windows	Intel Linux	Windows	Nag Linux	Windows
Whole array	0.019	0.018	0.013	0.015	0.034	0.053
Do loop	0.019	0.019	0.018	0.019	0.020	0.019
Do concurrent	0.019	0.018	0.018	0.020	0.019	0.020
openmp	0.016	0.016	0.012	0.012	0.016	0.016

33.8 Summary

This chapter briefly introduced the essentials of OpenMP programming. We have also seen the timing benefits that OpenMP programming can offer in the solution of the same problem as in the MPI chapter. We finished off by doing a comparison of summation in Fortran using four methods.

33.9 Problem

33.1 Compile and run the examples in this chapter with your compiler and compare the results.