11. OpenMP programming

[1]

[1] M. Ganesh, MATH440/540, SPRING 2018

[2] OpenMP

(http:www.openmp.org; 1997-present;May, 2008 OpenMP3.0)

- **Open Multi-Processing (OpenMP) is a set of Fortran or C/C++ directives that describe parallelism in a code along with a supporting library of procedures and environment variables**
- **These directives and libraries are described by the Application Programming Interface (API)**
- **Thus OpenMP is not a new computer language. OpenMP is an API that works in conjunction with Fortran or C/C++**
- **The directives in OpenMP are instructional notes to any compiler that supports OpenMP**
- **These directives take the form of comments in Fortran (that is, starting with `!` in Fortran 90+) or `# pragma` in C/C++**
- **The type parallelism used in OpenMP is multi-threading**
- **That is, OpenMP API facilitates development of codes that split into multiple simultaneous tasks (threads) for independent execution by sharing resources in multi-core computing environments**

[2] M. Ganesh, MATH440/540, SPRING 2018

- [3]Note that OpenMP is not a set of message passing routines
- OpenMP is an approach for giving directives to the OpenMP supportive compilers to allow parts of the code (such as loops) can be executed in parallel
- It facilitates an easy way of creating multi-threads in a serial code
- OpenMP uses multiple threads to implement fork-join method of parallelism:
  - ⋆ Take a sequential program in Fortran or C/C++
  - ⋆ Put parallel OpenMP directives into the program
  - ⋆ The single thread that forks into multiple threads is called the master thread
  - ⋆ Master thread creates new worker threads as specified in directives
  - ⋆ When threads complete the statements in parallel section, all the worker threads terminate and join the master synchronously, leaving the master thread to proceed with other operations
- Parallelism in OpenMP is added incrementally as the directives are added, making a sequential program evolve into a parallel program

[3]M. Ganesh, MATH440/540, SPRING 2018

- [4]A typical use of OpenMP programming is to parallelize loops:

Example (Fortran - serial version):

```
!serial code
.....................
w = d
do i = 1, 16000
   c(i) = a(i) + w*b(i)
end do
w = w+2
.....................
```

- In the above code, clearly the result from one loop of iteration does not depend on the result of any other iteration. The statements above and below the loop are serial in nature
- We may use the OpenMP `PARALLEL DO` directive to parallelize only this loop to facilitate a OpenMP supportive compiler to create multi-threads for faster execution of this code

[4]M. Ganesh, MATH440/540, SPRING 2018

[5]Example (Fortran - OpenMP multi-threaded parallel version):

```
! parallel  code  in  OpenMP
.....................
w = d
!$omp parallel do
     do i=1,16000
          c(i) = a(i) + w*b(i)
     end do
!$omp end parallel do
w = w+2
.....................
```

- The OpenMP directive `parallel do` directs the compiler that after executing the line `w = d` in serial (with a single master thread), the loop immediately following the directive should be executed in parallel by the forking multiple threads from the master thread
- The second OpenMP directive `end parallel do` directs the compiler to end the parallel section of the code and join the master so that the master can execute the statement `w = w+2` in serial

[5]M. Ganesh, MATH440/540, SPRING 2018

- [6]**The total number of multiple threads, say 8 on MIO/BlueM, are decided at the time of execution by the command** `export OMP_NUM_THREADS=8` **(bash) or** `setenv OMP_NUM_THREADS 8` **(csh)**
- **In this case the master thread creates additional seven slave/worker threads, forming a team of eight threads**
- **This OpenMP team (master + workers) of threads divide the iteration jobs in the loop among themselves**
- **The OpenMP directive** `end parallel do` **at the end of the** `parallel do` **has an implicit barrier (synchronization).**
- **Hence each thread waits for the remaining threads in the team to finish their iterations**
- **Once all the threads in the team have finished and all iterations have been executed, the barrier condition is complete and all threads are released from the barrier**
- **At this point, the slave threads disappear and the master thread resumes execution of the statement** `w = w + 2`

[6]M. Ganesh, MATH440/540, SPRING 2018

- [7]**The specific mapping of threads (execution vehicle) depends on the OpenMP implementation in the compiler**
- **Each thread is assigned a unique and distinct set of iterations to execute**
- **OpenMP does not specify how the iterations are to be divided among the team of threads and the choice is left to the OpenMP implementation in the compiler**
- **However, OpenMP provides additional attributes for the user to specify the load distribution among the team of threads. (We will discuss specific details later.)**
- **Compilation of the above code (with *simple static* schedule) using a Fortran compiler and executing the code (after, say, the command `export OMP_NUM_THREADS=8`) shall lead to creation of the following team of eight threads:**

[7]M. Ganesh, MATH440/540, SPRING 2018

```
    Thread : 0
do i=1,2000
   c(i) = a(i) + w*b(i)
end do
```

```
  Thread : 1
do i=2001,4000
  c(i) = a(i) + w*b(i)
end do
```

```
    Thread : 2
do i=4001,6000
   c(i) = a(i) + w*b(i)
end do
```

```
  Thread : 3
do i=6001, 8000
  c(i) = a(i) + w*b(i)
end do
```

```
    Thread : 4
do i=8001,10000
   c(i) = a(i) + w*b(i)
end do
```

```
  Thread : 5
do i=10001,12000
  c(i) = a(i) + w*b(i)
end do
```

```
    Thread : 6
do i=12001,14000
   c(i) = a(i) + w*b(i)
end do
```

```
  Thread : 7
do i=14001,16000
  c(i) = a(i) + w*b(i)
end do
```

[8]**Example (C/C++ - OpenMP multi-threaded parallel version):**

```
.....................
w = d;
 #pragma omp parallel for
{
   for ( i = 0; i < 16000; i++ )
          c[i] = a[i] + w*b[i];
}
w = w+2;
.....................
```

- **From the above example, it is clear that each thread should have access to the arrays a, b, c as well as the variable w**

- **The shared memory model within OpenMP prescribes that multiple OpenMP threads execute within the same shared space programming**

[8]M. Ganesh, MATH440/540, SPRING 2018

- [9]In shared memory model:
  - ⋆ All threads have access to the same, globally shared, memory
  - ⋆ Data can be shared or private
  - ⋆ Shared data is accessible by all threads
  - ⋆ Private data can be accessed only by the threads that owns it
  - ⋆ Data transfer is transparent to the programmer
  - ⋆ Synchronization takes place, but it is mostly implicit
  - ⋆ Variables have a label attached to them
  - ⋆ Variables labeled `private` are visible just one thread
  - ⋆ Variables labeled `shared` are visible to all threads
- In the last example the variables `a, b, c, w` are shared variables
- The variable `i` is a private variable
- The loop index variables in Fortran are treated as having private scope by defaults, unless specified otherwise. (In C/C++ this is not the case especially for nested loops. The loop index variables in sequential/nested loops must be declared private in C/C++.)

[9]M. Ganesh, MATH440/540, SPRING 2018

- [10]**The OpenMP parallel programming is useful mainly on SMP machines because of working mainly on thread levels**

- **From the examples, it is clear that programs that include OpenMP directives can also be compiled using any Fortran, C/C++ compiler (without OpenMP support) if we are interested in running the code in serial**

- **Current GNU, INTEL, PGI compilers provide OpenMP support**

- **GNU compilers starting from version 4.2 (GCC and Gfortran) support OpenMP (v2.5)**

- **To compile (with an option to support OpenMP) your parallel programs (with OpenMP directives) using the GNU compilers (4.2+) use the additional flag `-fopenmp` and run the resulting executable file, say, `a.out` as usual (That is, use `./a.out`.)**

[10]M. Ganesh, MATH440/540, SPRING 2018

- [11]**The current version of GNU compilers (gfortran, gcc, g++) on Sayers Lab machines are 7.3.1 and hence you may use Sayers Lab machines for your codes with OpenMP directives**

- **To compile your parallel programs in OpenMP on Sayers Lab, using instead the Portland Group compilers (pgf77, pgf90, pgcc), with an option to support OpenMP, use the additional flag `-mp`**

- **You may also use MIO/BlueM to compile and execute your OpenMP codes using Intel and Portland Group compilers**
- **To compile (with an option to support OpenMP) your parallel programs in OpenMP on MIO/BlueM using the Intel compilers (ifort or icc) use the additional flag `-openmp`**
- **To submit your job on MIO/BlueM with executable, say `a.out` containing OpenMP directives, with one node and eight cores so that OpenMP directives in your code can create an OpenMP team of eight threads, use the following SLURM script. (Recall that OpenMP is mainly useful for shared memory machines.)**

[11]M. Ganesh, MATH440/540, SPRING 2018

```
#!/bin/bash -x
#SBATCH --job-name="my_openmp_project"
#comment      = "Test oopenmp..."
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH --ntasks=8
#SBATCH --exclusive
#SBATCH --export=ALL
#SBATCH --time=hh:mm:ss
#SBATCH -o project.out
#SBATCH -e project.err
cd $SLURM_SUBMIT_DIR
JOBID=‘echo $SLURM_JOBID’

export OMP_NUM_THREADS=8
./a.out
echo "job has finished"
```

- [12]**In free-form Fortran (F90+) code, a line that begins with the following prefix keyword (known as the *sentinel*):**
  `!$omp ...`
  is treated as an OpenMP directive
- **The free-form sentinel may begin in any column as long as it appears as a single word and is preceded only by white space**
- **In fixed-form Fortran, the following sentinels are allowed**
  `c$omp ...`
  `*$omp ...`
- **In a fixed-form, there should be either a space or a zero in the sixth column for an OpenMP supportive compiler treat the associate line as a directive**
- **In C/C++, OpenMP pragmas follow the syntax**
  `#pragma omp ...`
- **Such well defined prefix for OpenMP directives facilitate compilers without OpenMP support to ignore these commands and also allow the user to have an option of compiling with or without OpenMP flag in any compiler**

[12]M. Ganesh, MATH440/540, SPRING 2018

- [13]**The library routine** `omp_get_num_threads` **returns the number of threads in the OpenMP team executing a parallel region**
- **The library routine** `omp_get_thread_num` **returns the thread number**

Example (Fortran) :

```
program thread_number_test
implicit none
integer :: omp_get_thread_num, omp_get_num_threads , iam, tot
!$omp parallel
    iam= omp_get_thread_num()
    tot= omp_get_num_threads()
    write(*,*) &
    "I am thread  ", iam,   " in an OpenMP team of ", tot, " threads"
!$omp end parallel
end program thread_number_test
```

- **Save the above code as** `thread_number_test.f90` **and compile on MIO/BlueM, for example, using** `ifort -openmp -o test_exe thread_number_test.f90` **and run the executable file** `test_exe`

[13]M. Ganesh, MATH440/540, SPRING 2018

- **Output from one run of `test_exe` (with `export OMP_NUM_THREADS=8`)**

```
I am thread          0  in an OpenMP team of          8  threads
I am thread          1  in an OpenMP team of          8  threads
I am thread          2  in an OpenMP team of          8  threads
I am thread          4  in an OpenMP team of          8  threads
I am thread          3  in an OpenMP team of          8  threads
I am thread          3  in an OpenMP team of          8  threads
I am thread          5  in an OpenMP team of          8  threads
I am thread          5  in an OpenMP team of          8  threads
```

- **Output from another run of `test_exe` (with `export OMP_NUM_THREADS=8`):**

```
I am thread          0  in an OpenMP team of          8  threads
I am thread          1  in an OpenMP team of          8  threads
I am thread          3  in an OpenMP team of          8  threads
I am thread          2  in an OpenMP team of          8  threads
I am thread          4  in an OpenMP team of          8  threads
I am thread          7  in an OpenMP team of          8  threads
I am thread          7  in an OpenMP team of          8  threads
I am thread          7  in an OpenMP team of          8  threads
```

- **Why? Is it possible to get the code to print all the eight thread numbers `0, ..., 7` in the team of `8` threads?**
- **The main reason for not obtaining the desired output of all the threads is due to the default scope in OpenMP for the variable `iam`**
- **The scope of the variable `iam` in the above code is `shared`**
- **Hence `iam` is visible to all threads and the values are updated in the global shared memory location of `iam`**
- **One of the approaches to avoid the problem in the above code is to declare the variable `iam` as private to each thread:**

Example (Fortran):

```
program private_thread_number_test
implicit none
integer :: omp_get_thread_num, omp_get_num_threads, iam, tot
!$omp parallel private(iam)
    iam= omp_get_thread_num()
    tot= omp_get_num_threads()
    write(*,*) &
    "I am thread  ", iam,   " in an OpenMP team of ", tot, " threads"
!$omp end parallel
end program private_thread_number_test
```

- **Let the executable from the code be `priv_test_exe`**
- **Output from one of the runs of `priv_test_exe`**

```
I am thread          0  in an OpenMP team of          8  threads
I am thread          1  in an OpenMP team of          8  threads
I am thread          2  in an OpenMP team of          8  threads
I am thread          7  in an OpenMP team of          8  threads
I am thread          3  in an OpenMP team of          8  threads
I am thread          6  in an OpenMP team of          8  threads
I am thread          5  in an OpenMP team of          8  threads
I am thread          4  in an OpenMP team of          8  threads
```

- **Output from another run of `priv_test_exe`**

```
I am thread          0  in an OpenMP team of          8  threads
I am thread          1  in an OpenMP team of          8  threads
I am thread          4  in an OpenMP team of          8  threads
I am thread          3  in an OpenMP team of          8  threads
I am thread          2  in an OpenMP team of          8  threads
I am thread          7  in an OpenMP team of          8  threads
I am thread          5  in an OpenMP team of          8  threads
I am thread          6  in an OpenMP team of          8  threads
```

- **[14]Another approach is to restrict only one thread to be in a parallel region at a time, using the command `!$omp critical`**

Example:

```
program critical_thread_number_test
implicit none
integer :: omp_get_thread_num, omp_get_num_threads, iam, tot
!$omp parallel
!$omp critical
    iam= omp_get_thread_num()
    tot= omp_get_num_threads()
    write(*,*) &
    "I am thread  ", iam,   " in an OpenMP team of ", tot, " threads"
!$omp end critical
!$omp end parallel
end program critical_thread_number_test
```

- **Let the executable from the code be `crit_test_exe`**

[14]M. Ganesh, MATH440/540, SPRING 2018

- **[15]Output from one of the runs of `crit_test_exe`**

```
I am thread          0  in an OpenMP team of          8  threads
I am thread          1  in an OpenMP team of          8  threads
I am thread          2  in an OpenMP team of          8  threads
I am thread          3  in an OpenMP team of          8  threads
I am thread          5  in an OpenMP team of          8  threads
I am thread          4  in an OpenMP team of          8  threads
I am thread          6  in an OpenMP team of          8  threads
I am thread          7  in an OpenMP team of          8  threads
```

- **Output from another run of `crit_test_exe`**

```
I am thread          0  in an OpenMP team of          8  threads
I am thread          1  in an OpenMP team of          8  threads
I am thread          4  in an OpenMP team of          8  threads
I am thread          5  in an OpenMP team of          8  threads
I am thread          3  in an OpenMP team of          8  threads
I am thread          2  in an OpenMP team of          8  threads
I am thread          7  in an OpenMP team of          8  threads
I am thread          6  in an OpenMP team of          8  threads
```

[15]M. Ganesh, MATH440/540, SPRING 2018

[16]**Example (C)**:

```
#include <stdio.h>
#include <omp.h>
int main() {int  iam, tot;
#pragma omp parallel
{iam= omp_get_thread_num();
 tot= omp_get_num_threads();
 printf( "I am thread  %d in an OpenMP team of %d threads\n", iam, tot);}
}
```

Example (C++):

```
using namespace std;
#include <iostream>
#include <omp.h>
int main() { int  iam, tot;
#pragma omp parallel
{iam= omp_get_thread_num();
tot= omp_get_num_threads();
cout << "\n I am thread " << iam << " in an OpenMP team of "
       <<  tot << " threads\n" ; }
}
```

[16]M. Ganesh, MATH440/540, SPRING 2018

[17]**Example (C)**:

```
#include <stdio.h>
#include <omp.h>
int main() {int  iam, tot;
#pragma omp parallel private(iam)
{iam= omp_get_thread_num();
 tot= omp_get_num_threads();
 printf( "I am thread  %d in an OpenMP team of %d threads\n", iam, tot);}
}
```

Example (C++):

```
using namespace std;
#include <iostream>
#include <omp.h>
int main() { int  iam, tot;
#pragma omp parallel  private(iam)
{iam= omp_get_thread_num();
tot= omp_get_num_threads();
cout << "\n I am thread " << iam << " in an OpenMP team of "
       <<  tot << " threads\n" ; }
```

[17]M. Ganesh, MATH440/540, SPRING 2018

Example (C):

```
#include <stdio.h>
#include <omp.h>
int main() {int  iam, tot;
#pragma omp parallel
#pragma omp critical
{iam= omp_get_thread_num();
 tot= omp_get_num_threads();
 printf( "I am thread  %d in an OpenMP team of %d threads\n", iam, tot);}
}
```

Example (C++):

```
using namespace std;
#include <iostream>
#include <omp.h>
int main() { int  iam, tot;
#pragma omp parallel
#pragma omp critical
{iam= omp_get_thread_num();
tot= omp_get_num_threads();
cout << "\n I am thread " << iam << " in an OpenMP team of "
       <<  tot << " threads\n" ; }
}
```

- [18]**OpenMP Fortran90+ (case insensitive) directives are of the form:**
  `!$omp directive [clause[,] [clause ...]`
- **OpenMP C/C++ (case sensitive) directives are of the form:**
  `#pragma omp directive [clause [clause] ...]`
- **Throughout, the notation `[...]` is used for optional information**
- **A typical example of OpenMP directive in Fortran is `parallel do` and in C/C++ is `parallel for`**
- **There are various types of `clauses`:**
  - ⋆ ***Scoping clauses*. Most important OpenMP scoping clauses are:** `private, shared, reduction, firstprivate, lastprivate`
  - ⋆ ***Schedule clause*. The schedule clause controls how iterations of the parallel loop are distributed across the team of parallel threads**
  - ⋆ ***If clause*. The if clause controls whether the loop should be executed in parallel or serially like an ordinary loop, based on a user-defined runtime test**
  - ⋆ ***Ordered clause*. This clause specifies ordering/synchronization**
  - ⋆ ***Copyin clause*.**
    Initializes certain kinds of private variables called `threadprivate`

[18]M. Ganesh, MATH440/540, SPRING 2018

- [19]**In OpenMP Fortran programming, each `parallel do` directive must be followed by a `do` loop of the form:**
  `do index = start, end, [, stride]`
- **In OpenMP C/C++ programming, `parallel for` directive must be followed by a `for` loop in canonical form (so that the trip-count can be precisely determined):**
  `for(i = start, i < end; incr_expr).`
  (Here `<` may be replaced with `<=`, `>`, `>=` and the forms of `incr_expr` are `i++`, `++i`, `i--`, `--i`, `i+=`, `i-=`, `...`)
- **In addition to requiring a computable trip-count, the `parallel do` (or `parallel for`) directive requires that the program complete all iterations of the loop**
- **Thus the program cannot use any control flow constructs that exit the loop before all iterations have been completed. Hence for OpenMP programming in Fortran `exit`, `goto` and in C/C++ `break`, `goto` to exit out of the loop should be avoided**
- **However, `cycle` in Fortran and `continue` in C/C++ and also `exit`, `break`, `goto` within the loop are allowed**

[19]M. Ganesh, MATH440/540, SPRING 2018

- [20]**Other directives such as `parallel, sections, single, master, critical, ordered` share restrictions similar to that in `parallel do`**
- **When a body of loop contains another loop, the second loop is nested inside the first loop**
- **In the first (left) example below , the outer loop is parallelized and the inner loop is executed in serial by all threads created for the outer loop and vice-versa for the second example**

Example:

```
.....................
!$omp parallel do
do j = 1, N
   a(0,j) = 0.0d0
   do i = 1, M
      a(0,j) = a(0,j) + a(i,j)
   end do
end do
.....................
```

Example:

```
.....................
do j = 1, N
   !$omp parallel do
      do i = 1, M
      a(i,j) = (a(i-1,j-1)+a(i,j-1) &
                a(i+1,j-1))/(3.0d0)
     end do
end do
     .....................
```

[20]M. Ganesh, MATH440/540, SPRING 2018

- [21]**The default scoping rules in OpenMP state that if a variable is used within a parallel construct and is not scoped explicitly, then the variable is treated as shared**
- **There are three main exceptions to this rule:**
  - ⋆ **loop index variables;**
  - ⋆ **local variables and value parameters in subroutines, called within a parallel region;**
  - ⋆ **automatic variables (C/C++) within a parallel region**
- **The index variable of a loop, to which a `parallel do` or `parallel for` is applied, is scoped by default as `private`**
- **In addition, only in Fortran, the index variable of a sequential loop that appears within a parallel region is scoped as `private`**
- **In C/C++, the index variables of sequential `for` loops are scoped as shared by default. Hence in C/C++ the index variables of serial loops must explicitly be scoped as `private`**

[21]M. Ganesh, MATH440/540, SPRING 2018

- [22]**When a subroutine is called from within a parallel region, then local variables within the called subroutine are `private` to each thread**
- **However, if any of these variables are marked with the `save` attribute (in Fortran) or as `static` (in C/C++), then these variables are have the `shared` scope.**
- **Identify scope in the following Fortran and C examples:**

```
subroutine caller(a,n)
integer:: n, a(n), i, j, m
m = 3
!$omp parallel do
do i = 1, n
   do j = 1, 5
   call callee(a(i), m, j)
   end do
end do
end subroutine caller
```

```
subroutine callee(x,y,z)
common /com/ c
integer:: x, y, z, c, ii, count
save count

count = count + 1
do ii= 1, z
   x = y + c
end do
end subroutine callee
```

- **In the above example, the variables with `shared` scope are: `a, n, m, x, y, c, count` and with `private` scope are: `i, j, z, ii`**
- **In this example, usage of the default scope for `count` is not safe**

[22]M. Ganesh, MATH440/540, SPRING 2018

Example:

```
void  caller(int a[],  int n)
{
  int i, j, m = 3;

  #pragma omp parallel for
  for(i = 0; i < n; i++) {
      int k = m;

   for(j = 1; j <= 5; j++)
       callee(&a[i], &k, j);
   }
}
extern int c;
```

```
void  callee(int *x,  int *y , int z)
{
   int ii;
   static int count

   count++;

     for(ii = 0; ii < z; i++)
        *x  = *y + c;

 }
```

- **In the above example, the variables with `shared` scope are:**
  `a, n, j, m, *x, c, count`
  and with `private` scope are: `i, k, x, y, *y, z, ii`

- **In this example, usage of the default scope for `j, count` is not safe**

- [23]**The default behavior in OpenMP can be changed using the `default` clause**

- **At most one `default` clause may appear on a parallel region**

- **The `default` clause allowed in Fortran are:** `default (shared)`, `default (private)`, **and** `default (none)`

- **The `default` clause allowed in C/C++ are:** `default (shared)`, **and** `default (none)`

[23]M. Ganesh, MATH440/540, SPRING 2018