Brief Introduction to OpenMP

in Large-Scale and Supercomputing Course

August 15 - 19, 2011

Napsu Karmitsa

Department of Mathematics
University of Turku, Finland

Contents

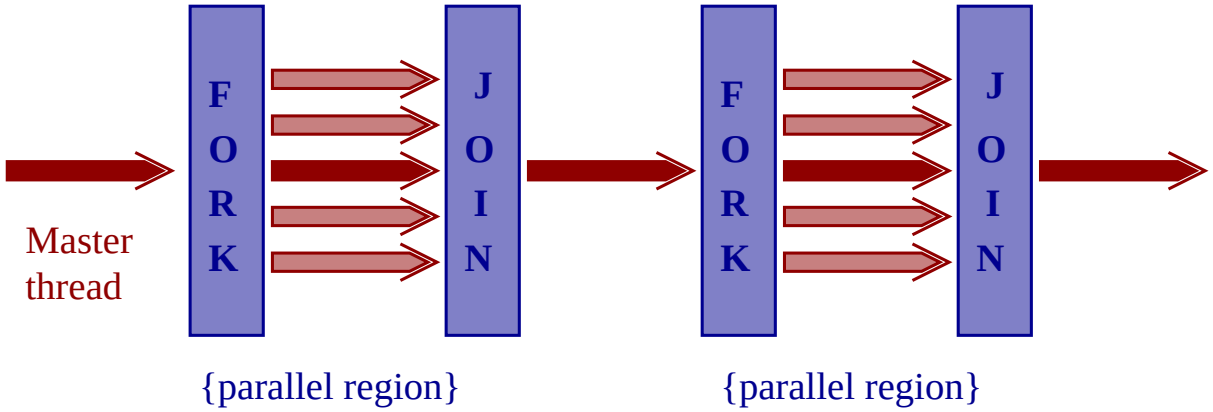
- What is OpenMP?
- OpenMP Programming Model
- OpenMP Directives
- Runtime Library Routines
- Environment Variables

WHAT IS OPENMP?

- OpenMP is an Application Program Interface (API) that may be used to explicitly direct **multi-threaded**, **shared memory parallelism**.
- Comprised of three primary API components:
  - Compiler Directives,
  - Runtime Library Routines,
  - Environment Variables.
- Simple:
  - Need not deal with message passing as MPI does.
- Portable:
  - The API is specified for C/C++ and Fortran,
  - Supported by many compilers, e.g. `gcc` and `gfortran`,
  - Most major platforms have been implemented including Unix/Linux platforms and Windows NT.

OpenMP Programming Model

Shared Memory, Thread Based Parallelism, Fork — Join Model:



- All OpenMP programs begin as a single process: **the master thread**. The master thread executes sequentially until the first **parallel region** construct is encountered.
- **FORK**: the master thread then creates a **team** of parallel threads.
- The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads.
- **JOIN**: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.

OpenMP Programming Model

C − General Code Structure

```
#include <omp.h>

main () {
int var1, var2, var3;

/* Serial code */
    .
    .
    .

/* Beginning of parallel section */
/* Fork a team of treads */
/* Specify variable scoping */

#pragma omp parallel private(var1,var2) shared(var3)
  {
  /* Parallel section executed by all threads */
      .
      .
      .

  /* All treads join master thread */
  }
/* Resume serial code */
    .
    .
    .

}
```

Fortran − General Code Structure

```
PROGRAM omp_prog
INTEGER :: var1, var2, var3

! Serial code
  .
  .
  .

! Beginning of parallel section
! Fork a team of threads
! Specify variable scoping

!$OMP PARALLEL PRIVATE(var1,var2) SHARED(var3)

  ! Parallel section executed by all threads
        .
        .
        .

  ! All treads join master thread

!$OMP END PARALLEL

! Resume serial code
  .
  .
  .

END PROGRAM omp_prog
```

OpenMP Programming Model

- Most OpenMP parallelism is specified through the use of **compiler directives** which are imbedded in C/C++ or Fortran source code.
  - C/C++ uses `#pragma` directives:

```
#pragma omp directive_name [clause, ...]
```

  - Fortran uses structured comments:

```
!$omp directive_name [clause, ...]
```

    Many Fortran directives come in pairs and have the form

```
!$omp directive
   [structured block of code]
!$omp end directive
```

- Compiler directives can usually be arranged such that the same code can be run sequentially.

OpenMP Programming Model

- In addition to compiler directives OpenMP includes some **library routines** and **environmental variables** to control the program behavior.
  - In C/C++ an `omp.h` file needs to be included:

```
#include ⟨ omp.h ⟩
```

  - Fortran uses `omp_lib` module:

```
USE omp_lib
```

- The OpenMP specifications for C and Fortran can be downloaded from

  http://openmp.org/wp/openmp-specifications/

- In order to enable OpenMP the compiler must be given the proper option: e.g. `-fopenmp` in `gfortran` and `gcc`.

Compiler Directives: Parallel

- The fundamental parallel construct in OpenMP is PARALLEL directive.
- Simple example:

```
#include <stdio.h>
#include <omp.h>

main () {

#pragma omp parallel
  printf("Greetings!\n");
}
```

```
PROGRAM omp_prog
  USE omp_lib
  IMPLICIT NONE

  !$OMP PARALLEL
    PRINT*,'Greetings!'
  !$OMP END PARALLEL
END PROGRAM omp_prog
```

```
> gfortran -fopenmp -o oprog omp_prog.f95
> ./oprog
Greetings!
Greetings!
Greetings!
Greetings!
>
```

  - When a thread reaches a PARALLEL directive, it creates a team of threads and becomes the master of the team. Threads are numbered from 0 to N-1. The master is a member of that team and has thread number 0 within that team.
  - Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.
  - There is an implied barrier at the end of a parallel section. Only the master thread continues execution past this point.

Compiler Directives: Parallel

How Many Threads?

- The default number of threads in a parallel region is implementation dependent: Often it is the number of CPUs in the machine.
- Number of threads can be set explicitly by

1. setting the `NUM_THREADS` clause,

   Fortran: `!$OMP PARALLEL [num_threads(3)]`
   C: `#pragma omp parallel [num_threads(3)]`

2. using the `omp_set_num_threads` library function,

   Fortran: `CALL omp_set_num_threads(3)`
   C: `omp_set_num_threads(3);`

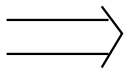
3. setting the `OMP_NUM_THREADS` environment variable:

   ```
   > setenv OMP_NUM_THREADS 3
   ```

Compiler Directives: Work Sharing Constructs

- **Work sharing constructs** are used to specify how to assign independent work to one or all of the threads.
- Work sharing constructs do not create threads themselves, so they must be inside a parallel region in order to execute parallel.
- OpenMP allows to use shortcuts so that a parallel work sharing directives can be specified in one line:

```
!$OMP PARALLEL
  !$OMP work_sharing_directive
     .
     .
     .
  !$OMP END work_sharing_directive
!$OMP END PARALLEL
```



```
!$OMP PARALLEL work_sharing_directive
   .
   .
   .
!$OMP END PARALLEL work_sharing_directive
```

Compiler Directives: Work Sharing Constructs

- Work sharing directives are
  - `DO`/`for`: used to split up loop iterations among the threads, also called loop constructs.
  - `sections`: assigning consecutive but independent code blocks to different threads
  - `single`: specifying a code block that is executed by only one thread, a barrier is implied in the end. May be useful when dealing with sections of the code that are not thread safe (such as I/O),
  - `master`: similar to single, but the code block will be executed by the master thread only and no barrier implied in the end.
  - `critical`: specifies a region of the code that must be executed by only one thread at a time.
  - `workshare`: divides the execution of the enclosed structured block into separate units of work, each of which is executed only once. Available only in Fortran.

Compiler Directives: Work Sharing Constructs

Sections Example

```
#pragma omp parallel
  {
  #pragma omp sections nowait
    {
    #pragma omp section
    for (i=0; i<n; i++)
      c[i] = a[i] + b[i];

    #pragma omp section
    for (i=0; i<n; i++)
      d[i] = a[i] * b[i];
    }
  }
```

```
!$OMP PARALLEL
!$OMP SECTIONS
!$OMP SECTION
  DO i=1,n
    c(i) = a(i) + b(i)
  END DO

!$OMP SECTION
  DO i=1,n
    d(i) = a(i) * b(i)
  END DO

!$OMP END SECTIONS nowait
!$OMP END PARALLEL
```

- There is an implied barrier at the end of a SECTIONS directive, unless the `nowait` clause is used (clauses are described in detail later).
- It is up to the implementation to decide which threads will execute a section and which threads will not, and it can vary from execution to execution.

Compiler Directives: Work Sharing Constructs

A Simple DO/for Example

```
#pragma omp parallel for
  for (i=0; i<n; i++)
    x[i] = x[i] + y[i];
```

```
!$OMP PARALLEL DO
  DO i=1,n
    x(i) = x(i) + y(i)
  END DO
!$OMP END PARALLEL DO
```

- Here x, y and n are shared variables among all threads and i is private in the parallel loop.
- The attributes can be made explicit by

```
#pragma omp parallel for shared(x,y,n) private(i)
  for (i=0; i<n; i++)
    x[i] = x[i] + y[i];
```

or

```
#pragma omp parallel for default(x,y,n) private(i)
  for (i=0; i<n; i++)
    x[i] = x[i] + y[i];
```

- The value of i is undefined after the loop.

. . . The above given specifications are called **clauses**.

Compiler Directives: Clauses

Directives have a couple of optional **clauses** that specify details of the parallel section. The most common ones are given here:

- **Data sharing attribute clauses:**
  - `shared`: the data within a parallel region is shared, which means visible and accessible by all threads simultaneously. By default, all variables in the work sharing region are `shared` except the loop iteration counter.
  - `private`: the data within a parallel region is private to each thread, which means each thread will have a local copy and use it as a temporary variable. A private variable is not initialized and the value is not maintained for use outside the parallel region. By default, the loop iteration counters in the OpenMP loop constructs are `private`.
  - `default`: allows the programmer to state that the default data scoping within a parallel region will be either `shared`, or `none` for C/C++, or `shared`, `firstprivate`, `private`, or `none` for Fortran. The `none` option forces the programmer to declare each variable in the parallel region using the data sharing attribute clauses.
  - `firstprivate`: like `private` except initialized to original value.

Compiler Directives: Clauses

- **Synchronization clauses:**
  - `barrier`: each thread waits until all of the other threads of a team have reached this point. A work-sharing construct has an implicit barrier synchronization at the end.
  - `nowait`: specifies that threads completing assigned work can proceed without waiting for all threads in the team to finish. In the absence of this clause, threads encounter a barrier synchronization at the end of the work sharing construct.
- **Scheduling clauses:**
  - `schedule(type, chunk)`: specifies how to divide the iterations among the threads. This is useful if the work sharing construct is a do-loop or for-loop. The `type` can be one of `static`, `dynamic`, `guided` or `runtime`.

    For example, `schedule(static,10)` divides the loop into pieces of 10 iterations and allocates them to threads.

Compiler Directives: Clauses

- **IF control:**
  - `if`: this will cause the threads to parallelize the task only if a condition is met. Otherwise the code block executes serially.
- **Reduction:**
  - `reduction(operator | intrinsic : list)`: the `reduction` clause performs a reduction on the variables that appear in its list. A `private` copy for each list variable is created for each thread. At the end of the `reduction`, the `reduction operator` is applied to all `private` copies of the `shared` variable, and the final result is written to the global `shared` variable.

Compiler Directives: Clauses

- The table below summarizes which clauses are accepted by which OpenMP directives

Clause	Directive					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS
IF	●				●	●
PRIVATE	●	●	●	●	●	●
SHARED	●	●			●	●
DEFAULT	●				●	●
FIRSTPRIVATE	●	●	●	●	●	●
REDUCTION	●	●	●		●	●
SCHEDULE		●			●	
NOWAIT		●	●	●		

- The `master` directive does not accept clauses.

Compiler Directives:
Critical Sections and Reduction Variables

Suppose that you want to parallize the next code snippet:

```
int sum=0;
  for (i=0; i<n; i++) {
    int val = f(i);
    sum = sum + val;
  }
```

A first attempt:

```
int sum=0;
#pragma omp parallel for
  for (i=0; i<n; i++) {
    int val = f(i);
    sum = sum + val;
  }
```

Problem: there is a *race condition* in the updating of sum.

One solution is to usea a critical section:

```
int sum=0;
#pragma omp parallel for
  for (i=0; i<n; i++) {
    int val = f(i);
#pragma omp critical
    sum = sum + val;
}
```

Only one thread at a time is allowed into a critical section.

An alternative is to use reduction variable:

```
int sum=0;
#pragma omp parallel for reduction(+:sum)
  for (i=0; i<n; i++) {
    int val = f(i);
    sum = sum + val;
  }
```

Reduction variables are in between private and shared.

Some OpenMP Library Routines

The OpenMP provides a number of useful run-time library routines. E.g.

- Execution environment subprograms
  - `omp_set_num_threads(integer)`: Sets the number of threads that will be used in the next parallel region. Must be a positive integer.
- Execution environment functions
  - `omp_get_num_threads`: Returns the number of threads that are currently in the team executing the parallel region from which it is called.
  - `omp_get_max_threads`: Returns the maximum value that can be returned by a call to the `omp_get_num_threads` function.
  - `omp_get_thread_num`: Returns the thread number of the thread, within the team, making this call. This number will be between 0 and `omp_get_num_threads`-1. The master thread of the team is thread 0.
  - `omp_get_num_procs`: Returns the number of processors that are available to the program.

Some OpenMP Library Routines

- Timing routines
  - `omp_get_wtime`: Returns a double-precision floating point value equal to the number of elapsed seconds since some point in the past. Usually used in ”pairs” with the value of the first call subtracted from the value of the second call to obtain the elapsed time for a block of code.
  - `omp_get_wtick`: Returns a double-precision floating point value equal to the number of seconds between successive clock ticks.

Environmental Variables

- Certain features of OpenMP execution environment can be controlled by **environmental variables**.
- Environmental variables are used to control loop iterations scheduling, default number of threads, etc.
- For example `OMP_NUM_THREADS` is used to specify number of threads for an application.

One More Example

```
PROGRAM omp_prog
  USE omp_lib
  IMPLICIT NONE

  INTEGER, PARAMETER :: n=100, chunksize=10
  REAL, DIMENSION(n) :: a,b
  REAL :: result
  INTEGER :: chunk, i

  ! Some initializations
  DO i=1,n
    a(i)=i*1.0
    b(i)=i*2.0
  END DO
  result = 0.0
  chunk = chunksize

  ! Set the number of threads
  CALL omp_set_num_threads(4)

  ! Parallel section
  !$OMP PARALLEL DO DEFAULT(shared) PRIVATE(i) &
  !$OMP SCHEDULE(static,chunk) &
  !$OMP REDUCTION(+:result)

      DO i=1,n
        result = result + (a(i)*b(i))
      END DO

  !$OMP END PARALLEL DO NOWAIT

  PRINT*,'result = ',result

END PROGRAM omp_prog
```

All the other variables but "i" will be shared with all threads.

Iterations of the parallel loop will be distributed in equal sized blocks to each thread in the team.

At the end of the parallel loop construct all threads will add their values of "result" to update the master thread's global copy.

Threads will not synchronize upon completing their individual pieces of work.

Finally, Be Aware of ...

- In C the OpenMP directives are case-sensitive but in Fortran they are case-insensitive.
- All environment variable names are uppercase. The values assigned to them are not case sensitive.
- OpenMP specifies nothing about parallel I/O. This is particularly important if multiple threads attempt to write/read from the same file.
  - If every thread conducts I/O to a different file, the issues are not as significant.
  - It is entirely up to the programmer to insure that I/O is conducted correctly within the context of a multi-threaded program.

References and Further Information

- **OpenMP website** : http://openmp.org
- **Tutorial to OpenMP** (main source of this course material):

  https://computing.llnl.gov/tutorials/openMP/
- **Wikipedia:** en.wikipedia.org/wiki/OpenMP
- **Tools for High Performance Computing 2009** -course material by Antti Kuronen, 2009.

  www.physics.helsinki.fi/courses/s/stltk/