# Programming Shared Address Space

**Chapter#7**

# OpenMP: a Standard for Directive Based Parallel Programming

- OpenMP is a directive-based API that can be used with FORTRAN, C, and C++ for programming shared address space machines.
- OpenMP directives provide support for concurrency, synchronization, and data handling while obviating the need for explicitly setting up mutexes, condition variables, data scope, and initialization.

# OpenMP Programming Model

- OpenMP directives in C and C++ are based on the `#pragma` compiler directives.
- A directive consists of a directive name followed by clauses.

```
#pragma omp directive [clause list]
```

- OpenMP programs execute serially until they encounter the `parallel` directive, which creates a group of threads.

```
#pragma omp parallel [clause list]
/* structured block */
```

- The main thread that encounters the `parallel` directive becomes the *master* of this group of threads and is assigned the thread id 0 within the group.

# OpenMP Programming Model

- The clause list is used to specify conditional parallelization, number of threads, and data handling.
  - **Conditional Parallelization:** The clause `if (scalar expression)` determines whether the parallel construct results in creation of threads.
  - **Degree of Concurrency:** The clause `num threads(integer expression)` specifies the number of threads that are created.
  - **Data Handling:** The clause `private (variable list)` indicates variables local to each thread. The clause `firstprivate (variable list)` is similar to the `private`, except values of variables are initialized to corresponding values before the parallel directive. The clause `shared (variable list)` indicates that variables are shared across all the threads.

# OpenMP Programming Model

```
int a, b;
main() {
  // serial segment
  #pragma omp parallel num_threads (8) private (a) shared (b)
  {
    // parallel segment
  }
  // rest of serial segment
}
```

Sample OpenMP program

```
int a, b;
main() {
  // serial segment

  for (i = 0; i < 8; i++)
      pthread_create (........, internal_thread_fn_name, ...);

  for (i = 0; i < 8; i++)
      pthread_join (.......);

  // rest of serial segment

}

void *internal_thread_fn_name (void *packaged_argument) [
    int a;
  // parallel segment
}
```

Code inserted by the OpenMP compiler

Corresponding Pthreads translation

- A sample OpenMP program along with its Pthreads translation that might be performed by an OpenMP compiler.

# OpenMP Programming Model

```
#pragma omp parallel if (is_parallel== 1) num_threads(8) \
    private (a) shared (b) firstprivate(c) {
 /* structured block */
}
```

- If the value of the variable is_parallel equals one, eight threads are created.
- Each of these threads gets private copies of variables a and c, and shares a single value of variable b.
- The value of each copy of c is initialized to the value of c before the parallel directive.
- The default state of a variable is specified by the clause default (shared) or default (none).

# Reduction Clause in OpenMP

- The `reduction` clause specifies how multiple local copies of a variable at different threads are combined into a single copy at the master when threads exit.
- The usage of the `reduction` clause is `reduction (operator: variable list)`.
- The variables in the list are implicitly specified as being private to threads.
- The `operator` can be one of `+, *, -, &, |, ^, &&,` and `||.`

```
#pragma omp parallel reduction(+: sum) num_threads(8) {
/* compute local sums here */
}
/*sum here contains sum of all local instances of sums */
```

# OpenMP Programming: Example

```
/*
   ******************************************
   **
An OpenMP version of a threaded program to compute PI.
******************************************
   */
#pragma omp parallel default(private) shared (npoints)
   \
    reduction(+: sum) num_threads(8)
{
    num_threads = omp_get_num_threads();
    sample_points_per_thread = npoints / num_threads;
    sum = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
        rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
        rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
             (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
             sum ++;
    }
}
```

# Specifying Concurrent Tasks in OpenMP

- The `parallel` directive can be used in conjunction with other directives to specify concurrency across iterations and tasks.
- OpenMP provides two directives - `for` and `sections` - to specify concurrent iterations and tasks.
- The `for` directive is used to split parallel iteration spaces across threads. The general form of a for directive is as follows:

```
#pragma omp for [clause list]
    /* for loop */
```

- The clauses that can be used in this context are: `private`, `firstprivate`, `lastprivate`, `reduction`, `schedule`, `nowait`, and `ordered`.

# Specifying Concurrent Tasks in OpenMP: Example

```
#pragma omp parallel default(private) shared (npoints) \
    reduction(+: sum) num_threads(8)
{
    sum = 0;
    #pragma omp for
    for (i = 0; i < npoints; i++) {
        rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
        rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
            (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            sum ++;
    }
}
```

# Assigning Iterations to Threads

- The `schedule` clause of the `for` directive deals with the assignment of iterations to threads.
- The general form of the `schedule` directive is `schedule(scheduling_class[, parameter]).`
- OpenMP supports four scheduling classes: `static, dynamic, guided,` and `runtime`.

# Assigning Iterations to Threads: Example

```
/* static scheduling of matrix multiplication loops */
#pragma omp parallel default(private) shared (a, b, c, dim) \
    num_threads(4)
    #pragma omp for schedule(static)
    for (i = 0; i < dim; i++) {
        for (j = 0; j < dim; j++) {
            c(i,j) = 0;
            for (k = 0; k < dim; k++) {
                c(i,j) += a(i, k) * b(k, j);
            }
        }
    }
```

# Data sharing attribute clauses [1/2]

- *shared*: the data declared outside 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 declared 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.

# Data sharing attribute clauses [2/2]

- *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.
- *lastprivate*: like *private* except original value is updated after construct.
- *reduction*: a safe way of joining work from all threads after construct.

# Synchronization clauses [1/2]

- *critical*: the enclosed code block will be executed by only one thread at a time, and not simultaneously executed by multiple threads. It is often used to protect shared data from race conditions.
- *atomic*: the memory update (write, or read-modify-write) in the next instruction will be performed atomically. It does not make the entire statement atomic; only the memory update is atomic. A compiler might use special hardware instructions for better performance than when using *critical*.

# Synchronization clauses [2/2]

- *ordered*: the structured block is executed in the order in which iterations would be executed in a sequential loop
- *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 [1/2]

- *schedule(type, chunk)*: This is useful if the work sharing construct is a do-loop or for-loop. The iteration(s) in the work sharing construct are assigned to threads according to the scheduling method defined by this clause. The three types of scheduling are:
- *static*: Here, all the threads are allocated iterations before they execute the loop iterations. The iterations are divided among threads equally by default. However, specifying an integer for the parameter *chunk* will allocate chunk number of contiguous iterations to a particular thread.

# Scheduling clauses [2/2]

- *dynamic*: Here, some of the iterations are allocated to a smaller number of threads. Once a particular thread finishes its allocated iteration, it returns to get another one from the iterations that are left. The parameter *chunk* defines the number of contiguous iterations that are allocated to a thread at a time.
- *guided*: A large chunk of contiguous iterations are allocated to each thread dynamically (as above). The chunk size decreases exponentially with each successive allocation to a minimum size specified in the parameter *chunk*

# Parallel For Loops

- Often, it is desirable to have a sequence of `for`-directives within a parallel construct that do not execute an implicit barrier at the end of each `for` directive.
- OpenMP provides a clause - `nowait`, which can be used with a for directive.

# Parallel For Loops: Example

```
#pragma omp parallel
{
   #pragma omp for nowait
      for (i = 0; i < nmax; i++)
         if (isEqual(name, current_list[i])
            processCurrentName(name);
   #pragma omp for
      for (i = 0; i < mmax; i++)
         if (isEqual(name, past_list[i])
            processPastName(name);
}
```

# The `sections` Directive

- OpenMP supports non-iterative parallel task assignment using the sections directive.
- The general form of the sections directive is as follows:

```
#pragma omp sections [clause list]
{
    [#pragma omp section
        /* structured block */
    ]
    [#pragma omp section
        /* structured block */
    ]
    ...
}
```

# The `sections` Directive: Example

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            taskA();
        }
        #pragma omp section
        {
            taskB();
        }
        #pragma omp section
        {
            taskC();
        }
    }
```

# Nesting `parallel` Directives

- Nested parallelism can be enabled using the `OMP_NESTED` environment variable.
- If the `OMP_NESTED` environment variable is set to `TRUE`, nested parallelism is enabled.
- In this case, each parallel directive creates a new team of threads.

# Synchronization Constructs in OpenMP

- OpenMP provides a variety of synchronization constructs:

```
#pragma omp barrier
#pragma omp single [clause list]
    structured block
#pragma omp master
    structured block
#pragma omp critical [(name)]
    structured block
#pragma omp ordered
    structured block
```

# OpenMP Library Functions

- In addition to directives, OpenMP also supports a number of functions that allow a programmer to control the execution of threaded programs.

```
/* thread and processor count */
void omp_set_num_threads (int
  num_threads);
int omp_get_num_threads ();
int omp_get_max_threads ();
int omp_get_thread_num ();
int omp_get_num_procs ();
int omp_in_parallel();
```

# OpenMP Library Functions

```
/* controlling and monitoring thread creation */
void omp_set_dynamic (int dynamic_threads);
int omp_get_dynamic ();
void omp_set_nested (int nested);
int omp_get_nested ();
/* mutual exclusion */
void omp_init_lock (omp_lock_t *lock);
void omp_destroy_lock (omp_lock_t *lock);
void omp_set_lock (omp_lock_t *lock);
void omp_unset_lock (omp_lock_t *lock);
int omp_test_lock (omp_lock_t *lock);
```

- In addition, all lock routines also have a nested lock counterpart
- for recursive mutexes.

# Environment Variables in OpenMP

- `OMP_NUM_THREADS`: This environment variable specifies the default number of threads created upon entering a parallel region.
- `OMP_SET_DYNAMIC`: Determines if the number of threads can be dynamically changed.
- `OMP_NESTED`: Turns on nested parallelism.
- `OMP_SCHEDULE`: Scheduling of for-loops if the clause specifies runtime

# Explicit Threads versus Directive Based Programming

- Directives layered on top of threads facilitate a variety of thread-related tasks.
- A programmer is rid of the tasks of initializing attributes objects, setting up arguments to threads, partitioning iteration spaces, etc.
- There are some drawbacks to using directives as well.
- An artifact of explicit threading is that data exchange is more apparent. This helps in alleviating some of the overheads from data movement, false sharing, and contention.
- Explicit threading also provides a richer API in the form of condition waits, locks of different types, and increased flexibility for building composite synchronization operations.
- Finally, since explicit threading is used more widely than OpenMP, tools and support for Pthreads programs are easier to find.