ECE5610/CSC6220
Introduction to Parallel and Distribution Computing

Lecture 5: Parallel Programming
with Thread (Part 3)
The OpenMP Programming Model

• OpenMP: A standard for directive based parallel programming
• Directives provide support for concurrency, synchronization and data handling.
• Avoids explicit setting for mutexes, condition variables, etc.
• Can be used with C, C++ and FORTRAN.
• Directives in C, C++ are based on #pragma compiler directives:
  #pragma omp directive [clause list]
The OpenMP Programming Model

• Programs execute serially until they encounter the parallel directive:
  
  `# pragma omp parallel [clause list]
  /* structured block */`

• Creates a group of threads.

• Number of threads is specified using an environment variable, in the directive or at runtime.

• The main thread that encounters the parallel directive becomes the master of the group (tid = 0).

• Each thread executes the structured block.
The OpenMP Programming Model (Cont’d)

• OpenMP uses the fork-join model of parallel execution:

• 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
Parallel Directive: Clauses

Conditional parallelization:

\[
\text{if (scalar expression)}
\]

Degree of concurrency:

\[
\text{num\_threads (integer expression)}
\]

Data handling:

\[
\text{private (variable list)}
\]
\[
\text{firstprivate (variable list)}
\]
\[
\text{shared (variable list)}
\]

Example:

\[
\text{# pragma omp parallel if (is\_parallel ==1) num\_threads (8) } \backslash
\text{ private (a) shared (b) firstprivate(c)}
\]

\[
\{
\text{/*structured block*/}
\}
\]
OpenMP to Pthreads Translation

Sample OpenMP program

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

Corresponding Pthreads translation

```c
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
}
```
**Parallel Directive: Data Handling**

Default state of a variable:

- **default (shared)**
- **default (none)** - state must be explicitly specified

Reduction: specifies how multiple local copies of a variable at different threads are combined at the master.

- **reduction (operator: variable list)**
  - The operator can be +, *, -, &, |, ^, &&, or ||.
  - The variables in the list are implicitly declared as being private to each thread

Example:

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

```c
/* ***********************************************
An OpenMP version of a threaded program to compute PI.
*********************************************** */

private (num_threads, sample_points_per_thread, rand_no_x, rand_no_y)

#pragma omp parallel
    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 ++;
    }
}

Note: There is no default(private) clause in C/C++.
This is because many C standard library facilities are
implemented using macros that reference global variables.
```
The **for** Directive

Used to split parallel iterations across threads.

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

Clauses:

- **private, reduction**
  - **firstprivate**: similar to private, except that values of variables on entering the threads are initialized to corresponding values before the parallel directive.
  - **lastprivate** - the value copied back into the original variable object is obtained from the last (sequentially) iteration or section of the enclosing construct.

- **schedule** - specifies how iterations will be assigned to threads

- **nowait** - threads can proceed to the next statement without waiting for all other threads to complete the for loop.

- **ordered** - specifies that there is ordering between successive iterations of the loop.
The **for** Directive: Example

- Each iteration of the for *loop* is independent and can be executed concurrently.
- Attention: loop index goes from 0 to npoints-1.
- Some restrictions for the loop: such as no break statement.

```c
#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

Schedule clause:

```
schedule(scheduling_class[, parameter])
```

Static scheduling:

```
static[, chunk-size]
```
- splits the iteration space into equal chunks of size chunk-size and assigns them to threads in a round-robin fashion. If no parameter specified, the number of chunks is the number of threads
Static Scheduling: Example

```c
#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);
        }
    }
}
```

schedule(static)

schedule(static, 16)

Splitting nested iterations allowed
Other Scheduling Classes

Dynamic scheduling:

dynamic[, chunk-size] - splits the iteration space into equal chunks of size chunk-size and assigns them to threads as they become idle. If no chunk_size is specified, it defaults to a single iteration per chunk.

Guided scheduling:

guided[, chunk-size] - reduces the chunk size as we proceed through the computation. For a default chunk size of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1. => reduced idling overhead.

Runtime scheduling:

runtime - the scheduling class and the chunk size is determined by the environment variable OMP_SCHEDULE.
Synchronization across Multiple for Directives

nowait clause: indicates that the threads can proceed to the next statement without waiting for all other threads to complete the for loop.

Example:

```c
#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< nmax; i++)
        if (isEqual(name, past_list[i])
            processPastName(name);
}
```
The **section** Directive

Assign independent tasks to different threads.

*Example:* executing three independent tasks A, B, C in parallel.

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

If no preceding parallel directive, the \textit{for} and \textit{sections} directives will execute serially.

\begin{verbatim}
#pragma omp parallel shared(n)
{
    #pragma omp for
    for (i=0; i<n; i++){
        /*body of parallel for loop */
    }
}
\end{verbatim}

is identical to:

\begin{verbatim}
#pragma omp parallel for shared(n)
{
    for (i=0; i<n; i++){
        /*body of parallel for loop */
    }
}
\end{verbatim}
# Merging Directives

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

```c
#pragma omp parallel sections
{
    #pragma omp section
    {
        taskA();
    }
    #pragma omp section
    {
        taskB();
    }
}
```
Nesting Parallel Directives

```c
#pragma omp parallel for default(private) shared (a, b, c, dim) \
    num_threads(2)
for (i = 0; i < dim; i++) {
    #pragma omp parallel for default(private) shared (a, b, c, dim) \
           num_threads(2)
    for (j = 0; j < dim; j++) {
        c(i,j) = 0;
        #pragma omp parallel for default(private) \
           shared (a, b, c, dim) num_threads(2)
        for (k = 0; k < dim; k++) {
            c(i,j) += a(i, k) * b(k, j);
        }
    }
}
```

**OMP_NESTED** must be **TRUE** to enable nested parallelism.
Synchronization Constructs in OpenMP

Synchronization point: barrier directive

#pragma omp barrier

All threads in a team wait until others have caught up and then release.

Single thread executions:

#pragma omp single [clause list]
/* structured block*/
- specifies a structured block that is executed by a single (arbitrary) thread.
- the first thread enters the block.
- there is an implicit barrier associated with single unless a nowait is specified.
- useful for computing global data as well as performing I/O

#pragma omp master
/* structured block*/
- only the master thread executes the structured block.
- no implicit barrier is associated.
Critical Sections: **critical Directive**

```c
#pragma omp parallel sections
{
    #pragma parallel section
    {
        /*producer thread*/
        task = produce_task();
        #pragma omp critical (task_queue)
        {
            insert_into_queue(task);
        }
    }
    #pragma parallel section
    {
        /*consumer thread*/
        #pragma omp critical (task_queue)
        {
            task = extract_from_queue();
        }
    }
}
```

- `#pragma omp critical [(name)]`
- Ensure that at any point in a program only one thread is within the critical section specified by a certain name.
- If no name is specified, the critical section maps to a default name that is the same for all unnamed critical sections.
Example: critical Directive

```c
#include <omp.h>
main()
{
    int x;
    x = 0;

    #pragma omp parallel shared(x)
    {
        #pragma omp critical
        x = x + 1;
    } /* end of parallel section */
}```
In-Order Execution: The `ordered` Directive

- To execute a segment of a parallel loop in the order in which the serial version would execute it
- Represent an ordered serialization point in the program

Example: computing the cumulative sum in an array.

```c
cumul_sum[0] = list[0];
#pragma omp parallel for private(i) \
    shared (cumul_sum, list, n) ordered
for (i=1; i<n; i++) {
    /* other processing on list[i] */ #pragma omp ordered
    {cumul_sum[i] = cumul_sum[i-1] + list[i]}
}
```
Memory Consistency: The `flush` Directive

- Register allocated variable may be inconsistent.
- Provides a mechanism for making memory consistent across threads.
  ```
  # pragma omp flush[(list)]
  ```
- Applies only to shared variables.
- All write operations are committed to memory at a flush.
- Implicit flush in: `barrier, critical, ordered, parallel, parallel for, parallel sections`, and at the exit of `for, sections and single`.
- Not implicit if `nowait` clause is used.
Data Handling in OpenMP

• If a thread initializes and uses a variable and no other thread accesses the data => make it private

• If a thread repeatedly reads a variable that has been initialized earlier in the program => make it firstprivate.

• Use reduction clause when threads manipulate local data that needs to be combined in a global data.

• Threads manipulating large data structures => break the data into smaller structures and make them private to the corresponding thread.
OpenMP Functions: Controlling Number of Threads and Processors

```c
void omp_set_num_threads (int num_threads);
    - sets the number of threads for the next parallel region.
int omp_get_num_threads ();
    - returns the number of threads
int omp_get_max_threads ();
    - returns the maximum number of threads that could be created by a parallel directive.
int omp_get_thread_num ();
    - returns the thread id.
int omp_get_num_procs ();
    - returns the number of available processors.
int omp_in_parallel ();
    - returns a non-zero value if called from a parallel region.
void omp_set_dynamic (int dynamic_threads);
    - enables or disables dynamic adjustment of the number of threads.
int omp_get_dynamic ();
    - returns a non-zero value if dynamic adjustment is enabled.
void omp_set_nested (int nested);
    - enables nested parallelism if nested is non-zero.
int omp_get_nested ();
    - returns a non-zero value if nested parallelism is enabled.
```
Mutual Exclusion

```c
void omp_init_lock (omp_lock_t *lock);
  - initializes a lock
void omp_destroy_lock (omp_lock_t *lock);
  - discards a lock
void omp_set_lock (omp_lock_t *lock);
  - lock
void omp_unset_lock (omp_lock_t *lock);
  - unlock
int omp_test_lock (omp_lock_t *lock);
  - attempt to set a lock, successful if returns a non-zero value
```

Nested Locks *(recursive mutex)*

```c
void omp_init_nest_lock (omp_nest_lock_t *lock);
  - initializes a nested lock
void omp_destroy_nest_lock (omp_nest_lock_t *lock);
  - discards a nested lock
void omp_set_nest_lock (omp_nest_lock_t *lock);
  - lock
void omp_unset_nest_lock (omp_nest_lock_t *lock);
  - unlock
int omp_test_nest_lock (omp_nest_lock_t *lock);
  - attempt to set a nested lock, successful if returns a non-zero value
```
Environment Variables in OpenMP

**OMP_NUM_THREADS**
- specifies the default number of threads created upon entering a parallel region.

**OMP_DYNAMIC**
- when set to **TRUE** allows the number of threads to be controlled at runtime.

**OMP_NESTED**
- when set to **TRUE** enables nested parallelism

**OMP_SCHEDULE**
- controls the scheduling class
  
  ```
  setenv OMP_SCHEDULE 'static,4'
  ```
Explicit Threads versus Directive Based Programming

• Advantage of using directives
  ➢ Directives layered on top of threads facilitate a variety of thread-related tasks.
  ➢ Programmers don’t have to take tasks of initializing attributes objects, setting up arguments to threads, partitioning iteration spaces, etc.

• Advantage of using threaded programming
  ➢ 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.