15-418/618, Spring 2025
OpenMP: A Short Introduction

This is a short introduction to OpenMP, an API that supports multi-threaded, shared address space (aka shared memory) parallelism. The purpose of this document is to give you some basic ideas about how to use OpenMP so that you can start working on your assignment.

1 Learning OpenMP

This document is only meant to get you started on coding and compiling your OpenMP programs. The official web site for OpenMP (https://www.openmp.org) includes specifications along with a collection of several useful OpenMP tutorials. For example, this OpenMP tutorial from Lawrence Livermore National Laboratory (llnl.org) is a useful reference.

There is also a nice book on OpenMP called "Parallel Programming in OpenMP", by Rohit Chandra *et al.* (ISBN 1558606718).

2 OpenMP Basics

OpenMP stands for *Open* specifications for *M*ulti*P*rocessing. It has been jointly defined and endorsed by a group of major computer hardware and software vendors. It is a standardized Application Program Interface (API) that supports multi-threaded, shared address space parallelism. However, different vendors are allowed to write different implementations that are optimized for their specific machine architectures.

OpenMP supports thread-based parallelism. It provides an explicit programming model to control the creation, communication and synchronization of multiple threads. 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.
- 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.
- When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.

The OpenMP API is comprised of three primary components:

- *Compiler Directives*: specify parallelism in the C source code.
- *Runtime Library Routines*: provide access to various runtime parameters.
- *Environment Variables*: direct how the runtime system should behave.

3 Compiler directives

Virtually all of OpenMP parallelism is specified through the use of compiler directives which are imbedded in C source code. Directives follow conventions of the C standard for compiler directives. Long directive lines can be "continued" on succeeding lines by escaping the newline character with a backslash "\" at the end of a directive line.

There are four different types of directives:

- Parallel construct
- Work-sharing construct
- Combined parallel work-sharing constructs
- Synchronization directives

3.1 Parallel construct

A parallel region is a block of code that will be executed by multiple threads. This is the fundamental OpenMP parallel construct.

The syntax of this construct is as follows:

```
#pragma omp parallel [clause ...]
                     if (scalar_expression)
                     private (list)
                     shared (list)
                     default (shared | none)
                     firstprivate (list)
                     reduction (operator: list)
                     copyin (list)

   structured_block
```

When a thread reaches a PARALLEL directive, it creates a team of threads and becomes the master of the team. 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.

The number of threads in a parallel region is determined by the following factors, in order of precedence:

- Use of the `omp_set_num_threads()` library function.
- Setting of the `OMP_NUM_THREADS` environment variable.
- Implementation default.

Threads are numbered from 0 (master thread) to N-1.

3.2 Work-sharing constructs

A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it. Note that work-sharing constructs do not launch new threads. There is no implied barrier upon entry to a work-sharing construct, however there is an implied barrier at the end of a work sharing construct.

There are three different work-sharing constructs.

The syntax of the `for` directive is as follows:

```
#pragma omp for [clause ...]
                schedule (type [,chunk])
                ordered
                private (list)
                firstprivate (list)
                lastprivate (list)
                shared (list)
                reduction (operator: list)
                nowait

   for_loop
```

In OpenMP, loop iterations are assigned to threads in contiguous ranges called chunks. By controlling how these chunks are assigned to threads (either *dynamically* or in some *static* fashion), and the number of iterations per chunk (i.e. the "*chunk size*"), a scheduling scheme attempts to balance the work across threads.

The schedule clause describes how iterations of the loop are divided among the threads in the team. There are four different schedules available. Here is a brief description of two of those schedules:

- `static`: Loop iterations are divided into pieces of size `chunk` and then statically assigned to threads. If `chunk` is not specified, then the iterations are divided evenly (if possible) across the threads in contiguous chunks.
- `dynamic`: Loop iterations are divided into pieces of size `chunk`, and dynamically scheduled among the threads. When a thread finishes one chunk, it is dynamically assigned another. The default *chunk* size is 1.

Note that chunk size must be specified as a loop-invariant integer expression, since there is no synchronization during its evaluation by different threads.

3.3 Combined parallel work-sharing constructs

Combined parallel working-sharing constructs are shortcuts for specifying a parallel region that contains only one work-sharing construct. The semantics of these directives are identical to the case of explicitly specifying a parallel directive followed by a single work-sharing construct. In particular, you may use the following syntax:

```
#pragma omp parallel sections [clause ...]
                              default (shared | none)
                              shared (list)
                              private (list)
                              firstprivate (list)
                              lastprivate (list)
                              reduction (operator: list)
                              copyin (list)
                              ordered

   structured_block
```

3.4 Synchronization directives

There are various synchronization constructs available to coordinate the work by multiple threads. Here is an incomplete list of directives that you may find useful.

`master`: specifies a region that is to be executed only by the master thread of the team. All other threads on the team skip this section of code. The syntax is:

```
#pragma omp master
   structured_block
```

`critical`: specifies a region of code that must be executed by only one thread at a time. The syntax is:

```
#pragma omp critical
   structured_block
```

`barrier`: synchronizes all threads in the team. When a `barrier` directive is reached, a thread will wait at that point until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier. The syntax is:

```
#pragma omp barrier
```

`atomic`: specifies that a specific memory location must be updated atomically, rather than letting multiple threads attempt to write to it. In essence, this directive provides a miniature critical section. The syntax is:

```
#pragma omp atomic
   statement_expression
```

`ordered`: specifies that iterations of the enclosed loop will be executed in the same order as if they were executed on a serial processor. The syntax is:

```
#pragma omp ordered
   structured_block
```

3.5 Data scope attribute clauses

An important consideration for OpenMP programming is the understanding and use of data scoping. Because OpenMP is based upon the shared address space programming model, most variables are shared by default. In particular, all of the variables visible when entering the parallel region are global (shared). Private variables include loop index variables and stack variables in subroutines called from parallel regions. In addition, data scope attribute clauses can be used to explicitly define how variables should be scoped. They include PRIVATE, FIRSTPRIVATE, LASTPRIVATE, SHARED, DEFAULT, REDUCTION and COPYIN. They can be used in conjunction with certain directives (PARALLEL, `for`) to control the scoping of enclosed variables.

4 Runtime Library Routines

The OpenMP standard defines an API for library calls that perform a variety of functions:

- Query the number of threads/processors, set number of threads to use.
- General purpose locking routines (semaphores).
- Set execution environment functions: nested parallelism, dynamic adjustment of threads.

You need to include file "`omp.h`" in your source code to have access to these functions.

The functions that you may need for your assignment are listed as follows. However, you may use other functions as necessary.

- `void omp_set_num_threads(int num_threads)`: Sets the number of threads that will be used in the next parallel region.
- `int omp_get_num_threads()`: Returns the number of threads that are currently in the team executing the parallel region from which it is called.
- `int omp_get_max_threads()`: Returns the maximum value that can be returned by a call to the `omp_get_num_threads` function.
- `int 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.
- `int omp_get_num_procs()`: Returns the number of processors that are available to the program.
- `int omp_in_parallel()`: May be called to determine if the section of code which is executing is parallel or not.
- `void omp_init_lock(omp_lock_t *lock)`: Initializes a lock associated with the lock variable. The initial state is unlocked.

- `void omp_destroy_lock(omp_lock_t *lock)`: Disassociates the given lock variable from any locks. It is illegal to call this routine with a lock variable that is not initialized.
- `void omp_set_lock(omp_lock_t *lock)`: Forces the executing thread to wait until the specified lock is available. A thread is granted ownership of a lock when it becomes available.
- `void omp_unset_lock(omp_lock_t *lock)`: Releases the lock from the executing subroutine.
- `void omp_test_lock(omp_lock_t *lock)`: Attempts to set a lock, but does not block if the lock is unavailable.

5 Environment Variables

OpenMP provides four environment variables for controlling the execution of parallel code. You may need to use `OMP_NUM_THREADS`, which sets the maximum number of threads to use during execution. For example:

```
setenv OMP_NUM_THREADS 8
```

Remember that `omp_set_num_threads` overrides the OMP_NUM_THREADS.