ARIZONA STATE UNIVERSITY

CSE 430 SLN 70967— **Operating Systems** — Fall 2014

Instructor: Dr. Violet R. Syrotiuk

Project #2

Available Tuesday 09/30/2014; due Tuesday 10/28/2014

The goal of this project is to implement a preprocessor for OpenMP directives using the `pthreads` library.

1 OpenMP Preprocessor

As you know, the OpenMP API was developed to enable shared memory parallel programming. It uses the fork/join programming model. For C/C++, pragmas are provided by the API to control parallelism. In OpenMP these are called directives. They always always start with `#pragma omp`, then a keyword that identifies the directive, followed by zero or more clauses. Clauses are used to further specify the behaviour of the directive, or to further control parallel execution.

In this project, you will write a preprocessor for select OpenMP directives. Your preprocessor will therefore read in a C/C++ program that contains OpenMP directives with select clauses as input. As output, it will produce a C/C++ program in which the required behaviour of each OpenMP directive is implemented using the `pthreads` library. Since your program is a preprocessor, it should be possible to compile and run the program your preprocessor produced as output! You should see the same behaviour as the original program compiled using the "real" OpenMP preprocessor.

1.1 OpenMP Constructs

In order to simplify your preprocessor, you are responsible to implement a subset of the directives, and a subset of clauses for each directive. Specifically, you are responsible to implement the following constructs:

The `parallel` construct: The parallel construct is used to specify the computations that should be executed in parallel. In this case, each of `num_threads(integer)` threads redundantly execute all of the code. See the sample program in file `par.cc`.

The combined `parallel for` construct: The `for` loop following the `#pragma` should be distributed among `num_threads(integer)` threads. Different threads execute different iterations of the loop and each loop iteration is performed exactly once. If there are more iterations than threads, multiple iterations are assigned to the threads in the team. In the input programs, the upper limit of the `for` loop is guaranteed to be an integer. See the sample program in file `parfor.cc`.

The `critical` construct: The `critical` construct provides a means to ensure that multiple threads do not attempt to update the same shared data simultaneously, i.e., the associated code is placed in a critical section. See the sample program in file `critical.cc`.

The `single` construct: The single construct is associated with the structured block of code immediately following it and specifies that this block should be executed by one thread only. It does not state which thread should execute the code block. The other threads wait at a barrier until the thread executing the single code block has completed. See the sample program in file `single.cc`.

The `openmp_get_thread_num()` library routine: Returns the thread identifier of the calling thread.

Only the `num_threads(integer)`, `private(list)`, and `shared(list)` clauses must be implemented. For more details, see among others [2].

2 Program Requirements

1. Your job is to write an OpenMP preprocessor. That is, you are to write a C/C++ program that read another C/C++ program `x.cc` that may contain the OpenMP directives and clauses, and calls to the `openmp_get_thread_num()` library routine; see §1.1. *You may read the input file (i.e., the C/C++ program file* `x.cc`*) as many times as you feel is necessary.*

2. The output of your preprocessor should also be a C/C++ program, `xpost.cc`, in which the required behaviour of the OpenMP directives is implemented using the `pthreads` library. It should be possible to compile `xpost.cc` and run the program. The output of the program should be identical (except for possibly thread assignment) to compiling `x.cc` using the "real" OpenMP preprocessor.

3 For Honours Credit or Bonus Credit

In addition, implement the additional constructs:

The `sections` construct: The sections construct is the easiest way to get different threads to carry out different kinds of work, since it permits to specify several different code regions, each of which is executed by one of the threads. It consists of two directives:

1. `#pragma omp sections` to indicate the start of the construct, and
2. `#pragma omp section` to mark each distinct section.

Each section must be a structured block of code that is independent of the other sections. At run time, the specified code blocks are executed by the threads in the team. Each thread executes one code block at a time, and each code block is executed exactly once. If there are fewer threads than code blocks, some or all of the threads execute multiple code blocks If there are fewer code blocks than threads, the remaining threads will be idle. See the sample program in file `sections.cc`.

The `barrier` construct: A barrier is a point in the execution of a program where threads wait for each other. No thread in the team of threads may proceed beyond a barrier until all threads in the team have reached that point. (Many OpenMP constructs imply a barrier.) See the sample program in file `barrier.cc`.

The same clauses as in §1.1 are to be implemented for these clauses as well.

4 Hand-in instructions

Submit electronically, before 9:00am on Tuesday, 10/28/2014 using the submission link on Blackboard for Project #2, a zip[1] file named `yourFirstName-yourLastName.zip` containing the following items:

Design and Analysis (30%): Provide a description of the methodology you followed to implement your OpenMP preprocessor. Specifically:

1. Discuss how you implemented each directive, clause, and library routine.
2. How many times did you read the input program? What did you look for on each pass?
3. Describe any other design decisions you made, or solution approaches of interest!

Implementation (50%): Provide a makefile to compile your preprocessor program, and your *documented* C/C++ source code.

You may write your code only in C or C++. You must not alter the requirements of this project in any way.

[1] **Do not** use any other archiving program except `zip`.

Correctness (20%) Your preprocessor program **must** run on `general.asu.edu` using a makefile you provide. Our TA will test them there. Each test will compare the output of the program your preprocessor generated, to the output of the original program using the "real" OpenMP preprocessor. You may use the sample programs as test input.

References

[1] B. Barney, "POSIX Threads Programming," `https://computing.llnl.gov/tutorials/pthreads/`.

[2] B. Chapman, G. Jost, and R. van der Pas, Notes on OpenMP from the book "Using OpenMP: Portable Shared Memory Parallel Programming," Cambridge, MA, USA, MIT Press, 2007.

[3] M. Damian, "Synchronizing Threads with POSIX Semaphores," `http://www.csc.villanova.edu/ mdamian/threads/posixsem.html`

[4] The Open Group, "Man pages for pthreads.h," `http://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html`