



F. Wermelinger
Office: Pierce Hall 211

Lab 2

Experiment with OpenMP locks, critical and atomic constructs

Issued: February 13, 2023
Due: February 24, 2023 11:59pm

In this lab you experiment with mutual exclusion in shared memory programming. You will work on the Counter class example that we have used in lecture 4 to study race conditions. You are measuring the wall time of your multi-threaded code as a function of the mutual exclusion operator and the number of threads you are running your code with.

Submission instructions: please see the lab submission section in the syllabus.^a

^a<https://harvard-iacs.github.io/2023-CS205/pages/syllabus.html#lab-submission>

Task 1: OpenMP Locks

We are going to experiment with the Counter class that we have discussed in conjunction with race conditions in lecture 4. Recall that race conditions happen in *critical sections* where multiple threads can modify a memory location in an unsafe way. It is *your* responsibility to avoid race conditions.

You can find the code to start with in [code/t1/counter.cpp](#). In addition to what you have seen in the lecture, the code includes a few extra variables to determine the number of threads it is running with and a timer section to determine the wall time of the executing code. Spend a few moments to make sure you understand this code

```
// get number of threads
int nthreads;
#pragma omp parallel
{
    #pragma omp master // only primary thread here (could also use omp single)
    nthreads = omp_get_num_threads();
}
```

It spawns a parallel region (which is necessary to determine the number of threads in a team) and the primary thread calls the OpenMP library routine `omp_get_num_threads()`¹.

¹<https://www.openmp.org/spec-html/5.1/openmpsu121.html>

The second use of OpenMP library routines is to measure the wall time of the running code. In this case we use the `omp_get_wtime()` routine. These values will be printed at the end when the program finishes.

- a) The OpenMP runtime library includes a set of general-purpose lock routines that can be used for synchronization. In this task you should extend the given code with an OpenMP mutex that you can use in the loop body to “lock” access to the `.increment()` method of the Counter class. See Section 3.9 in the OpenMP specification (you can find the pdf in the manuals directory in the class repository).

Define a shared variable called `mutex` which is of type `omp_lock_t`². We will use a simple lock for this example. You must initialize the mutex as well before you can use it. See the OpenMP specification or the link in the footnote to find the right runtime routine for this. Implement your code following the TODO: comments in the code.

Once you have initialized your mutex, you can use it in the loop body to protect a *critical section*. See the documentation for the `omp_set_lock` and `omp_unset_lock` routines to find out how to obtain and free a lock.

Finally, the initialization of your mutex allocates some memory internally to the OpenMP library. When you are done using the mutex you should free the memory by calling the `omp_destroy_lock` routine.

When you are done, you can compile the code using the provided Makefile by typing:

```
$ make
```

You can run your code with:

```
$ OMP_NUM_THREADS=1 ./counter
```

This will run your code with one thread. Change the number to run with more threads.

- b) Run your code on an interactive node on the compute cluster. Recall that you can allocate an interactive node with

```
$ salloc -N1 -c32 -t 1:00:00
```

Collect wall time measurements for 1, 2, 4, 8, 16 and 32 threads (put them in a file) and create a plot with the number of threads on the abscissa and the wall time on the ordinate. If all of the 32 core nodes are busy, you may try to request one of the 28 core nodes and collect measurements for 1, 2, 4, 8, 14 and 28 threads. You can request such a node with

```
$ salloc -N1 -c28 -t 1:00:00
```

²<https://www.openmp.org/spec-html/5.1/openmpse39.html>

You should be able to get a VSCode instance in the same way if you prefer that. Label the axes and name the curve “OpenMP lock”. Add the plot in your submission when you are done with the lab. When you build your code, make sure you load a recent GCC compiler like you did in the first lab.

You may use any plotting library you like. Options on the compute cluster are either matplotlib and python or gnuplot³

```
$ module load python/3.9.12-fasrc01
$ module load gnuplot/5.2.6-fasrc01
```

Task 2: OpenMP Critical Construct

In this task we are going to repeat what we did in task 1 but we are going to use the omp critical construct this time. You can find the code to start with in [code/t2/counter.cpp](#).

See Section 2.19.1 in the OpenMP specification for more information on the omp critical construct.

- a) OpenMP provides the critical construct

```
#pragma omp critical
```

which is intended to protect critical sections. It is a *synchronization* construct and a convenience shortcut for the OpenMP lock types we looked at in task 1. Implementation wise an omp critical construct is the same as a lock.

Add a #pragma omp critical construct somewhere in the provided code such that the critical section in the loop body is protected from the data race. Note that you have multiple options to achieve this. We do not indicate TODO comments here on purpose.

- b) Run your code on the interactive node you have allocated in the previous task. Collect wall time measurements for 1, 2, 4, 8, 16 and 32 threads (or 1, 2, 4, 8, 14 and 28 threads depending on the node you are working with) and put them in another file. Add these measurements to the plot you have created in task 1. Name the curve “OpenMP critical”.

Task 3: OpenMP Atomic Construct

In this task we are going to repeat what we did in task 2 but we are going to use the omp atomic construct this time. You can find the code to start with in [code/t3/counter.cpp](#).

See Section 2.19.7 for more information on the omp atomic construct.

- a) OpenMP provides the atomic construct

³<http://www.gnuplot.info/>

```
#pragma omp atomic
```

which creates an atomic block. Atomic operations are only possible for *simple* statements such as `++x` or `x++` for example (see Section 2.19.7 in the OpenMP specification). The trade off for this restriction is a faster synchronization construct than a lock based approach.

Add a `#pragma omp atomic` construct somewhere in the provided code such that the critical section in the loop body is protected from the data race. Unlike task 2, there is only one possible place in the code where you can do this. We do not indicate TODO comments here on purpose.

- b) Run your code on the interactive node you have allocated in the previous task. Collect wall time measurements for 1, 2, 4, 8, 16 and 32 threads (or 1, 2, 4, 8, 14 and 28 threads depending on the node you are working with) and put them in yet another file. *Add* these measurements to the plot you have created in task 1. Name the curve “OpenMP atomic”.

Briefly comment on your observations and journal them in a markdown file named `observations.md` inside your lab root directory.