Barriers

Barriers are used to synchronize all running threads at a particular location within the code. This is most useful when several threads execute the same piece of code in parallel. The threads are made to wait at the barrier until **all** threads reach the barrier. Then all the threads are allowed to continue.

```
Threads 1 ... n (assume all threads execute this code)
---------------

...
execute parallel section 1
....
BARRIER ↞ Threads wait here until all of them reach this location.
...
execute parallel section 2
...
BARRIER ↞ Threads wait here again until all of them reach this location.
```

The `pthreads` library supports the following functions to use barriers:

- `int pthread_barrier_init(pthread_barrier_t *restrict barrier,`
- `const pthread_barrierattr_t *restrict attr, unsigned count);`
- This function initializes the `barrier` with attributes specified by `attr`. If `attr` is NULL, the default condition variable attributes are used. Refer to the man pages for more information on the attributes. The `count` argument specifies the number of threads that must wait at the `barrier` before any of them successfully return from the call.
- `int pthread_barrier_destroy(pthread_barrier_t *barrier);`
- This function destroys the `barrier`. Attempting to destroy a `barrier` upon which other threads are currently blocked results in undefined behavior.
- `int pthread_barrier_wait(pthread_barrier_t *barrier);`
- This function blocks the calling thread until the required number of threads have called this function specifying the `barrier`. When the required number of threads have reached the `barrier` this function returns in all threads. It returns the constant `PTHREAD_BARRIER_SERIAL_THREAD` for a single (arbitrary) thread and zero for others.

Exercise 1

The given program (sor-pthreads.c) implements the **Red-Black Successive Over-Relaxation (SOR)** algorithm which is a method of solving partial differential equations. The algorithm iterates over all non-boundary blocks in an N by N grid and updates their values to the average of their neighbors. For example, in the grid shown in Figure 1, the value of block `A` is updated as follows: `val(A) = (val(B) + val(C) + val(D) + val(E))/4`. The algorithm typically continues until the change in all values between iterations is within a particular threshold.

The algorithm is parallelized by dividing the grid into roughly equal size bands of rows, assigning each band to a different thread (as shown in Figure 1). Then the grid is treated as a checkerboard with alternating red and black blocks. Each iteration is divided into two phases. In the first phase, only the red blocks are updated based on the values of their neighboring black blocks. Similarly, in the second phase, only the black blocks are updated based on the values of their neighboring red blocks. Note that each phase can proceed without any synchronization between the threads. But at the end of each phase, all threads must synchronize.

In the program provided to you, the algorithm only runs for a fixed number of iterations. Then the program prints the final sum of all blocks in the grid.

Compile the program: `make sor-pthreads`.

Execute the program: `./sor-pthreads -t <num_threads> -n <num_rows_cols>`
(default: num_threads = 4, num_rows_cols = 2000)

Is the result constant across multiple runs? Why not?

Now add barriers at appropriate places in the code to allow the threads to synchronize and to ensure that the algorithm works as expected. Compile and execute the program. Is the result constant across multiple runs now?

Figure 1: Red-black SOR