8 Assignment 6 [Assignment ID: `cpp_concurrency`]

8.1 Preamble (Please Read Carefully)

Before starting work on this assignment, it is **critically important** that you **carefully** read Section 1 (titled "General Information") which starts on page 1 of this document.

8.2 Topics Covered

This assignment covers material primarily related to the following: .

8.3 Problems — Part A

- 7.1 a b g i l m [data races]
- 7.10 [sequentially-consistent execution]
- 7.12 a b c [sequentially-consistent execution]

8.4 Problems — Part B

1. *Concurrent bounded queue class template (*`queue`*).* In this exercise, a class template called `queue` is developed that provides the functionality of a concurrent bounded queue. The `queue` class template has a single template parameter `T`, which corresponds to the type of the elements stored in the queue. The interface for the class template is given in Listing 11. (Note that the `queue` class template is in the `ra::concurrency` namespace.) All of the (template) code for the `queue` class template should be placed in the file `include/ra/queue.hpp`.

Listing 11: Interface for `queue` class template

```
namespace ra::concurrency {

    // Concurrent bounded queue class.
    template <class T>
    class queue
    {
    public:

        // The type of the elements stored in the queue.
        using value_type = T;

        // A type for the status of a queue operation.
        enum class status {
            success = 0,
            empty,
            full,
            closed,
        };

        // A queue is not default constructible.
        queue() = delete;

        // Constructs a queue with a maximum size of max_size.
        // The size must be greater than zero.
        queue(std::size_t max_size);

        // A queue is not movable or copyable.
        queue(const queue&) = delete;
        queue& operator=(const queue&) = delete;
        queue(queue&&) = delete;
        queue& operator=(queue&&) = delete;

        // Destroys the queue after first ensuring that the queue has
        // been closed.
        // Precondition: The queue should be empty.
        ~queue();

        // Inserts the value x at the end of the queue, blocking if
        // necessary.
        // If the queue is full, the thread will be blocked until the
        // queue insertion can be completed.
        // If the queue is closed, the function returns with a return
        // value of status::closed.
        // If the value x is successfully inserted on the queue, the
        // function returns status::success.
        // This function is thread safe.
        status push(value_type&& x);

        // Removes the value from the front of the queue and places it
        // in x, blocking if necessary.
        // If the queue is empty and not closed, the thread will be
        // blocked until a value can be removed from the queue.
        // If the queue is closed, the function will return a value of
        // status::closed.
        // If a value is successfully removed from the queue, the value
        // is placed in x and the function returns status::success.
        // This function is thread safe.
        status pop(value_type& x);

        // Closes the queue.
        // The queue is placed in the closed state.
        // The closed state prevents more items from being inserted
        // on the queue, but it does not clear the items that are
        // already on the queue.
        // Invoking this function on a closed queue has no effect.
        // This function is thread safe.
        void close();

        // Returns if the queue is currently full (i.e., the number of
        // elements in the queue equals the maximum queue size).
        // This function is not thread safe.
        bool is_full() const;

        // Returns if the queue is currently empty.
        // This function is not thread safe.
        bool is_empty() const;

        // Returns if the queue is closed (i.e., in the closed state).
        // This function is not thread safe.
        bool is_closed() const;

        // Returns the maximum number of elements that can be held in
        // the queue.
        // This function is not thread safe.
        std::size_t max_size() const;

    };

}
```

The `queue` class will need to utilize both a mutex and one or more condition variables. The following classes will be helpful for this exercise: `std::mutex` and `std::condition_variable`. Threads must block while waiting for events. That is, they must not wait by spinning in a loop. (Hence, the need for condition variables.)

Write a program called `test_queue` that tests the functionality of the `queue` class template. The source for this program should be placed in the file `app/test_queue.cpp`.

2. *Thread pool (*`thread_pool`*).* In this exercise, a class called `thread_pool` is developed that provides the functionality of a thread pool.

   The `thread_pool` class has the interface given in Listing 12. The source for this class should be placed in the following files:

   - `include/ra/thread_pool.hpp`. Code that is appropriate for a header file (e.g., interface specifications).
   - `lib/thread_pool.cpp`. Code that is not appropriate for a header file.

Listing 12: Interface for `thread_pool` class

```
namespace ra::concurrency {

    // Thread pool class.
    class thread_pool
    {
    public:

        // Creates a thread pool with the number of threads equal to the
        // hardware concurrency level (if known); otherwise the number of
        // threads is set to 2.
        thread_pool();

        // Creates a thread pool with num_threads threads.
        // Precondition: num_threads > 0
        thread_pool(std::size_t num_threads);

        // A thread pool is not copyable or movable.
        thread_pool(const thread_pool&) = delete;
        thread_pool& operator=(const thread_pool&) = delete;
        thread_pool(thread_pool&&) = delete;
        thread_pool& operator=(thread_pool&&) = delete;

        // Destroys a thread pool, shutting down the thread pool first
        // if necessary.
        ~thread_pool();

        // Enqueue a task for execution by the thread pool.
        // The task to be performed is the code associated with the
        // callable entity func.
        // Note: The rvalue reference parameter is intentional.
        // Precondition: The thread pool is not in the shutdown state.
        void schedule(std::function<void()>&& func);

        // Shuts down the thread pool.
        // This function does not return until all threads have
        // completed their tasks.
        // If the thread pool is already shutdown, this function has
        // no effect.
        // After the thread pool is shutdown, it can only be destroyed.
        void shutdown();

    };
}
```

None of the member functions of the `thread_pool` class should wait by spinning in a loop. In particular, `shutdown` must not wait (for all threads to complete their task) by spinning in a loop. (Condition variables should be used, as appropriate, to allow blocking waits.) The implementation of the `thread_pool` class should utilize the `queue` class template developed in Exercise 1 for queueing functionality.

Write a program called `test_thread_pool` that tests the `thread_pool` class. The source for this test program should be placed in the file `app/test_thread_pool.cpp`.

3. *Julia set generator (*`compute_julia_set`*).* In this exercise, a multithreaded algorithm is developed to compute an image representation of a Julia set (which is a type of fractal set). The interface to this algorithm is through a function template called `compute_julia_set`. Internally, this function utilizes a thread pool to perform the necessary computation.

   Consider a function f that maps $\mathbb{C}$ to $\mathbb{C}$ and is of the form

   $$f(z) = z^2 + c,$$

   where c is an arbitrary complex constant. Such a function can be used to define a type of fractal set known as a Julia set. The function f can be applied repeatedly to form a sequence $\{z_i\}$ as follows:

   $$z_n = \begin{cases} f(z_{n-1}) & n \geq 1 \\ 0 & n = 0. \end{cases}$$

   Define $\eta(z)$ as the smallest integer i for which $|z_i| > 2$ if such an integer exists and ∞ otherwise. Finally, define the function γ_m as

   $$\gamma_m(z) = \min\{\eta(z), m\}. \tag{9}$$

By sampling the function γ_m on a rectangular grid, we can obtain an image representation of a Julia set.

To compute an image representation of a Julia set, a function template called `compute_julia_set` must be provided. This function template has a single template parameter `Real`, which corresponds to the real-number type used for calculating the image. The template parameter `Real` can be chosen as any floating-point type (i.e., **`float`**, **`double`**, or **`long double`**). The interface for this function is given in Listing 13. The source for the `compute_julia_set` function template and its helper (template) code should be placed in the file `include/ra/julia_set.hpp`.

Listing 13: Interface for the `compute_julia_set` function template

```
namespace ra::fractal {
    template <class Real>
    void compute_julia_set(const std::complex<Real>& bottom_left,
      const std::complex<Real>& top_right, const std::complex<Real>& c,
      int max_iters, boost::multi_array<int, 2>& a, int num_threads);
}
```

The function template `compute_julia_set` computes the samples of the function γ_m (given by (9)) on a rectangular grid of width W and height H that corresponds to the region R in the complex plane with bottom-left corner `bottom_left` and top-right corner `top_right`. The quantities W and H are chosen to correspond to the number of rows and columns, respectively, in the array `a`. The quantities c and m are chosen as `c` and `max_iters`, respectively. The `thread_pool` class should be used to perform the computation using `num_threads` threads (which is an integer greater than or equal to 1). Each row of the array `a` should be computed using a separate task for the thread pool. That is, `schedule` method of `thread_pool` object should be invoked once for each row `a`. After the function returns, the elements of `a` are initialized to the samples of γ_m. The array elements `a[0][0]` and `a[`H` - 1][`W` - 1]` correspond to the bottom-left and top-right corners of R, respectively. The function template `compute_julia_set` guarantees that the elements of `a` to lie in the range 0 to `max_iters`.

A program called `test_julia_set` should be developed for testing the `compute_julia_set` function template. The source for this test program should be placed in the file `app/test_julia_set.cpp`. For testing purposes, it is probably quite helpful to be able to view the computed array as an image. This can be done relatively easily by writing the array to a file encoded in the grayscale PNM format. Then, the image stored in the file can be viewed with any software that supports the PNM format. For example, the ImageMagick software available via the `display` command on many UNIX-based systems can be used to view PNM images.

Consider the following choice of parameters for the `compute_julia_set` function:

- `width` and `height` are both 512;
- `bottom_left` is $(-1.25, -1.25)$;
- `top_right` is $(1.25, 1.25)$;
- `c` is $(0.37, -0.16)$; and
- `max_iters` is 255.

In this case, the computed result rendered as an 8-bits/sample image is shown in Figure 2.

Use the `chrono` functionality of the standard library to measure the amount of time required for the execution of the `compute_julia_set` function. Perform this measurement for the number of threads being used by this function chosen as 1, 2, 4, and 8. Comment on these measurements obtained. In particular, state whether the results obtained make sense and why. Include these comments in the `README.pdf` file for this assignment.

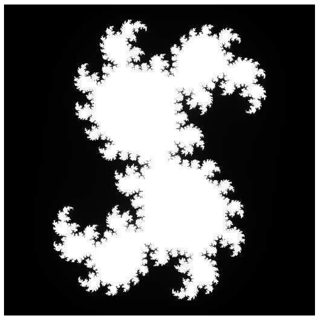

Figure 2: Julia set image.

The text-based PNM format for grayscale images is very simple. This format represents a grayscale image of width W and height H using a character sequence that consists of the following:

(a) A signature, which consists of the character “P” followed by the character “3”.
(b) A space character.
(c) The width W of the image, which consists of a sequence of one or more decimal digits.
(d) A space character.
(e) The height H of the image, which consists of a sequence of one or more decimal digits.
(f) A newline character.
(g) The maximum sample value for the image, which consists of a sequence of one or more decimal digits. For an image with n-bits/sample, this value would be $2^n - 1$ (e.g., 255 for an 8-bits/sample image).
(h) The image samples. For each of the H rows in the image starting with the top row, a sequence of W integers (on the same line) separated by space characters, followed by a newline character.

(Note that, in the PNM format, the sample data is encoded from top to bottom. The array holding the result produced by `compute_julia_set` is stored from bottom to top. So, the rows of the array must be encoded in reverse order.)