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 1-1 of this document.

8.2 Topics Covered

This assignment covers material primarily related to the following: concurrency, threads, mutexes, condition variables.

8.3 Problems — Part A — Nonprogramming Exercises

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

8.4 Problems — Part B — Thread Pool

B.1 *Concurrent bounded FIFO queue class template (*`queue`*).* In this exercise, a class template called `queue` is developed that provides the functionality of a concurrent bounded FIFO 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 FIFO queue class.
   template <class T>
   class queue
   {
   public:

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

      // An unsigned integral type used to represent sizes.
      using size_type = std::size_t;

      // A type for the status of a queue operation.
      enum class status {
         success = 0, // operation successful
         empty, // queue is empty (not currently used)
         full, // queue is full (not currently used)
         closed, // queue is closed
      };

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

      // Constructs a queue with a maximum size of max_size.
      // The queue is marked as open (i.e., not closed).
      // Precondition: The quantity max_size must be greater than
```

```
        // zero.
        queue(size_type 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 closing the queue (if not already
        // closed) and clearing the queue (if not already 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 or the queue is closed.
        // If the value x is successfully inserted on the queue, the
        // function returns status::success.
        // If the value x cannot be inserted on the queue (due to the
        // queue being closed), the function returns with a return
        // value of status::closed.
        // This function is thread safe.
        // Note: The rvalue reference parameter is intentional and
        // implies that the push function is permitted to change
        // the value of x (e.g., by moving from x).
        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 is blocked
        // until: 1) a value can be removed from the queue; or 2) the
        // queue is closed.
        // If the queue is closed, the function does not block and either
        // returns status::closed or status::success, depending on whether
        // a value can be successfully removed from the queue.
        // If a value is successfully removed from the queue, the value
        // is placed in x and the function returns status::success.
        // If a value cannot be successfully removed from the queue (due to
        // the queue being both empty and closed), the function returns
        // status::closed.
        // 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();

        // Clears the queue.
        // All of the elements on the queue are discarded.
        // This function is thread safe.
        void clear();

        // 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.
        size_type max_size() const;

    };

}
```

The `queue` class template employs a first-in first-out (FIFO) queuing policy. So, elements are always removed from the queue in the same order that they are placed in the queue.

At any given time, a queue is either marked as open or closed (i.e., not open). Normally, a queue is open. When a queue is created, it is always marked as open. A queue can be marked as closed by a close operation. After a queue is closed, no new elements may be pushed onto the queue. Any elements on a queue at the time of a close operation are unaffected (i.e., they remain on the queue). So, after a queue is closed, any elements remaining on the queue can still be popped. The threads using the `queue` class template may need a means to signal when the queue is no longer needed. The close operation effectively provides a means to gracefully shutdown a queue (by preventing any further elements from being placed on the queue).

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`, `std::unique_lock`, 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.) In order to simplify the code for the `queue` class template, it does not need to be exception safe.

For handling the storage of the underlying queue elements, the `queue` class template implementation may employ (non-concurrent) container classes (as well as container-adapter classes) from the standard library as appropriate.

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`.

B.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:

    // An unsigned integral type used to represent sizes.
    using size_type = std::size_t;

    // 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 not already shutdown).
    ~thread_pool();

    // Gets the number of threads in the thread pool.
    // This function is not thread safe.
    size_type size() const;

    // Enqueues a task for execution by the thread pool.
    // This function inserts the task specified by the callable
    // entity func into the queue of tasks associated with the
    // thread pool.
    // This function may block if the number of currently
    // queued tasks is sufficiently large.
    // Note: The rvalue reference parameter is intentional and
    // implies that the schedule function is permitted to change
    // the value of func (e.g., by moving from func).
    // Precondition: The thread pool is not in the shutdown state
    // and is not currently in the process of being shutdown via
    // the shutdown member function.
    // This function is thread safe.
    void schedule(std::function<void()>&& func);

    // Shuts down the thread pool.
    // This function places the thread pool into a state where
    // new tasks will no longer be accepted via the schedule
    // member function.
    // Then, the function blocks until all queued tasks
    // have been executed and all threads in the thread pool
    // are idle (i.e., not currently executing a task).
    // Finally, the thread pool is placed in the shutdown state.
    // If the thread pool is already shutdown at the time that this
    // function is called, this function has no effect.
    // After the thread pool is shutdown, it can only be destroyed.
    // This function is thread safe.
```

```
        void shutdown();

        // Tests if the thread pool has been shutdown.
        // This function is not thread safe.
        bool is_shutdown() const;

    };
}
```

None of the member functions of the `thread_pool` class should wait by spinning in a loop. For example, `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 B.1 for queuing functionality. The maximum size of the queue employed should be at least 32. Since the main objective of this exercise is to provide practice using mutexes and condition variables, you should not use other types of synchronization primitives in your implementation, such as latches and barriers (e.g., `std::latch` and `std::barrier`).

Since one of the main reasons for using a thread pool is to avoid the cost overhead associated with repeatedly creating and destroying threads, the thread pool must only create threads in the `thread_pool` constructor and must only destroy threads in the `thread_pool` destructor. That is, if a thread pool uses n threads, then during the entire lifetime of the thread pool, the thread pool is only allowed to employ n thread creation operations and n thread destruction operations, regardless of how many tasks are run by the thread pool.

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`.

8.5 Problems — Part C — Multithreaded Fractal Computation

C.1 *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. Given a point $z \in \mathbb{C}$, the function f can be applied repeatedly to form a sequence $\{z_i\}$ (starting with $i = 0$) as follows:

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

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.

For illustrative purposes, let us consider a simple example. In what follows, let j denote the quantity $\sqrt{-1}$. In particular, let us consider the Julia set associated with the complex constant $c = 1 + j$, and suppose that $m = 255$.

Suppose that one of the points z for which we wish to compute $\gamma_m(z)$ is $z = 0$. We have

$$\begin{aligned} z_0 &= 0, \\ z_1 &= f(z_0) = f(0) = 0^2 + (1+j) = 1+j, \quad \text{and} \\ z_2 &= f(z_1) = f(1+j) = (1+j)^2 + (1+j) = 1+3j. \end{aligned}$$

Since $|z_2| > 2$ (and it is not true that $|z_1| > 2$), we have that $\eta(0) = 2$ and consequently $\gamma_m(0) = \min\{2, 255\} = 2$. Suppose that another one of the points z for which we wish to compute $\gamma_m(z)$ is $z = j$. We have

$$\begin{aligned} z_0 &= j, \\ z_1 &= f(z_0) = f(j) = j^2 + (1+j) = j, \\ z_2 &= f(z_1) = f(j) = j^2 + (1+j) = j, \\ z_3 &= f(z_2) = f(j) = j^2 + (1+j) = j, \\ &\cdots \\ z_{255} &= f(z_{254}) = f(j) = j^2 + (1+j) = j. \end{aligned}$$

Since $|z_i| > 2$ is never true for any i, $\eta(j) = \infty$. Consequently, $\gamma_m(j) = \min\{\infty, 255\} = 255$. Note that as soon as $\eta(z)$ is known to be at least m, the value of γ_m is known to be m (since, by definition, γ_m can never exceed m). That is, the algorithm never needs to iterate indefinitely in order to determine γ_m. At most m iterations are ever required.

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 columns and rows, respectively, in the array `a` (as provided by the caller). 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, the `schedule` method of the `thread_pool` object should be invoked once for each row of `a`. Upon return from the `compute_julia_set` function, 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. That is, the value `a[`ℓ`][`k`]` is equal to $\gamma_m(z)$, where

$$z = \left[u_0 + \left(\tfrac{k}{W-1}\right)(v_0 - u_0)\right] + \sqrt{-1}\left[u_1 + \left(\tfrac{\ell}{H-1}\right)(v_1 - u_1)\right],$$

and u_0, u_1, v_0, and v_1 are the real part of `bottom_left`, imaginary part of `bottom_left`, real part of `top_right`, and imaginary part of `top_right`, respectively. The function template `compute_julia_set` guarantees that the elements of `a` to lie in the range 0 to `max_iters`.

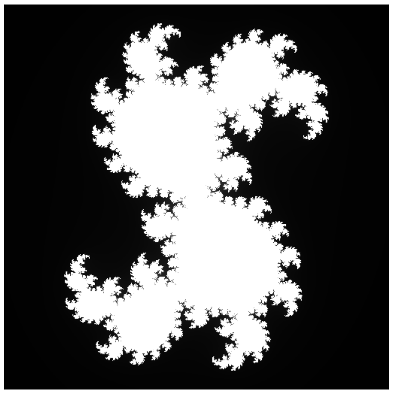

Figure 2: Julia set image.

Note that some examples of code using the `boost::multi_array` class template can be found in the Boost section of the lecture slides. (To find the relevant material, search for `multi_array` in the electronic version of the lecture slides.)

To simplify the code and make good use of the standard library, you are encouraged to use `std::complex` for the complex arithmetic associated with the Julia set computations.

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 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 (including the machines in the computer lab used for the course) can be used to view PNM images. To view an image stored in the file `image.pnm`, use a command like:

```
display image.pnm
```

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.

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 “2”.
(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 space 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) A newline character.
(i) 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.)

To further illustrate this image format, we now consider a simple example. Consider an image of width 9 and height 10 whose contents resemble a crude white letter "X" on a black background. This image would be encoded using the following sequence of characters:

```
P2 9 10 255
255 000 000 000 000 000 000 000 255
000 255 000 000 000 000 000 255 000
000 000 255 000 000 000 255 000 000
000 000 000 255 000 255 000 000 000
000 000 000 000 255 000 000 000 000
000 000 000 255 000 255 000 000 000
000 000 255 000 000 000 255 000 000
000 255 000 000 000 000 000 255 000
255 000 000 000 000 000 000 000 255
000 000 000 000 000 000 000 000 000
```

Note that the leading zeros used in the encoding of integer values are not required. They are only used so that the sample values for each column align vertically when the encoded character sequence is printed (i.e., to aid visualization).

8.6 Additional Remarks

On many platforms, the C++ Standard Library relies on an auxiliary library for threading support. On Unix-based systems, this auxiliary library is typically the POSIX Threads Library (i.e., the pthread library). In any case, in order to link against the appropriate threading library with CMake, the `Threads` package can be used. (An example of using the `Threads` package can be found in the CMake section of the lecture slide deck. To find the example, search for "threads" in that section.)