```
Name: Jude Onyia
Student ID: V00947095
Email: judeonyia10@gmail.com
Course: ECE596C
Section: T01

Assignment ID: cpp_concurrency
Assignment Title: Concurrency

Submission Source: https://github.com/uvic-seng475-2020-05/cpp_concurrency-JudeO
nyia.git
Commit ID: 223db394dec9c37c53f3285049e145999463bd1a

Submitted Files
===============

drwxrwxr-x      98 2020-07-29 00:12 ./app
-rw-rw-r--    1116 2020-07-29 00:12 ./app/test_julia_set.cpp
-rw-rw-r--    1993 2020-07-29 00:12 ./app/test_queue.cpp
-rw-rw-r--     498 2020-07-29 00:12 ./app/test_thread_pool.cpp
-rw-rw-r--     875 2020-07-29 00:12 ./CMakeLists.txt
-rw-rw-r--     145 2020-07-29 00:12 ./IDENTIFICATION.txt
drwxrwxr-x      24 2020-07-29 00:12 ./include
drwxrwxr-x      83 2020-07-29 00:12 ./include/ra
-rw-rw-r--    2017 2020-07-29 00:12 ./include/ra/julia_set.hpp
-rw-rw-r--    6000 2020-07-29 00:12 ./include/ra/queue.hpp
-rw-rw-r--    5012 2020-07-29 00:12 ./include/ra/thread_pool.hpp
drwxrwxr-x      37 2020-07-29 00:12 ./lib
-rw-rw-r--     561 2020-07-29 00:12 ./lib/thread_pool.cpp
-rw-rw-r--  887689 2020-07-29 00:12 ./README.pdf

Results
=======

Package          Operation Target           Status
nonprog          generate  ---              OK (0.1s)
tpool_orig       generate  ---              OK (0.5s)
tpool_orig       configure ---              OK (4.0s)
tpool_orig       build     test_queue       OK (1.3s)
tpool_orig       build     test_thread_pool OK (1.7s)
tpool_sane       generate  ---              OK (0.4s)
tpool_sane       configure ---              OK (3.8s)
tpool_sane       build     test_queue       OK (1.2s)
tpool_sane       build     test_thread_pool OK (1.9s)
fractal_orig     generate  ---              OK (0.4s)
fractal_orig     configure ---              OK (3.7s)
fractal_orig     build     test_julia_set   OK (3.8s)
fractal_sane     generate  ---              OK (0.7s)
fractal_sane     configure ---              OK (3.8s)
fractal_sane     build     test_julia_set   OK (3.3s)

Normally, an operation is indicated as having a status of either "OK" or
"FAIL".  A status of "?" indicates that the operation could not be performed
for some reason (e.g., due to an earlier error or being a manual step).
The time (in seconds) required for an operation is denoted by an expression
consisting of a number followed by the letter "s" (e.g., "5.0s").
In the case of a test that consists of multiple test cases, the number of
failed test cases and total number of test cases is expressed as a fraction
(e.g., "10/50" means 10 test cases failed out of 50 test cases in total).
```

```
The length (in lines) of the log file generated by an operation is denoted by
an expression consisting of a number followed by the letter "L" (e.g., "10L").
To ascertain the reason for the failure of an operation, check the contents
of the log file provided.

Legend
======

Package: nonprog
    Nonprogramming exercises

Package: tpool_orig
    The code as originally submitted by the student.
Build target: test_queue
    Build the test_queue program.
Build target: test_thread_pool
    Build the test_thread_pool program.

Package: tpool_sane
    Code with modifications to perform API sanity checking.
Build target: test_queue
    Build the (dummy) test_queue program.
Build target: test_thread_pool
    Build the (dummy) test_thread_pool program.

Package: fractal_orig
    The code as originally submitted by the student.
Build target: test_julia_set
    Build the test_julia_set program.

Package: fractal_sane
    Code with modifications to perform API sanity checking.
Build target: test_julia_set
    Build the (dummy) test_julia_set program.
```

```
commit 19d2b0001c79106add9efa3ecb5097509fe8653c
Author: JudeOnyia <60678029+JudeOnyia@users.noreply.github.com>
Date:   Mon Jul 27 00:54:28 2020 -0700

    First commit

commit f6e196dc3a5c67e55ddca3110ef2562ce1b8ca9d
Author: JudeOnyia <60678029+JudeOnyia@users.noreply.github.com>
Date:   Mon Jul 27 12:19:51 2020 -0700

    Added Sanitizers

commit 3f7ae0e44d6a141f9ef7fda32677fbbd4a243a85
Author: root <judeonyia10@gmail.com>
Date:   Mon Jul 27 19:42:58 2020 -0700

    1) Wrote function to push and pop elements in and from the queue
    2) Wrote function to clear and close the queue
    3) Wrote is_full, is_empty, is_closed and max_size functions

commit 023de44ae5f2f842985e89a2a34cf51390db511c
Author: JudeOnyia <60678029+JudeOnyia@users.noreply.github.com>
Date:   Mon Jul 27 20:24:48 2020 -0700

    1) Set up made for thread pool class

commit dcaa5f5de461c99086ecb690acfc88ccc1cc6186
Author: root <judeonyia10@gmail.com>
Date:   Tue Jul 28 02:28:57 2020 -0700

    1) Prevented some potential infinite waits in queue class
    2) Wrote the constructors and destructors of thread_pool class
    3) Wrote the friend function Worker to be used by threads
    4) Wrote the schedule function to put tasks on the queue
    5) Wrote the shutdown function
    6) Wrote the size and is_shutdown functions

commit 83961c676fb7b28c3eae3f0880e4a171919a9ce1
Author: root <judeonyia10@gmail.com>
Date:   Tue Jul 28 13:51:54 2020 -0700

    1) Checks if a thread is joinable before joining in threadpool
       destructor
    2) Made shutdown function blocking instead of schedule function

commit 61c2b12e95fe6fb755cf31ff04e1a802e9945d5d
Author: JudeOnyia <60678029+JudeOnyia@users.noreply.github.com>
Date:   Tue Jul 28 14:06:38 2020 -0700

    1) Set up for julia set

commit c1542b5b17394d01666d5a5f407176a68cfda637
Author: root <judeonyia10@gmail.com>
Date:   Tue Jul 28 20:59:57 2020 -0700

    1) Wrote helper function getZ to calculate z from the info given
    2) Wrote helper function julia_ym to calculate the ym term given z
    3) Wrote function to compute julia set
    4) Wrote test for function and visualized result

commit baa6932f2bbb41311db54bff27f6b5e5488d82f9
Author: root <judeonyia10@gmail.com>
```

```
Date:   Tue Jul 28 23:25:48 2020 -0700

    1) Used the chrono library to recored execution time of julia set program

commit 223db394dec9c37c53f3285049e145999463bd1a
Author: JudeOnyia <60678029+JudeOnyia@users.noreply.github.com>
Date:   Wed Jul 29 00:01:50 2020 -0700

    1) Recorded the execution time of julia set in README.pdf
    2) Removed Sanitizer.cmake
```

Name: Jude Onyia

Student ID: V00947095

Course: ECE 596C

Due Date: July 19, 2020

Assignment 6: Non – Programming Exercise

7.1 a)

This program creates two threads with shared variables x and y initially set to zero. Thread *t1* assigns the value 1 to x, then assigns the value 2 to y. While thread *t2* prints the value of y, then the value of x. Both threads are allowed to finish before the program ends. The program contains two possible data races: thread *t1* could writing to x at the same time that thread *t2* accesses x to print its value, and thread *t1* could be writing to y at the same time that thread *t2* accesses y to print its value. These two data races can be fixed by using mutexes to make the access of each variable mutually exclusive as shown below.

```
#include <iostream>
#include <thread>
#include <mutex>

int x = 0;
int y = 0;
std::mutex m_x;
std::mutex m_y;

int main()
{
	std::thread t1([ ] () {
		std::unique_lock lock_x(m_x, std::defer_lock);
		std::unique_lock lock_y(m_y, std::defer_lock);
		lock_x.lock();	x = 1;	lock_x.unlock();
		lock_y.lock();	y = 2;	lock_y.unlock();
	});
	std::thread t2([ ] () {
		std::unique_lock lock_x(m_x, std::defer_lock);
		std::unique_lock lock_y(m_y, std::defer_lock);
		lock_y.lock();	std::cout << y << " ";	lock_y.unlock();
		lock_x.lock();	std::cout << x << std::endl;	lock_x.unlock();
	});
	t1.join();
	t2.join();
}
```

Once these data races have been resolved, considering the SC-DRF memory model, there are six possible sequentially consistent orders that the program could follow, these are listed below along with the possible printed values of the variables. There is freedom for instructions A1 and B1 to occur concurrently. Also, instructions A2 and B2 can occur concurrently.

A1: x = 1, A2: y = 2, B1: std::cout << y << " ", B2: std::cout << x << std::endl

First instruction	Second instruction	Third instruction	Fourth instruction	Printed value of variable x at end of program	Printed value of variable y at end of program
A1	A2	B1	B2	1	2
A1	B1	A2	B2	1	0
A1	B1	B2	A2	1	0
B1	A1	A2	B2	1	0
B1	A1	B2	A2	1	0
B1	B2	A1	A2	0	0

7.1 b)

This program is similar to the previous, it creates two threads with shared variables x and y initially set to zero. Thread *t1* assigns the value 1 to x, then assigns the value 2 to y. While thread *t2* prints the value of y, then the value of x. Both threads are allowed to finish before the program ends. However, this program uses a single mutex to make the access to both variables mutually exclusive. This results in having no data races; however, this puts the entire code executed by each thread in a *happens-before* relationship. This causes only two possible sequentially consistent orders that the program could follow, as shown below. One case is that thread *t1* locks the mutex first, executes its code, and unlocks the mutex; thread *t2* can now lock the mutex, execute its code, and unlocks. The other case is vice-verse, thread *t2* locks and executes first, then thread *t1* follows. This forces the program to execute like it were a fully synchronous program.

A1: x = 1,   A2: y = 2,   B1: std::cout << y << " ",   B2: std::cout << x << std::endl

First instruction	Second instruction	Third instruction	Fourth instruction	Printed value of variable x at end of program	Printed value of variable y at end of program
A1	A2	B1	B2	1	2
B1	B2	A1	A2	0	0

7.1 g)

This program creates two threads with shared variables *x* initially set to zero and *done* initially set to *false*. Thread *t1* assigns 42 to *x*, then assigns *true* to *done*. Thread *t2* actively reads the *done* and does not stop until *done* is *true*, then *t2* uses an assertion to check if *x* is equals to 42. Both threads are allowed to finish before the program ends. With the way the program is written, there is only one possible data race that could occur: if *t1* is writing to *done* while *t2* is reading the state of *done*. This can be fixed by using a condition variable instead of actively checking *done* in *t2*. Due to the way the program is written, there are three possible sequentially consistent orders that the program could follow, as shown below. This is because B2 can not occur before A2, *t2* will continue to execute B1 until A2 has been executed. Therefore, the assert in B2 will always be true.

A1: x = 42,   A2: done = true,   B1: while (!done),   B2: assert(x == 42)

First instruction	Second instruction	Third instruction	Fourth instruction
A1	A2	B1	B2
A1	B1	A2	B2
B1	A1	A2	B2

7.1 i)

This program creates two threads with shared variables x and y initially set to zero. Thread *t1* checks if *x* is 1, and if it is, sets *y* to 1. Thread *t2* checks if y is 1, and if it is, sets *x* to 1. Both threads are allowed to finish before the program ends. Due to the way the program is written, there is no data race. There are only two possible sequentially consistent orders that the program could follow:

1. Thread *t1* checks the value of *x* in the if-statement which will result in *false*, thread *t2* checks the value of *y* in the if-statement which will result in *false* as well.
2. The vice-versa occurs where *t2* checks first, then *t1*.

7.1 l)

This program creates two threads with a shared variable *counter* initialized to zero. Each thread tries to increment *counter* 100,000 times. This will cause a data race when both threads try to write to *counter* at the same time. This can be fixed by making the access to *counter* mutually exclusive using a mutex. There is a wide number of ways in which the 100,000 incrementations of *counter* from each thread can be interleaved.

7.1 m)

This program creates two threads with a shared object *w* of class *Widget*. Objects of class *Widget* have two variables *x* and *y*, and two mutexes *xMutex* and *yMutex*. Thread *t1* locks *xMutex* and sets *x* of object w to 1. Thread *t2* locks *yMutex* and sets *y* of object w to 1. There are no data races in this program. Since the threads are locking different mutexes, line 18 and line 24 can execute concurrently. In a sequentially consistent order, either line 18 could be executed first, then line 24, or vice-versa.

7.10)

A1: $x = 1;$

A2: $a = y;$

B1: $y = 1;$

B2: $b = x;$

All possible sequentially consistent executions of this program

First instruction	Second instruction	Third instruction	Fourth instruction	Value of variable a	Value of variable b
A1	A2	B1	B2	0	1
A1	B1	A2	B2	1	1
A1	B1	B2	A2	1	1
B1	A1	A2	B2	1	1
B1	A1	B2	A2	1	1
B1	B2	A1	A2	1	0

There is one combination of that cannot be obtained for a and b: where $a = 0$ and $b = 0$.

7.12 a)

In this scenario, the **assertion in instruction B2** will be **true sometimes**. The only two possible sequentially consistent executions that reaches the line of the assertion and makes it true are {A1,A2,B1,B2} and {A1,B1,A2,B2}, shown in the table below. Where A1 executing before B1 allows the program to reach the assertion line and A2 executing before B2 makes the assertion true. The execution order that reaches the assertion line and makes it false is {A1,B1,B2,A2}, where A1 executing before B1 allows the program to reach the assertion line and B2 executing before A2 makes the assertion false. Other sequentially consistent execution orders do not reach the assertion line because B1 is executed before A1.

First instruction	Second instruction	Third instruction	Fourth instruction	State of assertion in B2
A1	A2	B1	B2	True
A1	B1	A2	B2	True
A1	B1	B2	A2	False
B1	A1	A2	B2	Not reached
B1	A1	B2	A2	Not reached
B1	B2	A1	A2	Not reached

7.12 b)

In this scenario, if the program can reach the **assertion in instruction B2**, the assertion will **always be true**. This is because there is only one possible sequentially consistent execution order that can reach the assertion: {A1,A2,B1,B2}, this is because A2 is executed before B1. In this case, since A1 must be executed before A2 and B1 must be executed before B2, this results in A1 executing before B2, causing the assert to be true. Other sequentially consistent execution orders do not reach the assertion line because B1 is executed before A2. Also, it is not possible for B2 to execute before A2 because thread 2 actively executes B1 until thread 1 executes A2, therefore, B2 cannot be executed until thread 2 stops executing B1 which happens after thread 1 executes A2. This case is marked in red.

First instruction	Second instruction	Third instruction	Fourth instruction	State of assertion in B2
A1	A2	B1	B2	True
A1	B1	A2	B2	Not reached
A1	B1	B2	A2	Not reached
B1	A1	A2	B2	Not reached
B1	A1	B2	A2	Not reached
B1	B2	A1	A2	Not reached

7.12 c)

In this scenario, the program reaches both assertions only in four possible sequentially consistent execution orders as shown in below, where A2 is executed before B1. The **assertion in instruction B2 will always be true** if the program reaches that assertion. This is because in those four possibilities, A1 is executed before B2. The **assertion in instruction B3 will be true sometimes**. Only if A3 is executed before B3, then the assertion is true, this is the case in 3 out of the 4 possibilities. In the fourth, the assertion is false because B3 is executed before A3. Also, it is not possible for B2 or B3 to execute before A2 because thread 2 actively executes B1 until thread 1 executes A2, therefore, B2 or B3 cannot be executed until thread 2 stops executing B1 which happens after thread 1 executes A2. This case is marked in red.

First	Second	Third	Fourth	Fifth	Sixth	State of assertion in B2	State of assertion in B3
A1	A2	A3	B1	B2	B3	True	True
A1	A2	B1	A3	B2	B3	True	True
A1	A2	B1	B2	A3	B3	True	True
A1	A2	B1	B2	B3	A3	True	False
A1	B1	A2	A3	B2	B3	Not reached	Not reached
A1	B1	A2	B2	A3	B3		
A1	B1	A2	B2	B3	A3		
A1	B1	B2	A2	A3	B3		
A1	B1	B2	A2	B3	A3		
A1	B1	B2	B3	A2	A3		
B1	A1	A2	A3	B2	B3		
B1	A1	A2	B2	A3	B3		
B1	A1	A2	B2	B3	A3		
B1	A1	B2	A2	A3	B3		
B1	A1	B2	A2	B3	A3		
B1	A1	B2	B3	A2	A3		
B1	B2	A1	A2	A3	B3		
B1	B2	A1	A2	B3	A3		
B1	B2	A1	B3	A2	A3		
B1	B2	B3	A1	A2	A3		

Part C Report and Comment on the Julia set program

The average amount of time required for the execution of the *compute_julia_set* function is recorded below. The results obtained are to be expected because the system that the program ran on utilizes an intel core i7-8565u processor which has 4 cores. Therefore, it makes sense that the fastest computation will typically occur with 4 threads running where the system can delegate each core to run one thread if the system is able to. Also, as expected, running the program with multiple threads improves performance in terms of speed compared to running the program with a single thread.

Threads	Time (seconds)
1	0.973576
2	0.502833
4	0.305151
8	0.378237

```
# Specify Minimum Required Version
cmake_minimum_required(VERSION 3.1 FATAL_ERROR)

# Specify Project and Language
project(cpp_concurrency LANGUAGES CXX)

# Require compliance with C++17
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED TRUE)

# Set Include Directory
include_directories(include)

# Find the threads library, indicating a preference for the pthread library
set(THREADS_PREFER_PTHREAD_FLAG true)
find_package(Threads REQUIRED)

# Add Executable Program
add_executable(test_queue app/test_queue.cpp)
add_executable(test_thread_pool app/test_thread_pool.cpp lib/thread_pool.cpp)
add_executable(test_julia_set app/test_julia_set.cpp lib/thread_pool.cpp)

# Set the libraries for the target
target_link_libraries(test_queue Threads::Threads)
target_link_libraries(test_thread_pool Threads::Threads)
target_link_libraries(test_julia_set Threads::Threads)
```

```cpp
#ifndef QUEUEHPP
#define QUEUEHPP
#include <cstddef>
#include <queue>
#include <utility>
#include <mutex>
#include <thread>
#include <condition_variable>

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) : capacity_(max_size), stat_(status::empty
) {}

            // 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(){
                close();
                clear();
            }

            // 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){
                std::unique_lock<std::mutex> lock(m_);
                c_full_.wait(lock, [this](){ return (stat_!= status::full);});
                if(stat_==status::closed){
                    return stat_;
                }
                else {
                    queue_.push(std::move(x));
                    if(queue_.size()==capacity_){
                        stat_ = status::full;
                    }
                    if(stat_==status::empty){
                        stat_ = status::success;
                    }
                    c_empty_.notify_one();
                    return (status::success);
                }
                //lock.unlock();
                //c_empty_.notify_one();
                //return (status::success);
            }

            // 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){
                std::unique_lock<std::mutex> lock(m_);
                c_empty_.wait(lock, [this](){ return (stat_!= status::empty);});
                if((stat_==status::closed) && (queue_.empty())){
                    return stat_;
                }
                else{
                    x = queue_.front();
                    queue_.pop();
                    if(stat_==status::full){
                        stat_ = status::success;
                    }
                    if((stat_!=status::closed) && (queue_.empty())){
                        stat_ = status::empty;
                    }
                    c_full_.notify_one();
                    return (status::success);
                }
                //lock.unlock();
                //c_full_.notify_one();
                //return (status::success);
```

```
		}

		// 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(){
			std::scoped_lock<std::mutex> lock(m_);
			if(stat_ != status::closed){
				stat_ = status::closed;
			}
			c_full_.notify_all();
			c_empty_.notify_all();
		}

		// Clears the queue.
		// All of the elements on the queue are discarded.
		// This function is thread safe.
		void clear(){
			std::scoped_lock<std::mutex> lock(m_);
			if(stat_ != status::empty || (!queue_.empty())){
				while(!(queue_.empty())){
					queue_.pop();
				}
				if(stat_ != status::closed){
					stat_ = status::empty;
					c_full_.notify_all();
				}
			}
		}

		// 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{
			//std::scoped_lock<std::mutex> lock(m_);
			return (queue_.size()==capacity_);
		}

		// Returns if the queue is currently empty.
		// This function is not thread safe.
		bool is_empty() const{
			//std::scoped_lock<std::mutex> lock(m_);
			return (queue_.empty());
		}

		// Returns if the queue is closed (i.e., in the closed state).
		// This function is not thread safe.
		bool is_closed() const{
			//std::scoped_lock<std::mutex> lock(m_);
			return (stat_==status::closed);
		}

		// Returns the maximum number of elements that can be held in
		// the queue.
		// This function is not thread safe.
		size_type max_size() const{
			//std::scoped_lock<std::mutex> lock(m_);
			return capacity_;
```

```
            }

        private:
            std::queue<value_type> queue_;
            mutable std::mutex m_;
            mutable std::condition_variable c_empty_;
            mutable std::condition_variable c_full_;

            size_type capacity_;
            status stat_;

    };




}
#endif
```

```
#include "ra/queue.hpp"
#include <cstddef>
#include <thread>
#include <mutex>
#include <iostream>
#include <utility>

using value_type = double;
using queue = ra::concurrency::queue<value_type>;
using status = ra::concurrency::queue<value_type>::status;
using queueForStat = ra::concurrency::queue<status>;
using std::cout;
using std::endl;

constexpr std::size_t num_elements(10);
queue q(2); //std::mutex m_q;
queue result(10);
queueForStat stat_store(30);

int main(){
    std::thread t1([](){
        //std::unique_lock<std::mutex> lock(m_q,std::defer_lock);
        for(std::size_t i=0; i<num_elements; ++i){
            //std::unique_lock<std::mutex> lock(m_q);
            status stat = q.push(value_type(i));// lock.unlock();
            stat_store.push(std::move(stat));
        }
        q.close();
    });
    std::thread t2([](){
        bool keep_running(true);
        //lstd::unique_lock<std::mutex> lock(m_q,std::defer_lock);
        //while(keep_running){
        for(std::size_t i=0; i<num_elements; ++i){
            //lstd::unique_lock<std::mutex> lock(m_q);
            //if((q.is_closed()) && (q.is_empty())){ keep_running = false; }// lock.unlock();
            value_type x;
            status stat = q.pop(x);
            stat_store.push(std::move(stat));
            x *= value_type(2);
            stat = result.push(std::move(x));
            stat_store.push(std::move(stat));
            //if((q.is_closed()) && (q.is_empty())){ keep_running = false; }// lock.unlock();
        }
    });

    t1.join(); t2.join();
    cout<<"Is q closed (should be true): "<<(q.is_closed())<<endl;
    value_type res(0);
    // Print out the results
    while(!result.is_empty()){
        result.pop(res);
        cout<<res<<" ";
    }
    cout<<endl;
    cout<<"Is stat_store full (should be true): "<<(stat_store.is_full())<<endl;
    cout<<"Max size of stat_store: "<<(stat_store.max_size())<<endl;
    //stat_store.clear();
    // Print out the status
    status stat(status::success);
```

```
    while(!stat_store.is_empty()){
        stat_store.pop(stat);
        if(stat==status::success){ cout<<"success"<<endl; }
        else if(stat==status::closed){ cout<<"closed"<<endl; }
        else { cout<<"Problem here"<<endl; }
    }

}
```

```
#ifndef THREADPOOLHPP
#define THREADPOOLHPP
#include "ra/queue.hpp"
#include <cstddef>
#include <vector>
#include <mutex>
#include <thread>
#include <condition_variable>
#include <functional>
#include <utility>


namespace ra::concurrency {

    // Thread pool class.
    class thread_pool
    {
        public:

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

        using func = std::function<void()>;
        using my_queue = typename ra::concurrency::queue<func>;

        // Function performed by each thread
        friend void worker(thread_pool*);
        /*void worker(){
            std::unique_lock<std::mutex> lock(m_);
            while(!(shutDown_ && queue_.is_empty())){
                c_task_.wait(lock, [this](){ return ((allThreadsMade_ && (!queue_.is_empty())) || (shutDown_));});
                if(!queue_.is_empty()){
                    func task_;
                    queue_.pop(task_);
                    lock.unlock();
                    c_add_.notify_one();
                    task_();
                    lock.lock();
                }
            }
            c_done_.notify_all();
            shutDownFinished_ = true;
        }*/

        // 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() : allThreadsMade_(false), shutDown_(false), shutDownFinished_(false), queue_(64) {
            size_type num_threads = std::thread::hardware_concurrency();
            if(num_threads == size_type(0)){ num_threads = size_type(2); }
            num_threads_ = num_threads;
            void worker(thread_pool*);
            for(size_type i=0; i<num_threads; ++i){
                workers.emplace_back(worker,this);
            }
            std::scoped_lock<std::mutex> lock(m_);
            allThreadsMade_ = true;
            c_task_.notify_one();
        }

```

```
        // Creates a thread pool with num_threads threads.
        // Precondition: num_threads > 0
        thread_pool(std::size_t num_threads) : allThreadsMade_(false), shutDown_(false), shutDownFinished_(false), queue_(64),num_threads_(num_threads) {
            void worker(thread_pool*);
            for(size_type i=0; i<num_threads; ++i){
                workers.emplace_back(worker,this);
            }
            std::scoped_lock<std::mutex> lock(m_);
            allThreadsMade_ = true;
            c_task_.notify_one();
        }

        // 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(){
            shutdown();
            for(auto& i : workers){
                if(i.joinable()){
                    i.join();
                }
            }
        }

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

        // 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()>&& funcc){
            std::unique_lock<std::mutex> lock(m_);
            /*if(shutDown_){
                c_done_.wait(lock, [this](){ return (queue_.is_empty()); });
            }
            else{*/
                c_add_.wait(lock, [this](){ return ((!queue_.is_full()) || (shutDown_)); });
                if(!shutDown_){
                    queue_.push(std::move(funcc));
                }
            //}
        }
```

```

        // 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(){
            std::unique_lock<std::mutex> lock(m_);
            if(!shutDown_){
                shutDown_ = true;
                queue_.close();
                c_task_.notify_all();
                c_done_.wait(lock, [this](){ return (queue_.is_empty()); });
            }
        }

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

        private:
        std::vector<std::thread> workers;
        mutable std::mutex m_;
        mutable std::condition_variable c_task_;
        mutable std::condition_variable c_done_;
        mutable std::condition_variable c_add_;

        my_queue queue_;
        bool allThreadsMade_;
        bool shutDown_;
        bool shutDownFinished_;
        size_type num_threads_;

    };




}
#endif
```

```cpp
#include "ra/thread_pool.hpp"

namespace ra::concurrency {
    void worker(thread_pool* obj){
        std::unique_lock<std::mutex> lock(obj->m_);
        while(!(obj->shutDown_ && obj->queue_.is_empty())){
            obj->c_task_.wait(lock, [obj](){ return ((obj->allThreadsMade_ && (!
obj->queue_.is_empty())) || (obj->shutDown_));});
            if(!obj->queue_.is_empty()){
                std::function<void()> task_;
                obj->queue_.pop(task_);
                lock.unlock();
                obj->c_add_.notify_one();
                task_();
                lock.lock();
            }
        }
        obj->shutDownFinished_ = true;
        obj->c_done_.notify_all();
    }

}
```

```cpp
#include "ra/thread_pool.hpp"
#include <iostream>
#include <cstddef>

using thr_pool = ra::concurrency::thread_pool;
using std::cout;
using std::endl;

int main(){
    thr_pool tp(8);
    for(std::size_t i=0; i<30; ++i){
        tp.schedule([](){cout<<"hello world"<<endl;});
    }
    tp.shutdown();
    //cout<<"Is shutdown (must be false): "<<(tp.is_shutdown())<<endl;
    //cout<<"Num of threads "<<(tp.size())<<endl;
    //while(!tp.is_shutdown()){}
    //cout<<"Is shutdown (must be true): "<<(tp.is_shutdown())<<endl;

}
```

```
#ifndef JULIASETHPP
#define JULIASETHPP
#include "ra/thread_pool.hpp"
#include <complex>
#include <boost/multi_array.hpp>
#include <cstddef>
#include <thread>
#include <mutex>
#include <functional>


namespace ra::fractal {
    using size_type = int;

    boost::multi_array<int, 2> a_; // shared pointer to matrix a
    std::mutex m_a; // Mutex used to make access to a_ptr mutually exclusive

    template <class Real>
    std::complex<Real> getZ(size_type l, size_type k, size_type W, size_type H, std::complex<Real> U, std::complex<Real> V){
        Real U0 = U.real();
        Real U1 = U.imag();
        Real V0 = V.real();
        Real V1 = V.imag();
        Real Z0 = U0 + ( (Real(k)/Real(W-1)) * (V0-U0) );
        Real Z1 = U1 + ( (Real(l)/Real(H-1)) * (V1-U1) );
        std::complex<Real> Z(Z0,Z1);
        return Z;
    }

    template <class Real>
    int julia_ym(int m, std::complex<Real> z, std::complex<Real> c){
        int i(0);
        while(!(  (i>=m) || (std::abs(z)>Real(2))   )){
            z = (z * z) + c;
            ++i;
        }
        if(i>=m) { return m; }
        else{ return i; }
    }


    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){
        size_type W = a.shape()[1]; // num of cols
        size_type H = a.shape()[0]; // num of rows
        a_.resize(boost::extents[a.shape()[0]][a.shape()[1]]);
        a_ = a;
        //a_ = a;
        ra::concurrency::thread_pool tp(num_threads); // Create Thread pool
        for(size_type l=0; l<H; ++l){
            //judeboost::multi_array<int, 2>* a_row_ptr = &(a[l][0]);
            tp.schedule([l, W, H, bottom_left, top_right, c, max_iters](){
                std::unique_lock lock(m_a, std::defer_lock);
                for(size_type k=0; k<W; ++k){
                    std::complex<Real> z = getZ(l,k,W,H,bottom_left, top_right);
                    int a_ym = julia_ym(max_iters,z,c);
                    lock.lock();
                    a_[l][k] = a_ym;
                    lock.unlock();
```

```
                        }
                    });
                }
                tp.shutdown();
                std::unique_lock lock(m_a);
                a.resize(boost::extents[a_.shape()[0]][a_.shape()[1]]);
                a = a_;
        }






}
#endif
```

```cpp
#include "ra/julia_set.hpp"
#include <complex>
#include <boost/multi_array.hpp>
#include <cstddef>
#include <iostream>
#include <fstream>
#include <chrono>

int main(){
    using Real = double;
    using std::cout;
    using std::endl;
    std::complex<Real> bottom_left(-1.25,-1.25);
    std::complex<Real> top_right(1.25,1.25);
    std::complex<Real> c(0.37,-0.16);
    int max_iters(255);
    int num_threads(4);
    int num_rows(512);
    int num_cols(512);
    boost::multi_array<int, 2> a(boost::extents[num_rows][num_cols]);
    auto start = std::chrono::steady_clock::now();
    ra::fractal::compute_julia_set(bottom_left,top_right,c,max_iters,a,num_threads);
    auto end = std::chrono::steady_clock::now();
    std::chrono::duration<Real> elapsed_time = end - start;
    cout<<"For "<<num_threads<<" threads, elapsed time: "<<(elapsed_time.count())<<" seconds "<<endl;

    std::ofstream outFile("image.pnm");
    outFile<<"P2 "<<num_cols<<" "<<num_rows<<" "<<max_iters<<"\n";
    for(int row(num_rows-1); row>=0; --row){
        for(int col(0); col<num_cols; ++col){
            if(col){ outFile<<" "; }
            outFile<<(a[row][col]);
        }
        outFile<<"\n";
    }
    outFile.close();


}
```