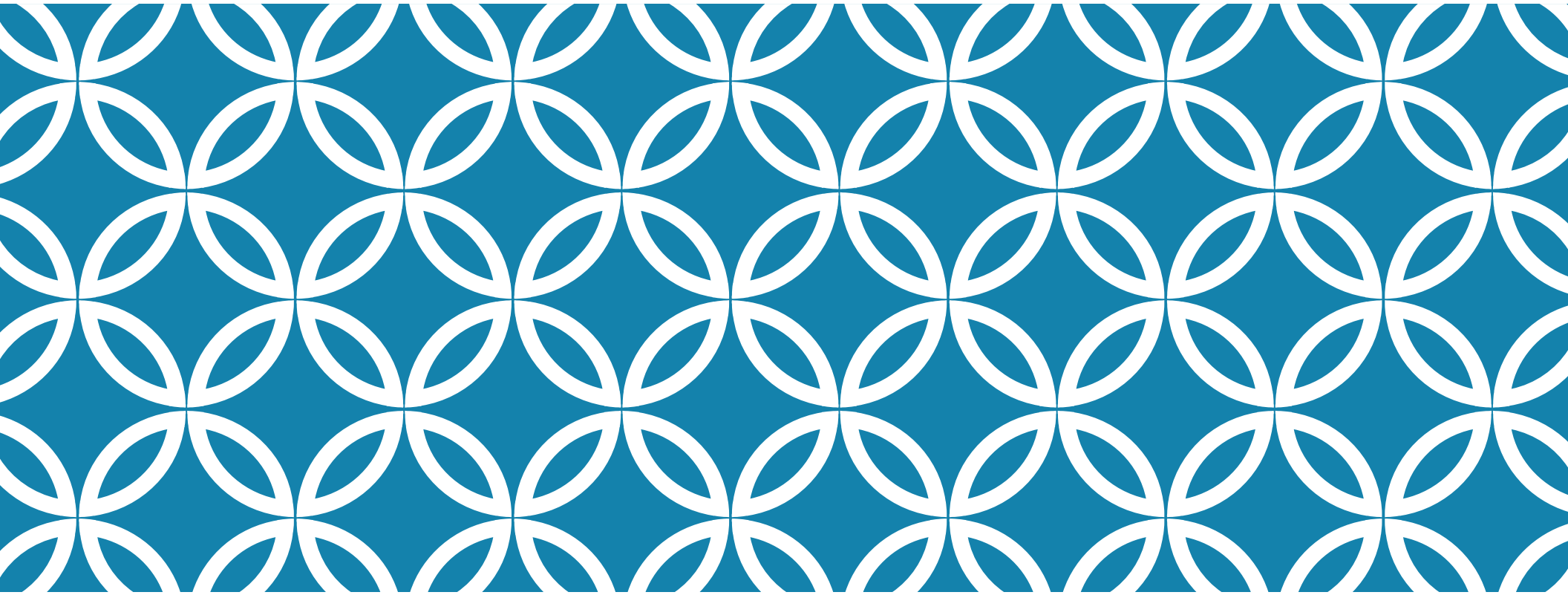


# CS597: CONCURRENCY AND ALGORITHMS

Data Management

Francis Joseph Serina

# LAUNCHING THREADS

Threads can work with **any callable** type

- Functions
- Functors
- Lambda Expressions

Functors would be **copied** into the thread and executed there

- Caller must guarantee behavior of copy is well-defined

Lambda Expressions could capture variables

# AFTER LAUNCHING

Join

- Wait for thread to finish

Detach

- Leave it running
- “Fire and Forget”
- Data passed should be valid until the thread is done with it

If neither is selected and std::thread object gets destroyed, your std::thread dtor calls std::terminate and forces your program to terminate

# PASSING DATA

All data passed would be **copied** and live in the local space of the thread.

- Be wary when passing references or pointers to the threads!

Global and Static Variables are accessible by all threads unless declared with **thread_local** - then the global or static variable would have separate instances per thread.

For objects that cannot be copied, they will be moved instead. Such as std::unique_ptr.

# TRANSFERRING OWNERSHIP OF THREADS

Threads are not copyable but they are movable.

# SHARING DATA

Data may need to be shared between caller and thread or between threads or both.

Threads cannot return values. Use out parameters instead.

Passing references to a thread require **std::ref**

Caller's responsibility to ensure the lifetime of the object being passed by std::ref to live throughout the duration of the thread's life.

Global and Static Variables would also work but less advisable

See Study03 demo

# SHARING DATA

Read-only data being shared has no adverse effects

Writeable data poses concerns

- Given 2 threads that try to increment the same data
- When they both read it, they get the same value. Each thread will increment it and store its value. When both threads are done with the data, they leave it incremented **only by one** instead of the expected 2

(Problematic) Race Conditions

- Multiple threads trying to modify data at the same time

# MUTEXES

Locking an operation before modification and unlocks it right after

Lock only the critical section, as fine-grained as possible

std::mutex

- A variable that can be read by all threads that is in either *locked* or *unlocked* state

std::lock_guard<std::mutex>

- A class that takes a mutex as parameter in the ctor and immediately locks it
- Uses RAII pattern to guarantee that the mutex is unlocked upon destruction

See Study02

# ALTERNATIVES TO MUTEXES

## Lock-Free Programming

- Design the data structure to not need locks

## Software Transactional Memory

- Transaction = set of instructions guaranteed to execute without interruption. Transaction failure would leave data untouched.
- Not implemented in C++
- Implemented in databases like SQL

# MUTEX WEAKNESS

Can only lock an operation

Mutex cannot protect data modified if it is exposed via a **reference** or **pointer**

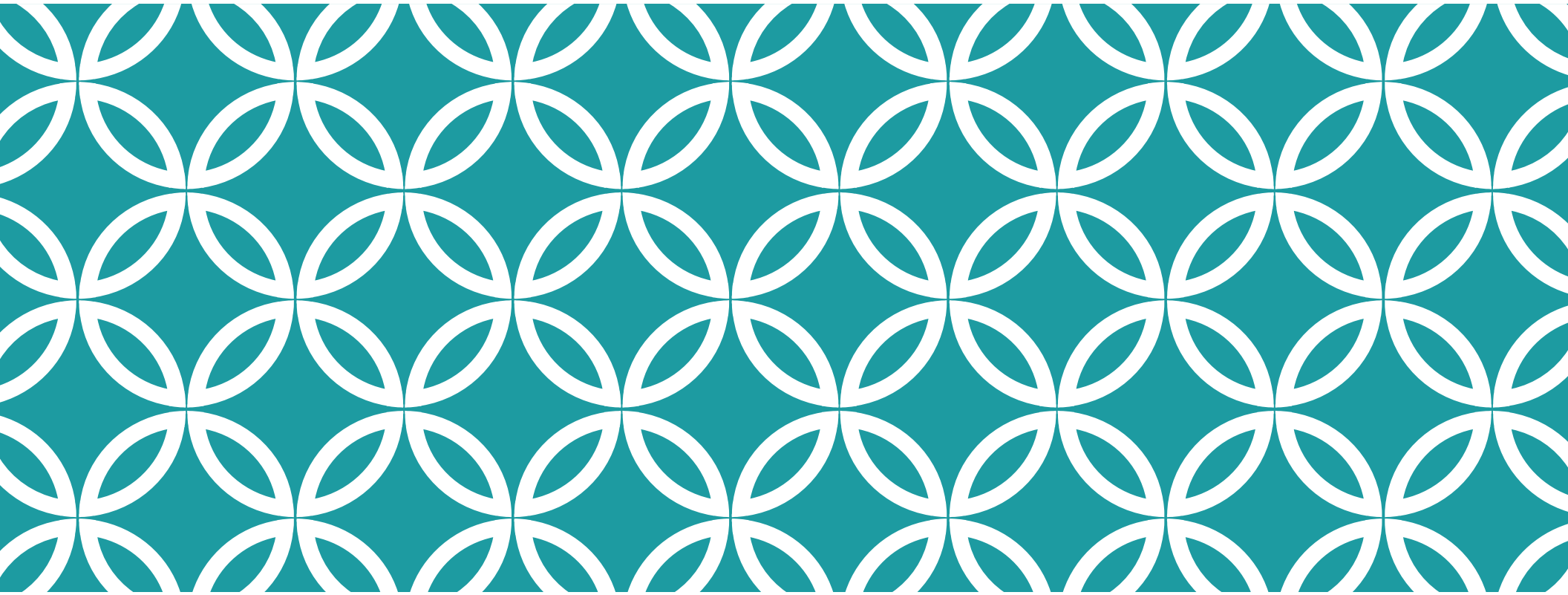
# MULTIPLE LOCKS

## Deadlock

- Two threads, each with their own lock, are waiting for the other thread to unlock preventing any progress

## Solutions

- Avoid nested locks
- Avoid calling user-supplied code while holding a lock
- Acquire locks in the same order for all threads
- Use a lock hierarchy



# PARALLEL FILE COPY

# OBJECTIVE

Transfer Multiple Files simultaneously

- std::experimental::filesystem (C++17)

See Study04

# OPTIMAL NUMBER OF THREADS

Prevent context switching within a hardware thread

**std::thread::hardware_concurrency()**

- Number of Hardware Threads in given machine
- Only a hint, may return 0 if the number is unknown

# OPTIMAL NUMBER OF THREADS

```
unsigned int GetOptimalNumberOfThreads(const unsigned int numTasks)
{
    auto hardwareThreadCount = std::thread::hardware_concurrency();
    if (hardwareThreadCount == 0)
    {
        hardwareThreadCount = 2;
    }
    return (numTasks > hardwareThreadCount) ? hardwareThreadCount : numTasks;
}
```

# DIVISION OF TASKS

Distribute tasks as evenly as possible between hardware threads

```
const unsigned int taskCount;
const auto numThreads = GetOptimalNumberOfThreads(taskCount);
const auto numTasksPerThread = taskCount / static_cast<float>(numThreads);
```

Note that the numTasksPerThread is a float. Rounding off is done during assignment.

# DIVISION OF TASKS

Ex: 11 tasks, 4 threads available

numTasksPerThread = 11/4 = 2.75

Thread 1 will take tasks 1 to 3 (round(2.75) = 3)

Thread 2 will take tasks 4 to 6 (round(2.75 * 2) = 6)

Thread 3 will take tasks 7 to 8 (round(2.75 * 3) = 8)

Thread 4 will take tasks 9 to 11 (round(2.75 * 4) = 11)

# DATA SHARED

Threads copy the list of assigned tasks (filenames to be copied) and destination path

All data from the caller guaranteed to outlive all the threads

# RESULTS

Data Set

- several binary files ranging from 3-8MB

Test 1: < 100MB total

- Single Thread: ~0.3s
- 8-Threads: ~0.15s
- **Time reduced by: 50%**
- Tried varying the number of threads but the gain was at most, 50%

Test 2: ~ 5GBs total

- Single Thread: ~102.9s
- 8-Threads: ~79.5s
- **Time reduced by: 20%**

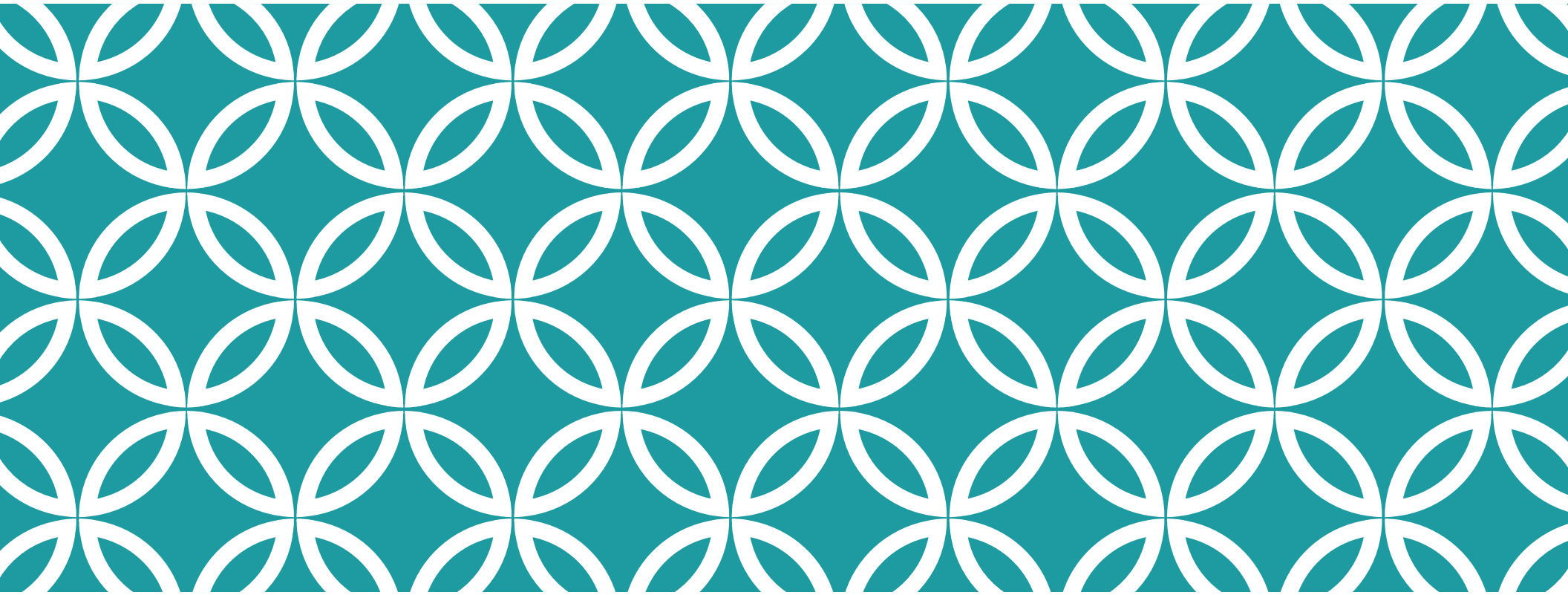
# OBSERVATIONS

Visual Studio Debugger showed

- High memory usage throughout (~180MB)
- CPU usage was minimal throughout

Conclusion

- Bottleneck *might be* happening at the Disk
- Parallel File Copy is a bad exercise for multi-threading



# CONVERT IMAGE TO GRAYSCALE

# OBJECTIVE

Convert an image, pixel-by-pixel, to grayscale on the CPU

- Lightweight image encoder/decoder (lodepng)

See Study05

# TASK DISTRIBUTION

Number of Tasks = height of image

- Task = convert pixels of 1 row

Used the same GetOptimalNumberOfThreads

# DATA SHARED

Image is read from disk to memory on main thread

Image is stored as std::vector<unsigned char>

Threads read/write unto the same image but only operate on their own pixel/row

# RESULTS

Data Set

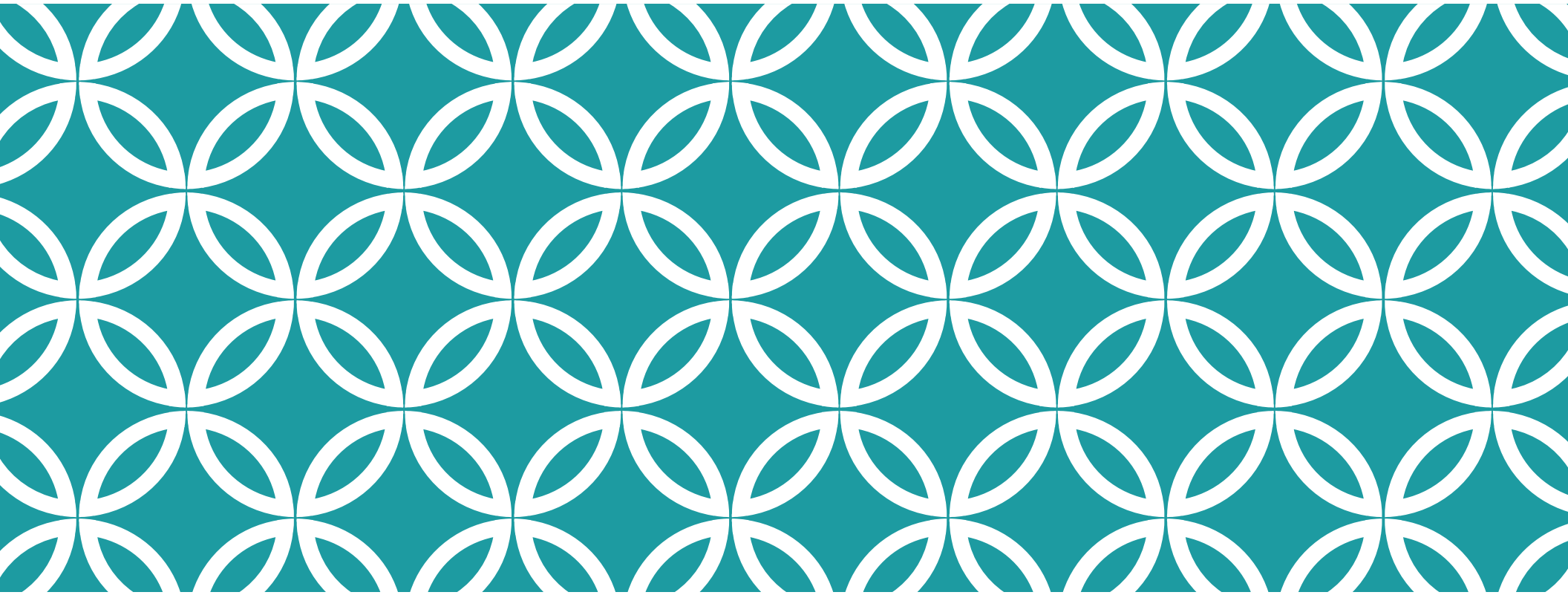
- 15MP image (courtesy of NASA)

Single Thread: ~10s

8-Threads: ~2.4s

**Time reduced by 75%**

Note: File I/O is all on main thread and is not counted towards computation of duration



# END