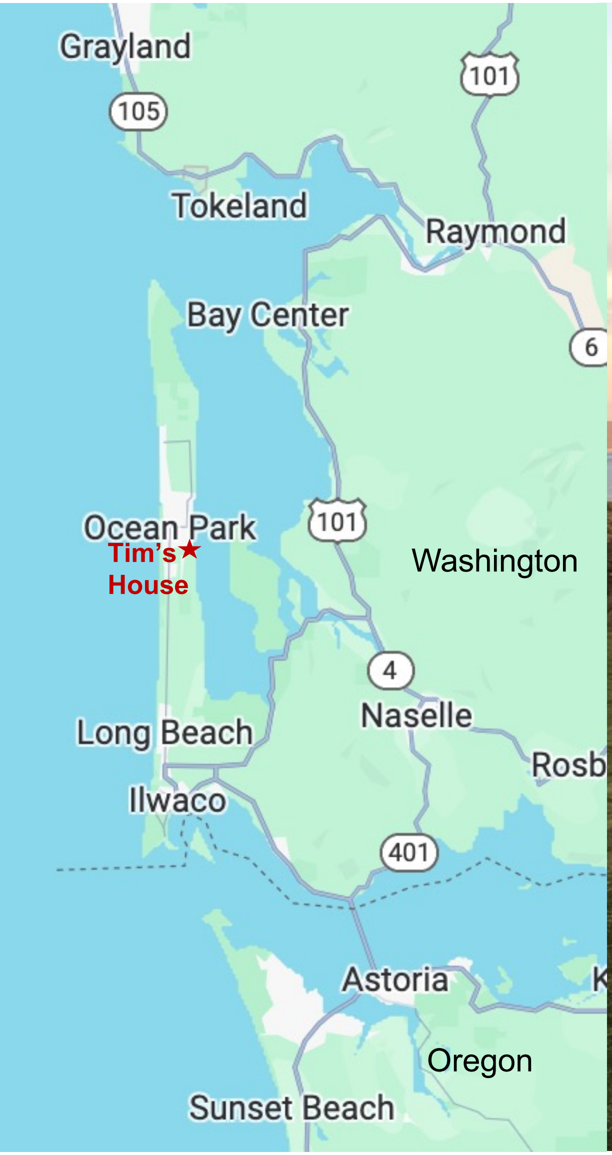
# An Introduction to multithreading in C++

**Tim Mattson**




Tim's Backyard around sunset

# Threads and C++

- Threads built into the C++ language (since C++11)
  - The inclusion of threads mandated the definition of a formal memory model in C++
  - Memory Models in Rust, OpenMP, and OpenCL are based on the C++ memory model.
- Expose the threading interface with: **#include <threads>**
- Provides the full range of capabilities needed for low-level thread management

**Thread Management***

- creation through constructors.
- Join: wait for a thread to finish
- Detach: thread runs independently from parent thread
- Swap: swap state of threads
- Asynchronous thread execution and futures

**Synchronization mechanisms***

- Mutex and locks,
- condition variables,
- semaphores,
- Barriers
- Atomic variables and flags
- Memory order control

*The capabilities are the most essential commonly used subset. For a complete list, go to: https://en.cppreference.com/w/cpp/thread/thread.html

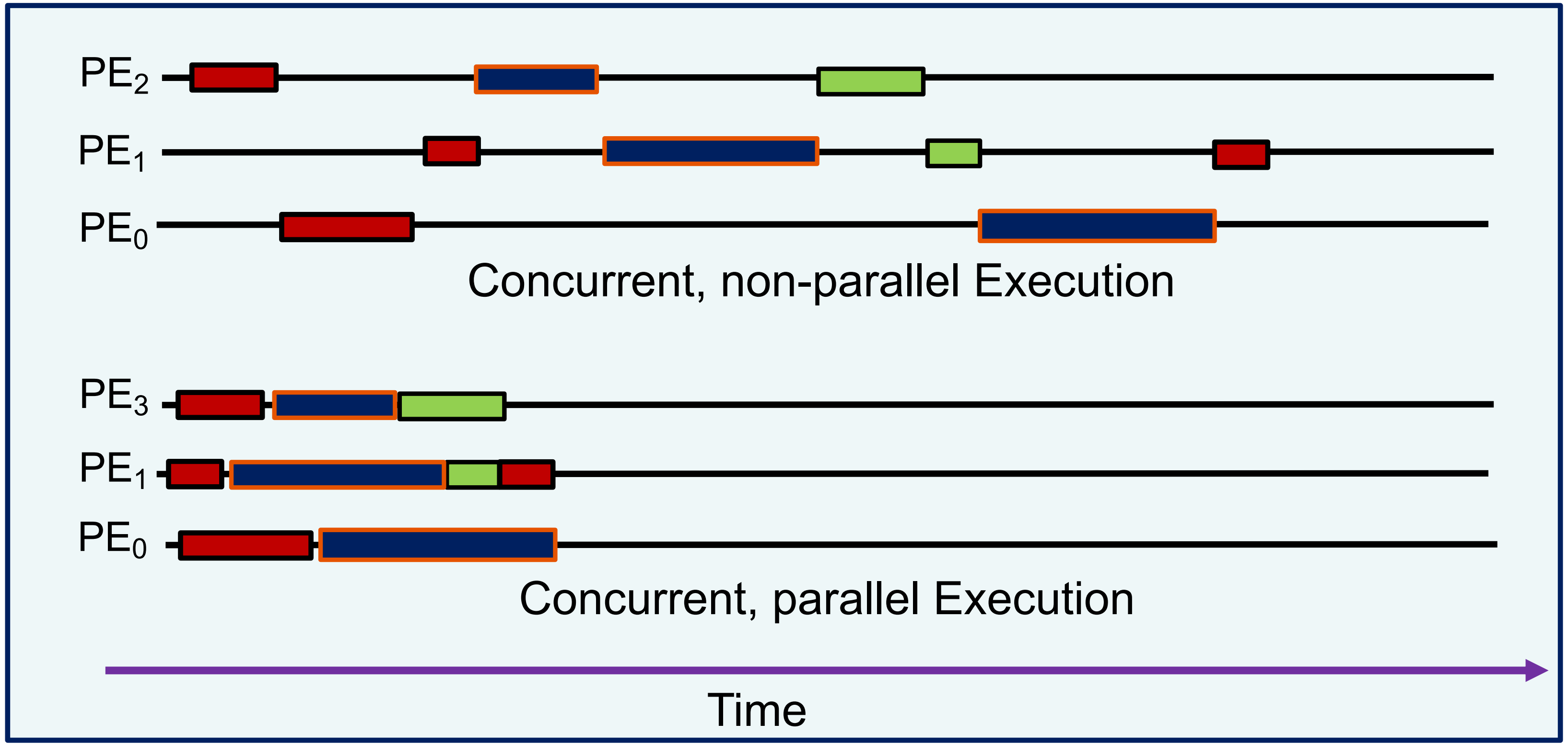
# Concurrency vs. Parallelism

The traditional definition in use since the 1970s

- Two important definitions:
  - Concurrency: A condition of a system in which multiple tasks are active and unordered. If **scheduled fairly**, they can be described as ***logically*** making **forward progress** at the same time.

C++ changed the definition to include fair scheduling … which is specific to CPUs.

  - Parallelism: A condition of a system in which multiple tasks are **actually** making **forward progress** at the same time.




PE = Processing Element

Figure from "An Introduction to Concurrency in Programming Languages" by J. Sottile, Timothy G. Mattson, and Craig E Rasmussen, 2010

# C++ threads: Hello World program (hello.cpp)

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


int main() {

   auto f = [ ] (int i){
     std::cout << "Hello ";
     std::cout << "world from thread " << i << std::endl;
   };

   std::thread t0(f,0);

   t0.join();

}
```

Constructor forks a thread running the lambda function f with the argument i=0

Wait for thread t0 to complete.

Destructor for thread t0 invoked as the thread variable t0 goes out of scope.

```
% g++ hello.cpp
% ./a.out
Hello world from thread 0
%
```

# C++ threads: Hello World program (hello.cpp)

```cpp
#include <thread>
#include <iostream>


int main() {

   auto f = [ ] (int i){
     std::cout << "Hello ";
     std::cout << "world from thread " << i << std::endl;
   };

   std::thread t0(f,0);




}
```

Constructor forks a thread running the lambda function f with the argument i=0

Destructor for thread t0 invoked as the thread variable t0 goes out of scope.
If thread status vague due to the lack of a join or a detach, it throws an exception

What happens if you forget to join or detach thread t0 before its destructor is called?

```
% g++ hello.cpp
% ./a.out
Hello world from thread 0
libc++abi: terminating
zsh: abort      ./a.out
%
```

# std::thread behavior violates the RAII principle

**RAII**: Resource Acquisition Is Initialization. Resources are tied to the lifecycle of an object* …

- The constructor acquires the resource and establishes all class invariants or throws an exception if that cannot be done.
- The destructor releases the resource and never throws exceptions.

## C++ threads: Hello World program (hello.cpp)

What happens if you forget to join or detach thread t0 before its destructor is called?

```cpp
#include <thread>
#include <iostream>


int main() {

   auto f = [ ] (int i){
     std::cout << "Hello ";
     std::cout << "world from thread " << i << std::endl;
   };

   std::thread t0(f,0);



}
```

```
% g++ hello.cpp
% ./a.out
Hello world from thread 0
libc++abi: terminating
zsh: abort      ./a.out
%
```

Constructor forks a thread running the lambda function f with the argument i=0

Destructor for thread t0 invoked as the thread variable t0 goes out of scope.
If thread status vague to do a lack of a join or a detach, it throws an exception

* source: https://en.cppreference.com/w/cpp/language/raii.html

# C++20 and jthreads (joining threads) to the rescue

- **std::jthread** is a wrapper class added in C++20 and included in **<thread>**.

- **std::jthread** works the same as the original threads from C++11 but …
  - A std::jthread will **join** by default … no exception is thrown if the destructor is called as the thread goes out of scope without prior invocation of join or detach.
  - A capability to control how threads can be cancelled/stopped (which we will not discuss)

- Since **std::jthread** was added with C++20, you must tell the compiler to follow the C++20 language specification … for example to compile the file hello.cpp

```
g++ -std=c++20 hello.cpp
```

# C++ threads: Hello World program (hello.cpp)

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


int main() {

  auto f = [ ] (int i){
    std::cout << "Hello ";
    std::cout << "world from thread " << i << std::endl;
  };

  std::jthread t0(f,0);



}
```

Constructor forks a thread running the lambda function f with the argument i=0.

Join invoked by default when function completes

Destructor for thread t0 invoked as the thread variable t0 goes out of scope.

What happens if you forget to join or detach thread t0 before its destructor is called?

```
% g++ -std=c++20 hello.cpp
% ./a.out
Hello world from thread 0
%
```

Nothing bad happens if you use std::jthread

# C++ threads: the multithread Hello World program

```
#include <thread>
#include <vector>
#include <iostream>

int main() {
   auto f = [] (int i){
     std::cout << "Hello world from thread ”  <<  i << std::endl;
   };

   auto nthreads = std::thread::hardware_concurrency();
   std::cout << nthreads << " HW threads available "<< std::endl;

   std::vector<std::thread> threadVec;

   for (auto i = 0; i<nthreads; i++){
      threadVec.emplace_back(f,i) ;
   }

   for (auto& thrd : threadVec){
        thrd.join();
   }
}
```

How do we write a program that uses multiple threads?

Returns the number of concurrent threads the implementation supports

A vector container of std::thread handles

Appends an object to the end of the container and invokes the constructor using the provided arguments

Loop over threadVec container invoking join on each thread

# C++ threads: the multithread Hello World program

```
#include <thread>
#include <vector>
#include <iostream>

int main() {
   auto f = [] (int i){
     std::cout << "Hello world from thread ”  <<  i << std::endl;
   };

   auto nthreads = std::thread::hardware_concurrency();
   std::cout << nthreads << " HW threads available "<< std::endl;

   std::vector<std::jthread> threadVec;

   for (auto i = 0; i<nthreads; i++){
      threadVec.emplace_back(f,i) ;
   }

}
```

Or with jthreads …

Returns the number of concurrent threads the implementation supports

A vector container of std::thread handles

Appends an object to the end of the container and invokes the constructor using the provided arguments

An explicit join is not needed

# Synchronization: when you need to order events in concurrent programs

- Creating threads that run concurrently is easy.

- Writing a program that is correct regardless of how the instructions from those threads are allowed to interleave … can be brutally difficult.

- Synchronization (ordering events between threads) is the key to safe execution of multiple threads. There are many ways to synchronize threads.

  - Mutexes and locks,
  - condition variables,
  - semaphores,
  - Barriers
  - Atomic variables and flags
  - Memory order control

- We will cover only one case … a mutex exposed through a scoped lock
  - **mutex**: Supports mutual exclusion. A thread holds a lock on a mutex object and all other threads trying to access the object will wait until it has been released.
  - **scoped_lock**: wraps a mutex for RAII style management … constructor acquires the mutex which is released when the scoped_lock object goes out of scope.

*The capabilities are the most essential commonly used subset. For a complete list, go to: https://en.cppreference.com/w/cpp/thread/thread.html

# Synchronization: when you need to order events in concurrent programs

- Creating threads that run concurrently is easy.

- Writing a program that is correct regardless of how the instructions from those threads are allowed to interleave … can be brutally difficult.

- Synchronization (ordering events between threads) is the key to safe execution of multiple threads. There are many ways to synchronize threads.

  - Mutexes and locks,
  - condition variables,
  - semaphores,
  - Barriers
  - Atomic variables and flags
  - Memory order control

OpenMP only includes a subset of these synchronization mechanisms

- We will cover only one cas … a scoped lock
  - **mutex**: Supports mutual exclusion. A thread holds a lock on a mutex object and all other threads trying to access the object will wait until it has been released.
  - **scoped_lock**: wraps a mutex for RAII style management … constructor acquires the mutex which is released when the scoped_lock object goes out of scope.

*The capabilities are the most essential commonly used subset. For a complete list, go to: https://en.cppreference.com/w/cpp/thread/thread.html

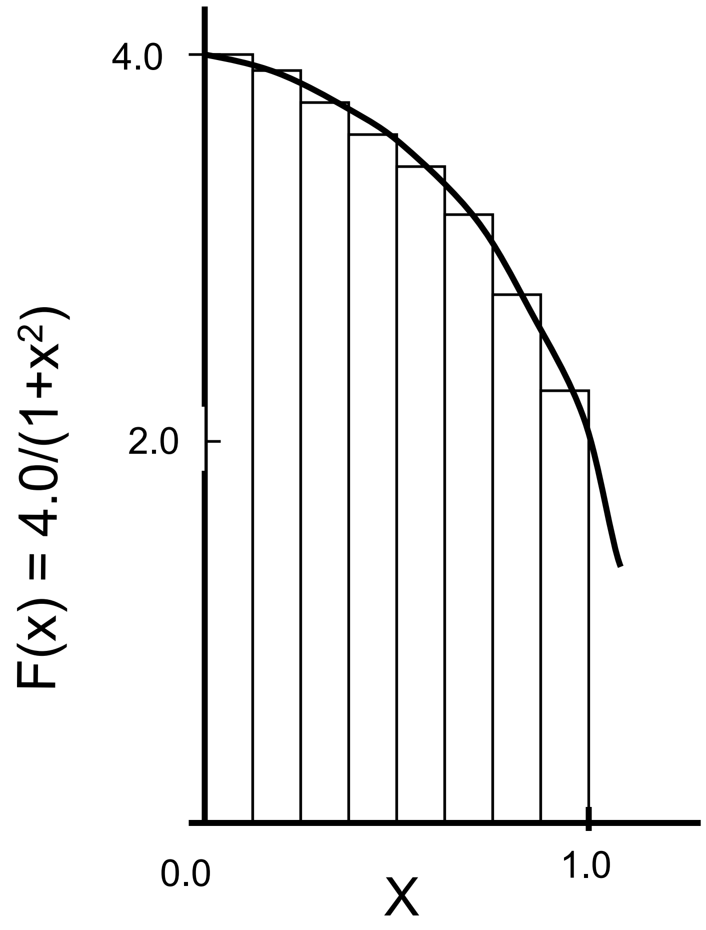
# Scoped Locks with Mutex for mutual exclusion

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

std::mutex critical;
int count = 0.0;
int big_calc();  // function not shown

int main() {
  std::vector<std::jthread> threadVec;
  auto f = [] (){
      int test = big_calc();
      {
         std::scoped_lock myLock(critical);
         if (test>0) count++;
      }
   };
   auto nthreads = std::thread::hardware_concurrency();
   for (auto i = 0; i<nthreads; i++){
      threadVec.emplace_back(f);
   }
   std::cout << count << " postive cases." << std::endl;
}
```

Create a mutex object in a scope that makes it visible to all threads that need to use the mutex

Create a scoped_lock object passing a mutex to it's constructor.  Lock resources released when the scoped_lock object goes our of scope.

# Exercise:  Parallel Pi program with C++ threads




Create a parallel version of the pi program using C++ threads.

```
static long num_steps = 100000;
double step;
int main ()
{           double x, pi, sum = 0.0;

            step = 1.0/(double) num_steps;

            for (int i=0;i< num_steps; i++){
                        x = (i+0.5)*step;
                        sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

How does the performance of the program based on C++ threads compare to the OpenMP program?

# Conclusion

- C++11 std::threads provides the full set of capabilities needed for multithreaded programming … including the low level primitives for advanced synchronization, building runtimes for multithreading, and concurrent data structures.

- This contrasts with OpenMP which only supports the subset of synchronization operations that applications programmers require.

- C++20 addressed a key flaw in std::threads that violated RAII and didn't let threads stop their own (and each other's) execution
  - std::jthreads
    - std::jthreads calls join in its destructor if join or detach was not invoked before.
    - Stop tokens to control ways to stop thread execution: get_stop_source, get_stop_token, request_stop.

**RAII**: Resource Acquisition Is Initialization. Resources are tied to the lifecycle of an object