Parallelism in C++

Lectures on Modern Scientific Programming 2016

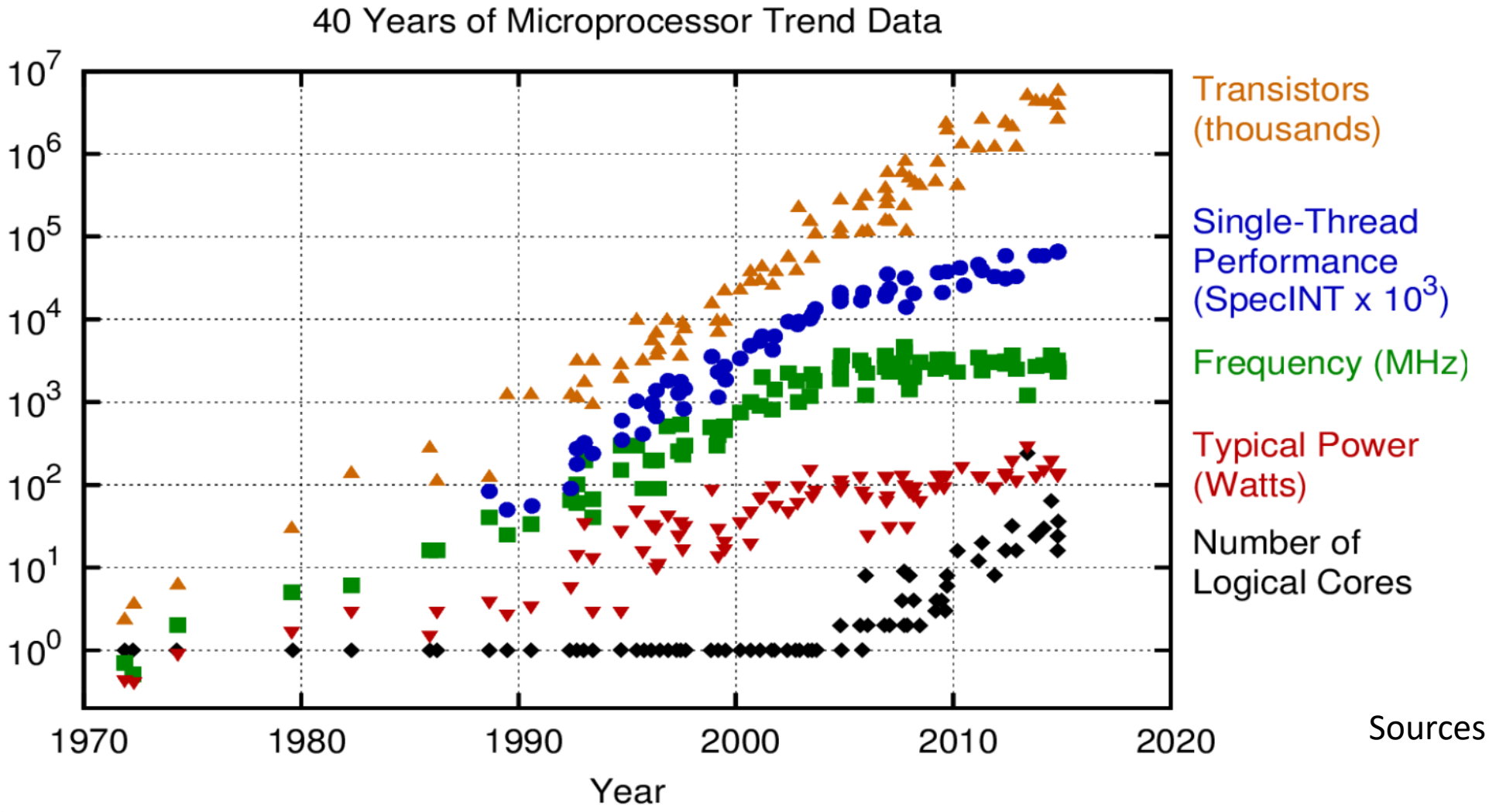
Dániel Berényi
Wigner GPU Lab

Outline

- This talk introduces the fundamental parallel programming constructs that became standardized with C++11

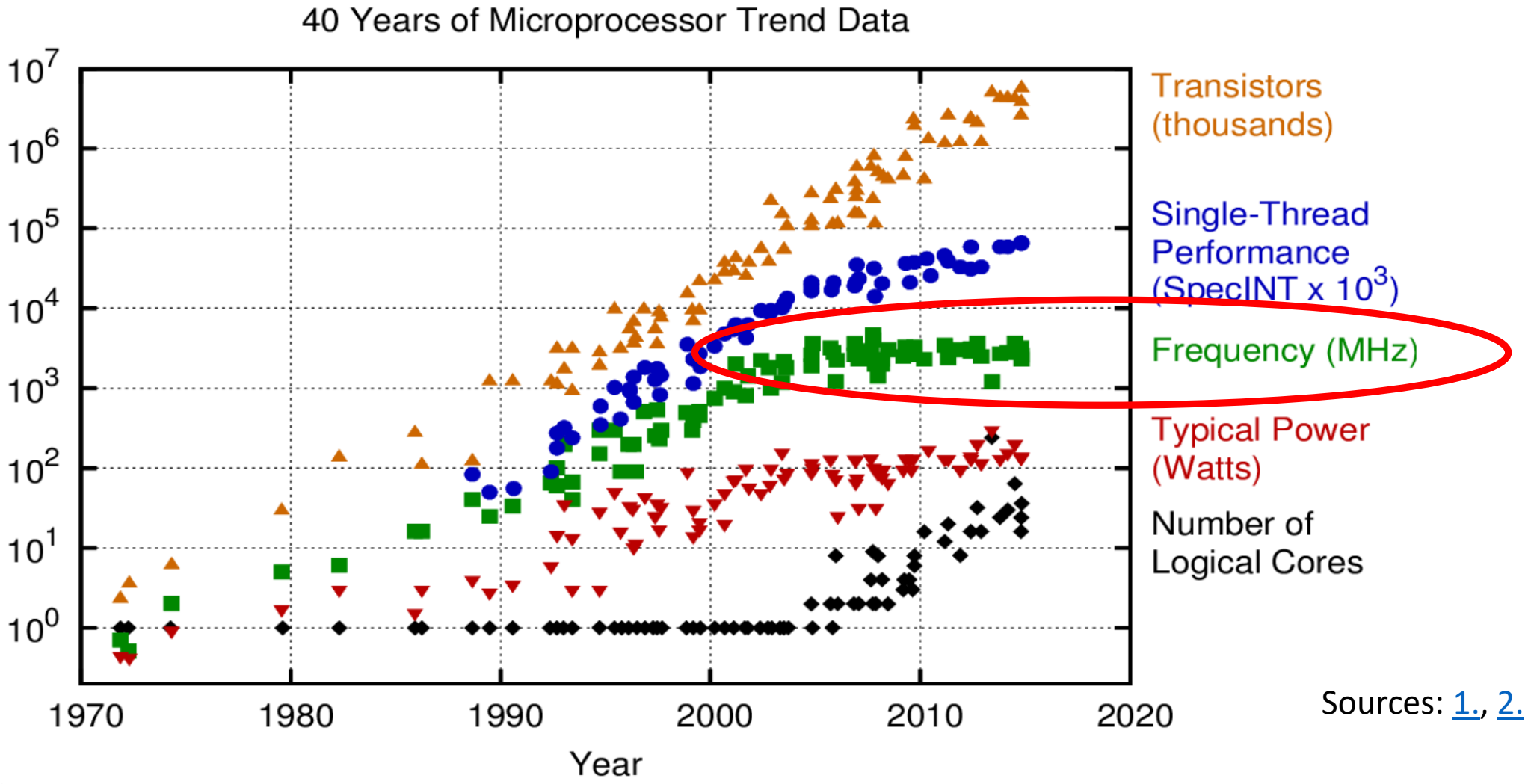
- So these constructs are now portable and easily accessible

Why do we need parallelism?



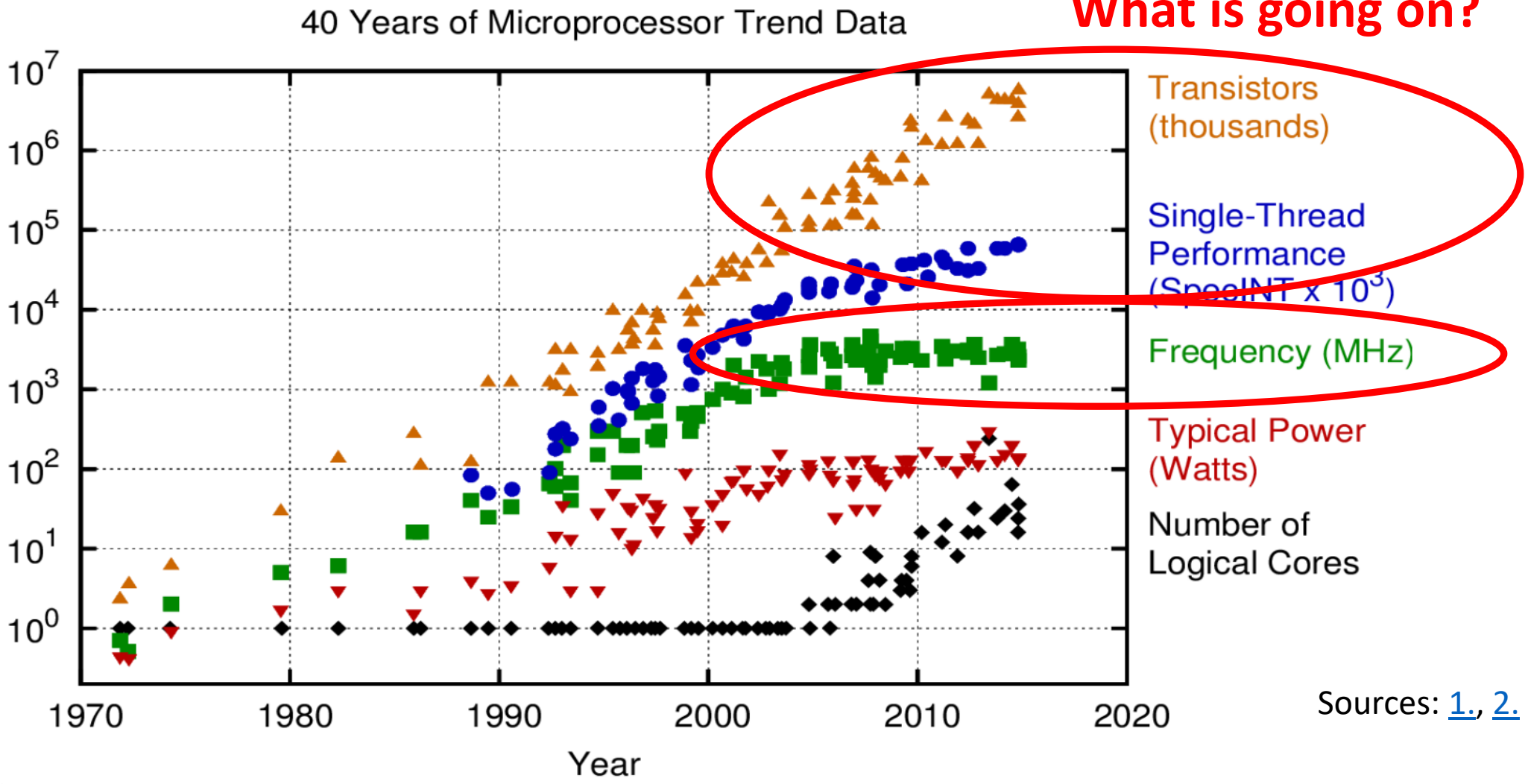
Sources: 1., 2.

Why do we need parallelism?



Sources: 1., 2.

Why do we need parallelism?



Sources: 1., 2.

Why do we need parallelism?

- Before 2000's computers become faster because the clock frequency increased every year.
- It hit a certain threshold because of cooling, so it converged to approximately 1-3 GHz.

- The only way to increase computing power is to parallelize...
- Additional transistors are used to duplicate execution units to run more and more tasks in parallel

Levels of Parallelism

- There are multiple levels of parallelism available:
  - Bit-level
  - Instruction-level
  - Vector-level
  - Task/Device-level
  - Process/Cluster-level

Levels of Parallelism

- There are multiple levels of parallelism available:
  - Bit-level  Multiple bits inside one register☺
  - Instruction-level
  - Vector-level
  - Task/Device-level
  - Process/Cluster-level

Levels of Parallelism

- There are multiple levels of parallelism available:
  - Bit-level
  - Instruction-level
  - Vector-level
  - Task/Device-level
  - Process/Cluster-level

Multiple instructions executing at the same time in the pipeline

Levels of Parallelism

- There are multiple levels of parallelism available:
  - Bit-level
  - Instruction-level
  - Vector-level
  - Task/Device-level
  - Process/Cluster-level

The same instruction operates on multiple data at the same time

Levels of Parallelism

- There are multiple levels of parallelism available:
  - Bit-level
  - Instruction-level
  - Vector-level
  - Task/Device-level
  - Process/Cluster-level

Multiple threads are running at the same time, but they share in some way a common memory

Levels of Parallelism

- There are multiple levels of parallelism available:
  - Bit-level
  - Instruction-level
  - Vector-level
  - Task/Device-level
  - Process/Cluster-level

Multiple processes are running on the same or different computers, no direct memory access (just message passing)

Levels of Parallelism

- There are multiple levels of parallelism available:
  - Bit-level
  - Instruction-level
  - Vector-level
  - Task/Device-level
  - Process/Cluster-level

Taken care by the processor and the compiler

In some edge cases you might want to tweak at the vector level

Levels of Parallelism

- There are multiple levels of parallelism available:
  - Bit-level
  - Instruction-level
  - Vector-level
  - Task/Device-level
  - Process/Cluster-level

It is the programmer's task to write the parallel logic

Levels of Parallelism

- There are multiple levels of parallelism available:
  - Bit-level
  - Instruction-level
  - Vector-level
  - Task/Device-level
  - Process/Cluster-level

If you don't use this, you are wasting resources, and your code will never be faster

Levels of Parallelism

- There are multiple levels of parallelism available:
  - Bit-level
  - Instruction-level
  - Vector-level
  - Task/Device-level This talk deals with this level!
  - Process/Cluster-level

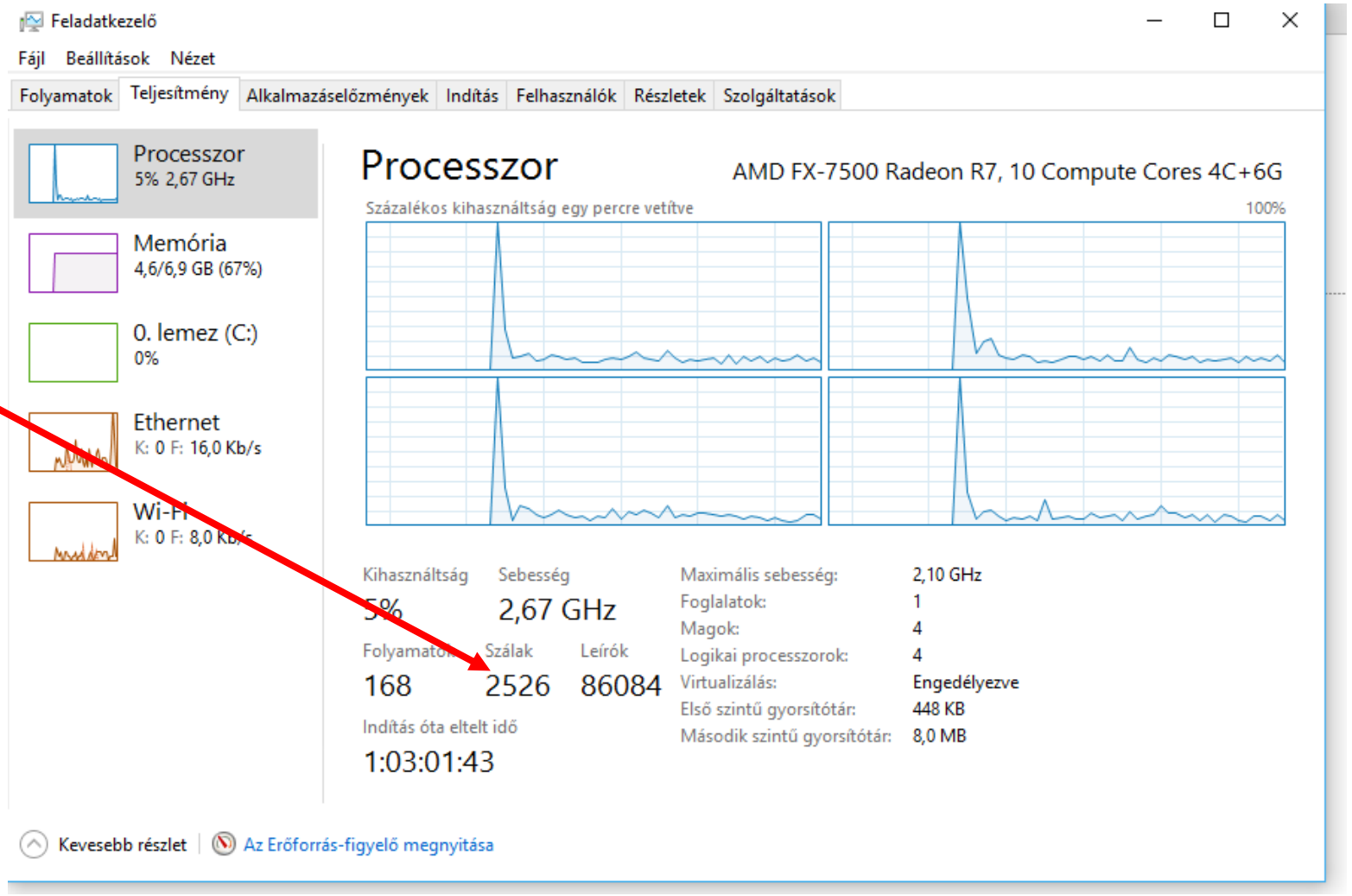
Threads

- Thread is the logical unit of execution.

- It has a separate stack
- A state of it’s own
- It is scheduled by the operating system (CPUs) or by the hardware (GPUs)
- When it is executing it is associated to some hardware resource.

Threads

- Todays computers manage hundreds or thousands of threads!
- On a handful of cores...
- How?



Threads

- Threads have states:
  - Executing
  - Waiting
  - Suspended
  - ...etc

The trick is, that only this one is using the execution unit

Threads

- Threads have states:
  - Executing
  - Waiting
  - Suspended
  - ...etc

These are put aside, they aren't executing

Threads in C++

- There is always at least one thread in the program:

- The “main” thread, the one that hosts the `main()` function
- It is created when the operating system starts our program and destroyed at exit.

Threads in C++

The thread executing the current function at hand can be managed by the following functions in the `std::this_thread` namespace:

- `std::yield`
  - allow other threads to be scheduled first
- `std::sleep_for, std::sleep_until`
  - suspend current thread for a given time or until a specific time point

Threads in C++

All other threads are represented in C++ as objects:

```
#include <thread>

std::thread t;
```

Threads in C++

A simple example to start a thread:

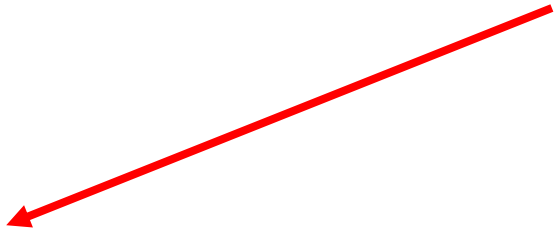
```
#include <thread>
#include <iostream>

int main()
{
	auto task = [](){ std::cout << "done.\n"; };
	std::thread t(task);
	t.join();
	return 0;
}
```

Threads in C++

A simple example to start a thread:

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

int main()
{
      auto task = [](){ std::cout << "done.\n"; };
      std::thread t(task);
      t.join();
      return 0;
}
```

This is the function that we wish to execute in a separate thread



Threads in C++

A simple example to start a thread:

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

int main()
{
	auto task = [](){ std::cout << "done.\n"; };
	std::thread t(task);
	t.join();
	return 0;
}
```

The constructor of `thread` takes the function and starts it in a new logical thread.

Threads in C++

A simple example to start a thread:

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

int main()
{
      auto task = [](){ std::cout << "done.\n"; };
      std::thread t(task);
      t.join();
      return 0;
}
```

Here the “main” thread waits for `t` to finish.

Threads in C++

A simple example to start a thread and pass some arguments to it:

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

int main()
{
	auto task = [](int x){ std::cout << x << "\n"; };
	std::thread t(task, 7);
	t.join();
	return 0;
}
```

Threads in C++

A simple example to start a thread and pass some arguments to it:

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

int main()
{
	auto task = [](int x){ std::cout << x << "\n"; };
	std::thread t(task, 7);
	t.join();
	return 0;
}
```

Threads in C++

```
std::thread t(task, 7);
```

- This only works for functions, that do not return values...
- How to get back a result?

Promises and Futures in C++

```
#include <future>
std::promise<T> promise;
std::future<T> future = promise.get_future();
```

- The promise – future pair represents a one-time asynchronous communication channel:

- The thread that has a value can set the promise once
- Another thread that has a future linked to the promise can wait for the value to become available and extract it.

Promises and Futures in C++

```
#include <thread>
#include <future>
int main()
{
    std::promise<int> promise;
    std::future<int> future = promise.get_future();
    auto f = [](std::promise<int> p_in, int x){ p_in.set_value(2*x); };

    std::thread t(f, std::move(promise), 21 );
    std::cout << "Result: " << future.get() << "\n";
    t.join();
}
```

Promises and Futures in C++

```
#include <thread>
#include <future>
int main()
{
    std::promise<int> promise;
    std::future<int> future = promise.get_future();
    auto f = [](std::promise<int> p_in, int x){ p_in.set_value(2*x); };

    std::thread t(f, std::move(promise), 21 );
    std::cout << "Result: " << future.get() << "\n";
    t.join();
}
```

Create a promise-future linked pair, that will transfer an `int`.

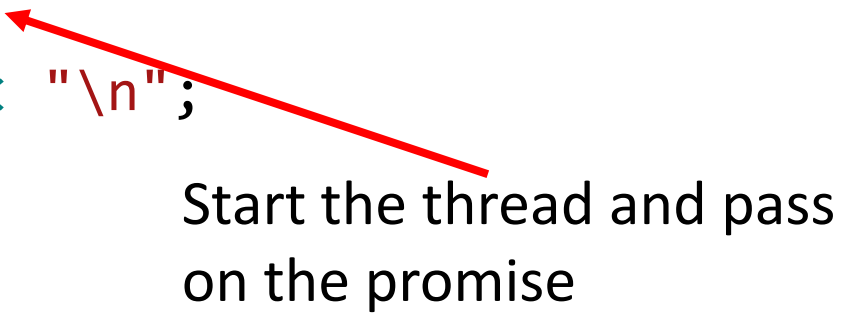
Promises and Futures in C++

```
#include <thread>
#include <future>
int main()
{
    std::promise<int> promise;
    std::future<int> future = promise.get_future();
    auto f = [](std::promise<int> p_in, int x){ p_in.set_value(2*x); };

    std::thread t(f, std::move(promise), 21 );
    std::cout << "Result: " << future.get() << "\n";
    t.join();
}
```

Create a function that will take the promise and sets it

Promises and Futures in C++

```
#include <thread>
#include <future>
int main()
{
    std::promise<int> promise;
    std::future<int> future = promise.get_future();
    auto f = [](std::promise<int> p_in, int x){ p_in.set_value(2*x); };

    std::thread t(f, std::move(promise), 21 );
    std::cout << "Result: " << future.get() << "\n";
    t.join();
}
```



Promises and Futures in C++

```
#include <thread>
#include <future>
int main()
{
    std::promise<int> promise;
    std::future<int> future = promise.get_future();
    auto f = [](std::promise<int> p_in, int x){ p_in.set_value(2*x); };

    std::thread t(f, std::move(promise), 21 );
    std::cout << "Result: " << future.get() << "\n";
    t.join();
}
```

The main thread will wait here, until the future is set, and `.get()` extracts the value (42)

std::async

Sometimes we just need a function returning a value,
and we don't want to mess with starting the thread and linking the promise...

`std::async` solves precisely this problem!

std::async

```
#include <future>

int main()
{
    auto f = [](int x){ return 2*x; };

    auto future = std::async( std::launch::async, f, 21 );

    std::cout << "Result: " << future.get() << "\n";
}
```

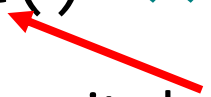
std::async

```
#include <future>

int main()
{
    auto f = [](int x){ return 2*x; };

    auto future = std::async( std::launch::async, f, 21 );

    std::cout << "Result: " << future.get() << "\n";
}
```

Starts function in a new thread,
returns a future that *will* hold the result.

std::async

```
#include <future>

int main()
{
    auto f = [](int x){ return 2*x; };

    auto future = std::async( std::launch::async, f, 21 );

    std::cout << "Result: " << future.get() << "\n";
}
```



"main" thread waits here for the result to become available.

std::async

Rule of the thumb:

- Most of the time all you need is an async. Start the task, get back result.
- If you want intermediate one-time communication and synchronization between two threads one choice is to use promise-future pairs.

Example: large vector average

Task: average a large vector with multiple threads:

```
std::vector<double> vec(10'000'000);


auto averager = [](auto it0, auto it1)
{
  auto sum = std::accumulate(it0, it1, 0.0);
  return sum / std::distance(it0, it1);
};
```

Example: large vector average

Task: average a large vector with multiple threads:

```
std::vector<double> vec(10'000'000);
```

C++11 fancy: digit separators, does not change the number just helps reading

```
auto averager = [](auto it0, auto it1)
{
  auto sum = std::accumulate(it0, it1, 0.0);
  return sum / std::distance(it0, it1);
};
```

Example: large vector average

Task: average a large vector with multiple threads:

```
std::vector<double> vec(10'000'000);
```

Helper: takes two iterators and averages the numbers between them.

```
auto averager = [](auto it0, auto it1)
{
  auto sum = std::accumulate(it0, it1, 0.0);
  return sum / std::distance(it0, it1);
};
```

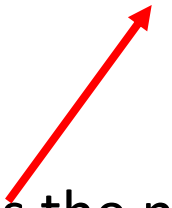
Example: large vector average

```
auto n = std::thread::hardware_concurrency();
```

```
std::vector<std::future<double>> futures(n);
```

Example: large vector average

```
auto n = std::thread::hardware_concurrency();
```



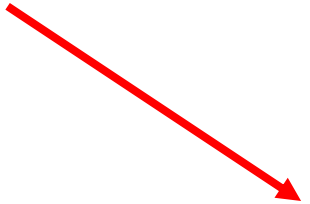
Returns the number of threads that can run in parallel
(usually number of logical cores)

```
std::vector<std::future<double>> futures(n);
```

Example: large vector average

```
auto n = std::thread::hardware_concurrency();
```

We can simply create a vector of futures



```
std::vector<std::future<double>> futures(n);
```

Example: large vector average

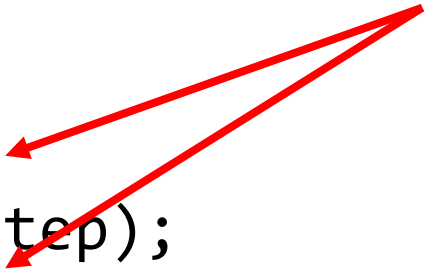
Start the threads:

```
auto step = 1 + (int)vec.size() / n;
for(int k=0; k<n; ++k )
{
    auto it0 = std::next(vec.begin(), k * step);
    auto it1 = std::next(vec.begin(),
                         std::min((k+1) * step,(int)vec.size()));
    futures[k] = std::async( std::launch::async,
                             averager, it0, it1 );
}
```

Example: large vector average

Start the threads:

Here we divide the whole range into ‘step’ sized chunks



```
auto step = 1 + (int)vec.size() / n;
for(int k=0; k<n; ++k )
{
    auto it0 = std::next(vec.begin(), k * step);
    auto it1 = std::next(vec.begin(),
                         std::min((k+1) * step,(int)vec.size()));
    futures[k] = std::async( std::launch::async,
                             averager, it0, it1 );
}
```

Example: large vector average

Start the threads:

```
auto step = 1 + (int)vec.size() / n;
for(int k=0; k<n; ++k )
{
    auto it0 = std::next(vec.begin(), k * step);
    auto it1 = std::next(vec.begin(),
                         std::min((k+1) * step,(int)vec.size()));
    futures[k] = std::async( std::launch::async,
                             averager, it0, it1 );
}
```

And start the thread, store the future…

Example: large vector average

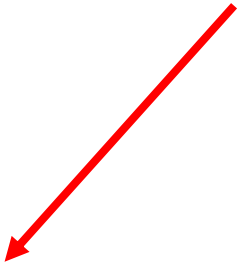
Collect the results:

```
auto partial_avg =
     std::accumulate(futures.begin(),
                     futures.end(),
                     0.0,
        [](double acc, std::future<double>& f)
        {
            return acc + f.get();
        } );
```

Example: large vector average

Collect the results:

Iterate over the futures of the threads

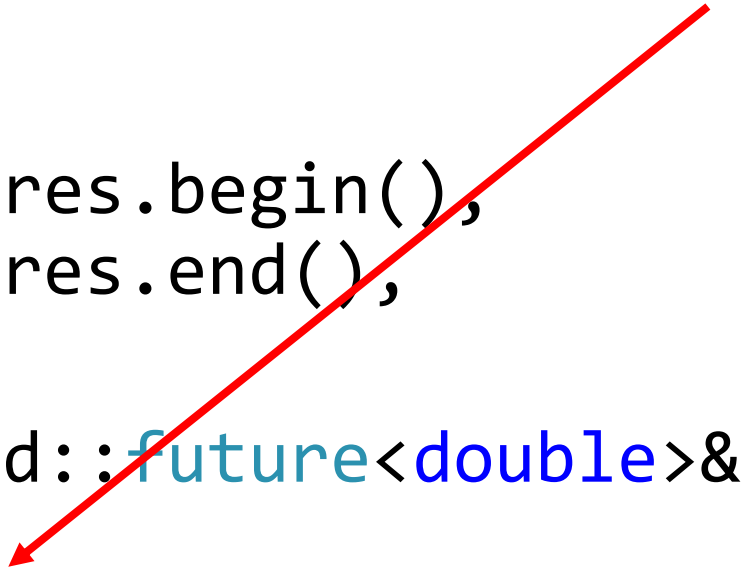


```
auto partial_avg =
     std::accumulate(futures.begin(),
                     futures.end(),
                     0.0,
        [](double acc, std::future<double>& f)
        {
            return acc + f.get();
        } );
```

Example: large vector average

Collect the results:

At each step, wait for the future and add the thread's result to the accumulator.



```
auto partial_avg =
     std::accumulate(futures.begin(),
                     futures.end(),
                     0.0,
        [](double acc, std::future<double>& f)
        {
            return acc + f.get();
        } );
```

Example: large vector average

And the final result is:

```
std::cout << "Average is: "
          << result / (double)n << "\n";
```

(where n was the number of threads☺)

Example: large vector average

Notes:

- When in C++17 parallel algorithms become available, you don't need to write such constructs anymore (at least in simple cases covered by transform and reduce).

- If you need to write it anyway, you can always write the range division once, and parametrize over the task.
  This becomes very useful if you need overlapping ranges for some reason.

- The optimal number of thread may not be simply `std::thread::hardware_concurrency()`. Experiment with less and more.

Now comes the messy part...

Sharing and mutation

There are two basic aspects of data in relation to threads:

- Sharing
- Access type

Sharing and mutation

There are two basic aspects of data in relation to threads:

- Sharing:
  - More than a single thread have access to the same data, resource, ...
- Access type:
  - A resource can be read (used) and written (modified, mutated)

Sharing and mutation

There are two basic aspects of data in relation to threads:

	Not shared	Shared
Immutable		
Mutable		

Sharing and mutation

There are two basic aspects of data in relation to threads:

	Not shared	Shared
Immutable	✓	✓
Mutable		

Sharing and mutation

There are two basic aspects of data in relation to threads:

	Not shared	Shared
Immutable	✓	✓
Mutable	✓	

Sharing and mutation

There are two basic aspects of data in relation to threads:

	Not shared	Shared
Immutable	✓	✓
Mutable	✓	!!!

Sharing and mutation

- Shared mutable state is the *root of all evil*!
- If two or more threads would like to modify the same shared resource we are dealing with a *race condition*.
- Specifically data writes occurring under a race condition can produce wrong results or corrupted data.

Sharing and mutation

- Shared mutable state is the *root of all evil*!

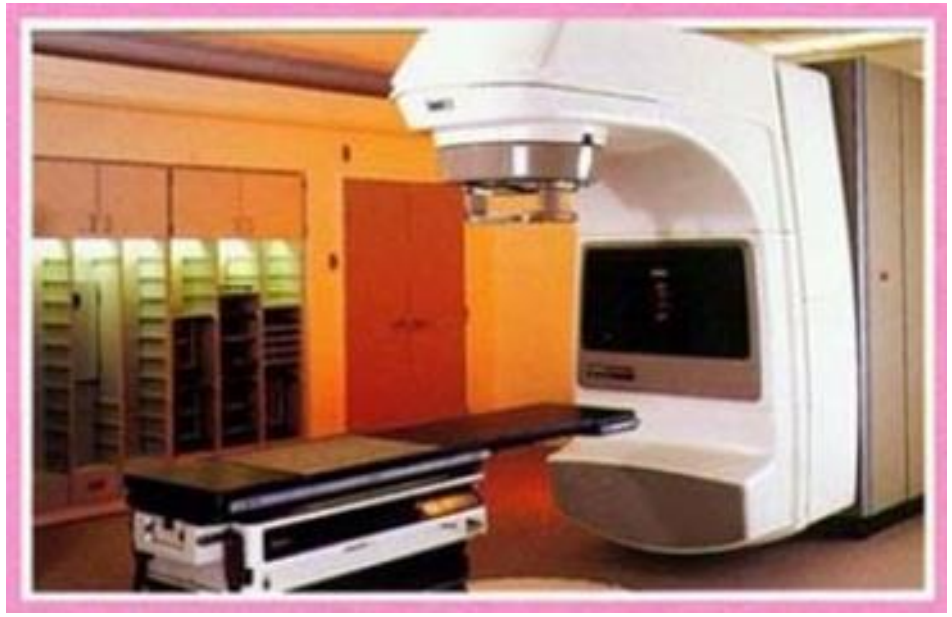
Moreover!

- Since thread execution is essentially random, race conditions are extremely hard to reproduce or debug!

Race conditions

Some illustrations of the consequences:

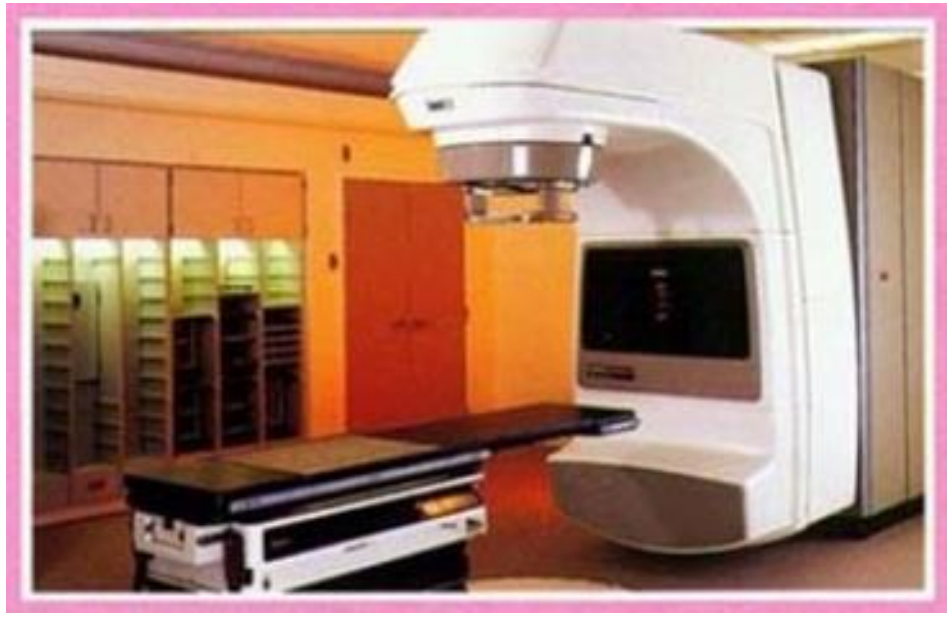
- Therac-25 was a linear accelerator to deliver x-rays and electron beams for cancer treatment.
- This 3rd generation device was the first to have full computer control



Race conditions

Some illustrations of the consequences:

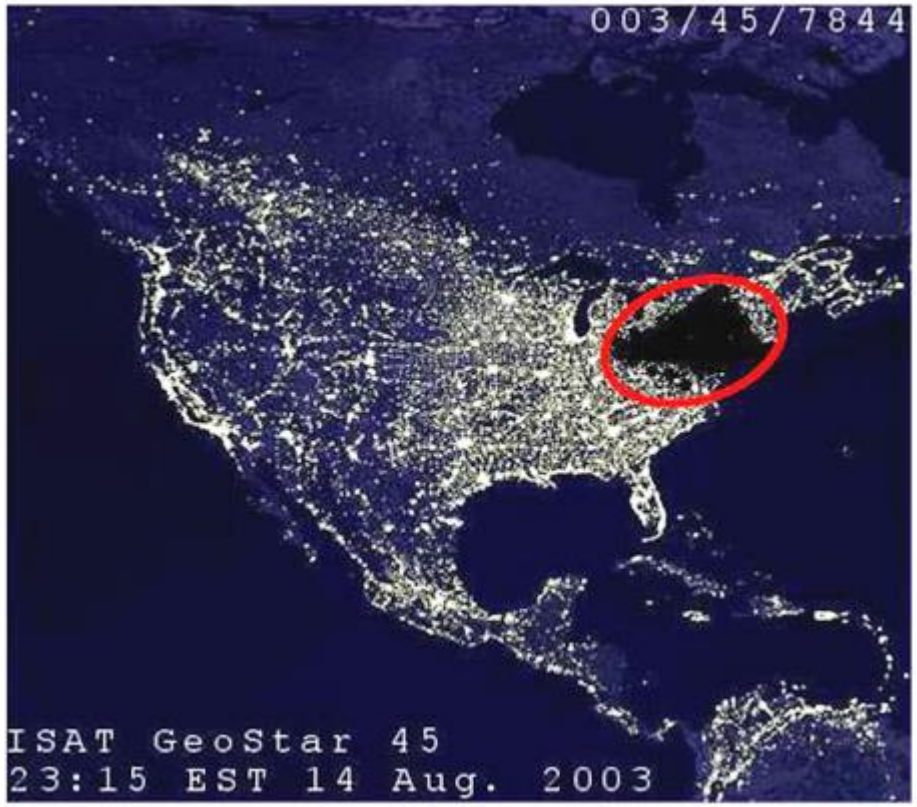
- Between 1985-87 multiple patients were overdosed because of a race condition in the design of the driver logic.



Race conditions

Some illustrations of the consequences:

- In 2003 a large blackout affecting 55 million people in Canada and North US happened.
- The crew of the control room did not became aware of the problem because of a race condition prevented alarms.



Avoiding race conditions

There are multiple ways of avoiding such cases:

- Use no shared mutable state!
- Use synchronization and concurrency control primitives...

Synchronization primitives in C++

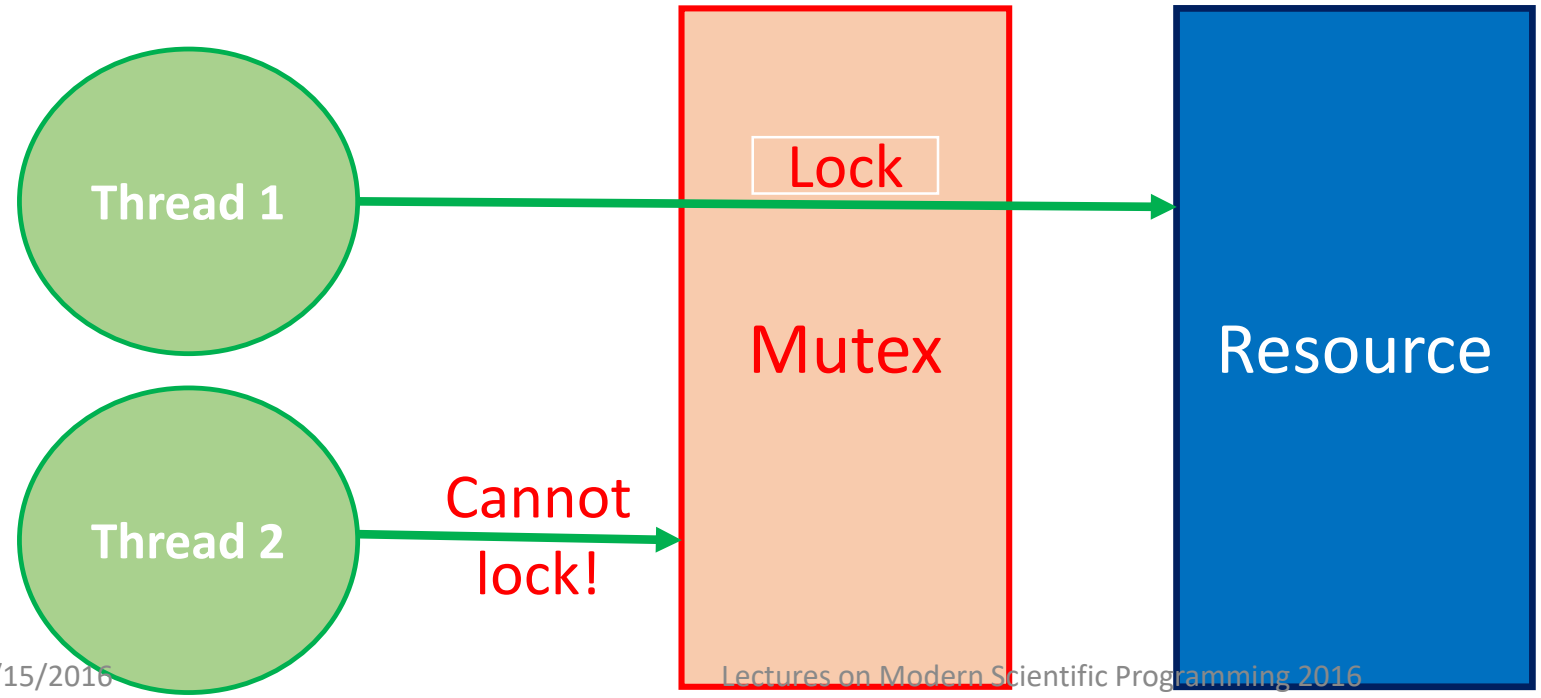
In C++11 the following primitives were standardized:

- Mutexes
- Locks
- Condition variables

Mutexes

Mutex stands for “mutual exclusion”:

- Protects shared data from being accessed by more than 1 thread at a time.



Mutexes

Types of mutexes available since C++11:

- `std::mutex` two operations: can be locked and unlocked
- `std::timed_mutex` can keep trying to lock for or until some time
- `std::recursive_mutex` can be locked multiple times, unlock need to be repeated the same number of times.
- `std::recursive_timed_mutex` combination of the above two

Mutexes and locks

Mutexes are low-level primitives.

For a safer usage, locks are recommended:

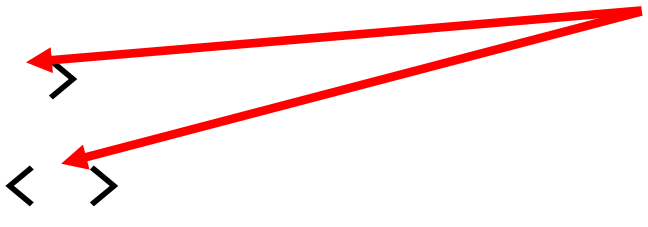
- `std::lock_guard< >`
- `std::unique_lock< >`

Mutexes and locks

Mutexes are low-level primitives.

For a safer usage, locks are recommended:

- `std::lock_guard< >`
- `std::unique_lock< >`



The template parameter is any mutex from the previous list.

Mutexes and locks

`std::lock_guard< >` is a RAII driven helper:

- It's constructor locks,
- it's destructor unlocks.

Typical usage is in a scope:

```
std::mutex m;

{     //some scope, like a function body
      std::lock_guard<std::mutex> lock(m);
      //use the shared resource

}     //at scope end the mutex is automatically unlocked
```

Mutexes and locks

Example: safely resizing a shared vector

```
std::mutex mutex;
std::vector<int> data;

{
    std::lock_guard<std::mutex> guard(mutex);
    data.resize( 100 );
}
```

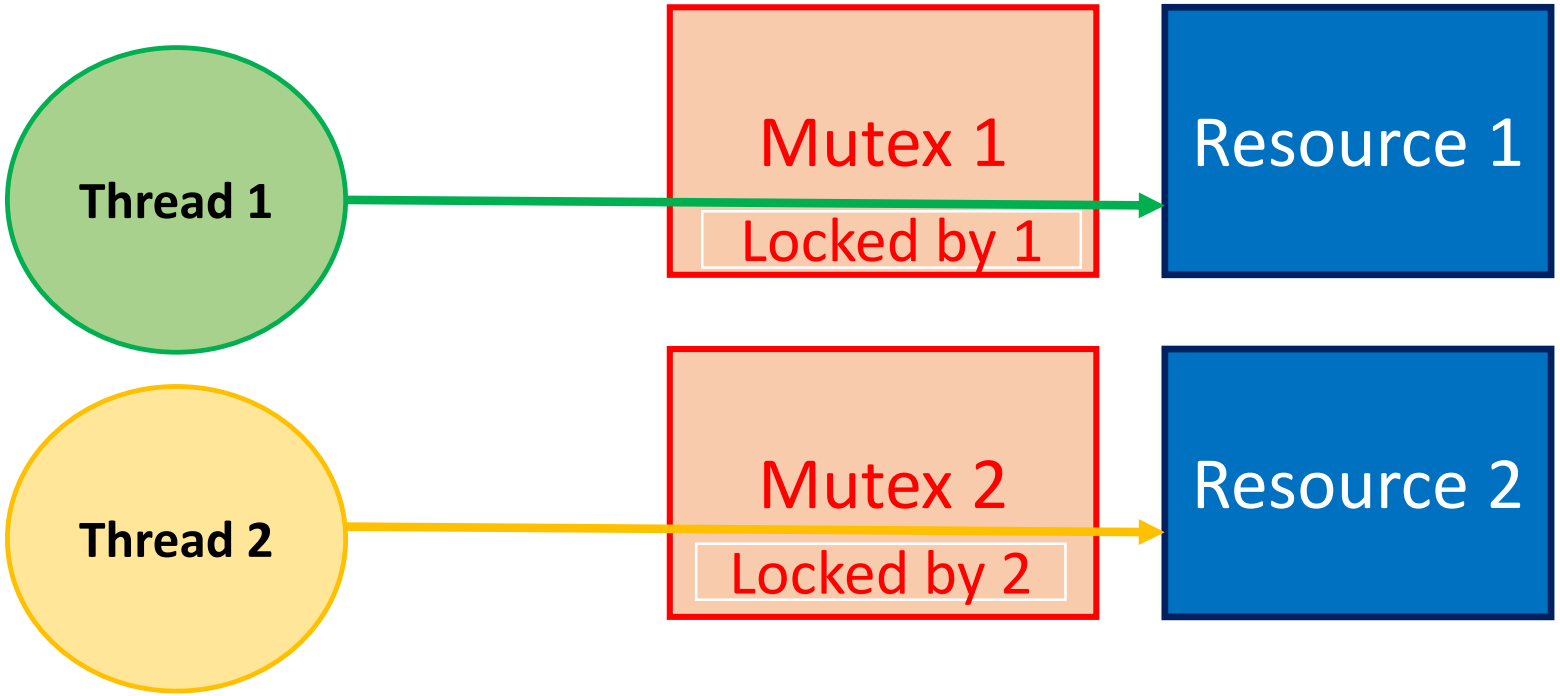
Mutexes and locks

`std::unique_lock< >` is a mutex wrapper, that is

- movable and assignable
- RAII style as lock_guard
- But has the same interface as a mutex: can be locked and unlocked

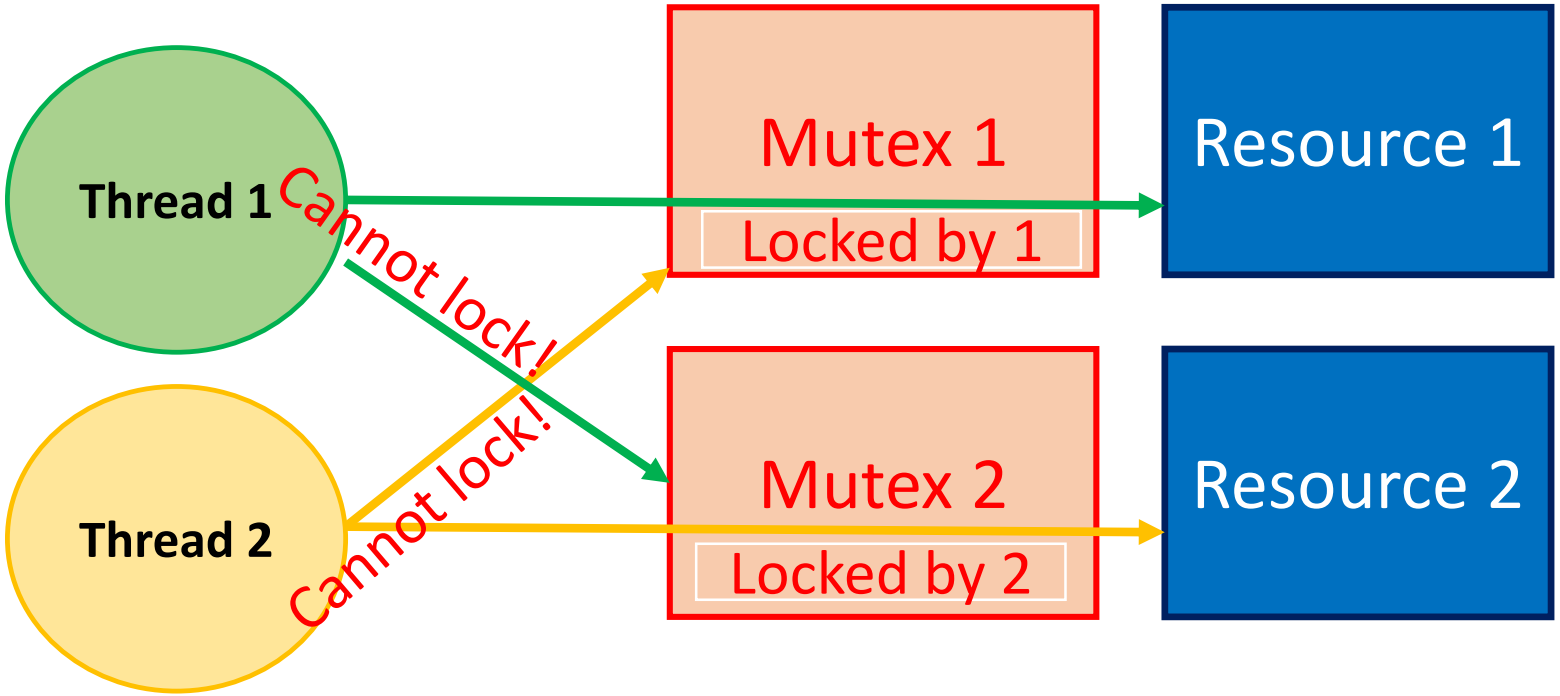
Mutexes and deadlocks

When locking multiple mutexes a deadlock may occur if threads try to lock different mutexes in a different order:



Mutexes and deadlocks

When locking multiple mutexes a deadlock may occur if threads try to lock different mutexes in a different order:



Mutexes and deadlocks

There are deadlock avoiding algorithms, but you don’t have to implement them, `std::lock` knows them!

Mutexes and deadlocks

There are deadlock avoiding algorithms, but you don't have to implement them, `std::lock` knows them!

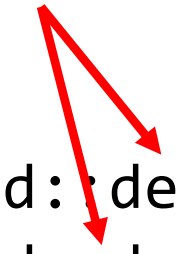
```
std::mutex m1, m2;

{
      std::unique_lock<std::mutex> lock1(m1, std::defer_lock);
      std::unique_lock<std::mutex> lock2(m2, std::defer_lock);

      std::lock(lock1, lock2);
      //use resources
}
```

Mutexes and deadlocks

There are deadlock avoiding algorithms, but you don't have to implement them, `std::lock` knows them!

```
std::mutex m1, m2;

{
      std::unique_lock<std::mutex> lock1(m1, std::defer_lock);
      std::unique_lock<std::mutex> lock2(m2, std::defer_lock);

      std::lock(lock1, lock2);
      //use resources
}
```

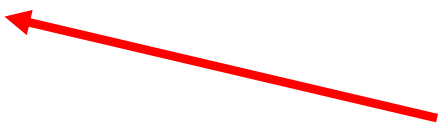
Mark constructors not to lock yet!



Mutexes and deadlocks

There are deadlock avoiding algorithms, but you don't have to implement them, `std::lock` knows them!

```
std::mutex m1, m2;

{
      std::unique_lock<std::mutex> lock1(m1, std::defer_lock);
      std::unique_lock<std::mutex> lock2(m2, std::defer_lock);

      std::lock(lock1, lock2);
      //use resources
}
```



Safely lock multiple lockables, avoiding deadlock

Mutexes and deadlocks

There are deadlock avoiding algorithms, but you don't have to implement them, `std::lock` knows them!

```
std::mutex m1, m2;

{
      std::unique_lock<std::mutex> lock1(m1, std::defer_lock);
      std::unique_lock<std::mutex> lock2(m2, std::defer_lock);

      std::lock(lock1, lock2);
      //use resources
}
```

Unlock happens automatically at the scope end

More primitives

In C++ 14 and 17 the following additional primitives were introduced:

```
std::shared_mutex
std::shared_timed_mutex
std::shared_lock<   >
```

Shared means that there are two lock states: shared and exclusive

- Multiple threads can take a shared lock (eg.: multiple read access)
- Only 1 thread can take an exclusive lock (eg.: write access)
- All shared locks must unlock before an exclusive lock can take place!

Conditional variables

`std::condition_variable` is grouping together mutual exclusion and wait-for-modification functionality:

Modifying thread need to	Waiting threads need to
1.: Acquire a mutex	1.: Lock on the mutex
2.: Modify the shared resource	2.: Wait on the lock with the cond. var.
3.: Release the mutex and notify the others	3.: When woken up, they have the mutex acquired

Conditional variables

`std::condition_variable` is grouping together mutual exclusion and wait-for-modification functionality:

```
std::condition_variable cv;
std::mutex m;
```

```
//Writing thread:
{
   std::unique_lock<std::mutex> lock(m);
   //modify resource
   cv.notify_all();
}
```

```
//Reading threads:
{
   std::unique_lock<std::mutex> lock(m);
   cv.wait(lock);
   //use resource
}
```

Conditional variables

The wakeup is guaranteed to affect only those threads that started waiting *before* the notification was signaled.

However: it is not guaranteed that a wake-up is always preceded by a notification! This is called *spurious wake-up*.
Reasons for these root deep in OS kernel programming (link).

Spurious wake-ups

To ignore spurious wake-ups, it is recommended to check whether a modification took place. This is provided by the wait overloads that take a predicate:

```
template< class Predicate >
void wait( std::unique_lock<std::mutex>& lock,
           Predicate pred )
{
      while (!pred())
      {
            wait(lock);
      }
}
```

Atomics

An atomic operation is a simple operation that cannot be interrupted by an other thread.

- Atomics are for very simple types and very simple operations
- They are provided by hardware
- They are the cheapest way to protect data from races

Atomics

Since C++11 we have `std::atomic<T>`

The template is defined for *any type*, but only simple types are using atomics directly, more complex ones use locks under the hood.

You can query if it is lock-free or not.

Atomics

The following operations are provided on `std::atomic<T>`

- Load and store
- Exchange, compare-and-exchange
- Add, subtract
- Logical AND, OR, XOR

Atomics

The following operations are provided on `std::atomic<T>`

- Load and store
- Exchange, compare-and-exchange
- Add, subtract
- Logical AND, OR, XOR

} These are available as operators too!

Atomics

Unfortunately, an `std::atomic<T>` is not copy able and not assignable!

This means, you cannot create an `std::vector<std::atomic<T>>`!

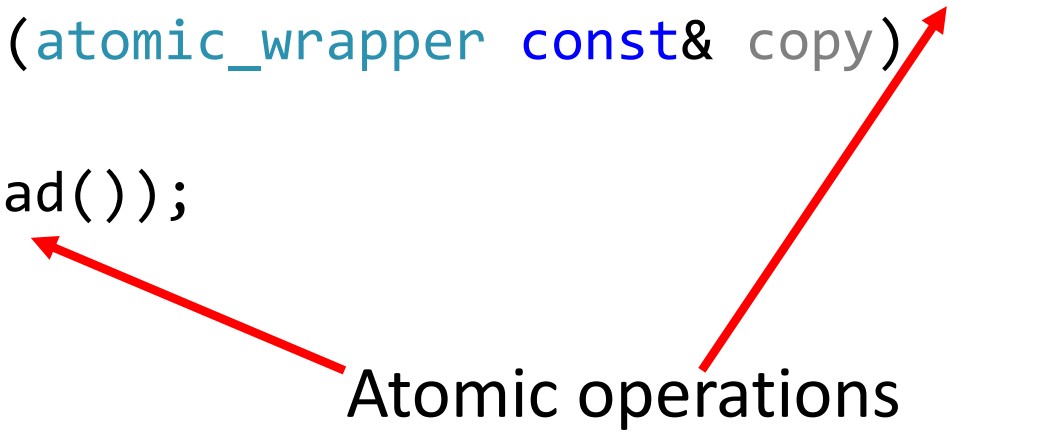
You can however, work around...

Atomics

```
template <typename T>
struct atomic_wrapper
{
  std::atomic<T> data;
  atomic_wrapper():data(){}
  atomic_wrapper(atomic_wrapper const& copy) :
                                    data(copy.data.load()){}
  atomic_wrapper& operator=(atomic_wrapper const& copy)
  {
    data.store(copy.data.load());
    return *this;
  }
};
```

Atomics

```
template <typename T>
struct atomic_wrapper
{
  std::atomic<T> data;
  atomic_wrapper():data(){}
  atomic_wrapper(atomic_wrapper const& copy) :
                                    data(copy.data.load()){}
  atomic_wrapper& operator=(atomic_wrapper const& copy)
  {
    data.store(copy.data.load());
    return *this;
  }
};
```



Atomics

Now you can create a vector, since `atomic_wrapper` is copy able.

```
std::vector<std::atomic_wrapper<T>> v;
```

Atomics example

Typical use case: multi threaded histogramming

Case:

- 10M data points
- 4 threads
- 30 bins

Result:

- Without atomics ~3M hits are lost due to race conditions, total time is 470 ms
- With atomics, the results are correct, total time is 1070 ms

Outlook to C++17

- What to expect from the new standard

Outlook to C++17

- Parallelized forms of std algorithms:

```
std::transform( policy, it1, it2, it3, f );
```

Policies:

- seq – sequential (normal, as now) execution
- par – parallel execution in multiple threads
- par_vec – vectorized parallel execution
  (loads and evaluations might be interleaved)

Outlook to C++17

- Continuation form in `std::future`:

You can chain functions with this:

```
int x = 2;
auto future = std::async( std::launch::async,
                          [](auto y){ return y+1; }, x );
auto future2 = future.then( [](auto y){ return y*10; } );
```

Outlook to C++17

- Merging of `std::futures`

You can merge and wait for multiple futures:

```
auto final_future1 = std::when_all(future1, future2, ...);

auto final_future2 = std::when_any(future1, future2, ...);
```

Outlook to C++17

- New synchronization primitives:

- `std::latch`
  - implements a count-down barrier: a given number of threads can run into and get blocked. All threads released when the number is reached. Single use!
- `std::barrier`
  - Similar to latch, but reusable
- `Std::flex_barrier`
  - Similar to barrier, but the user can specify an arbitrary function to execute when the specified number of threads hit the barrier.

Outlook to C++17

- `std::atomic_shared_ptr<T>`:

- Merges the merits of `std::shared_pointer` and `std::atomic`.

Summary

Parallel programming is not overly complicated if you have the right primitives

- Most of the time std::async is good for you
- Finer control can be done with std::thread
- Inter thread communication possibilities
- Beware of shared mutable states!
- C++17 parallel algorithms will take even more burden off from the developers