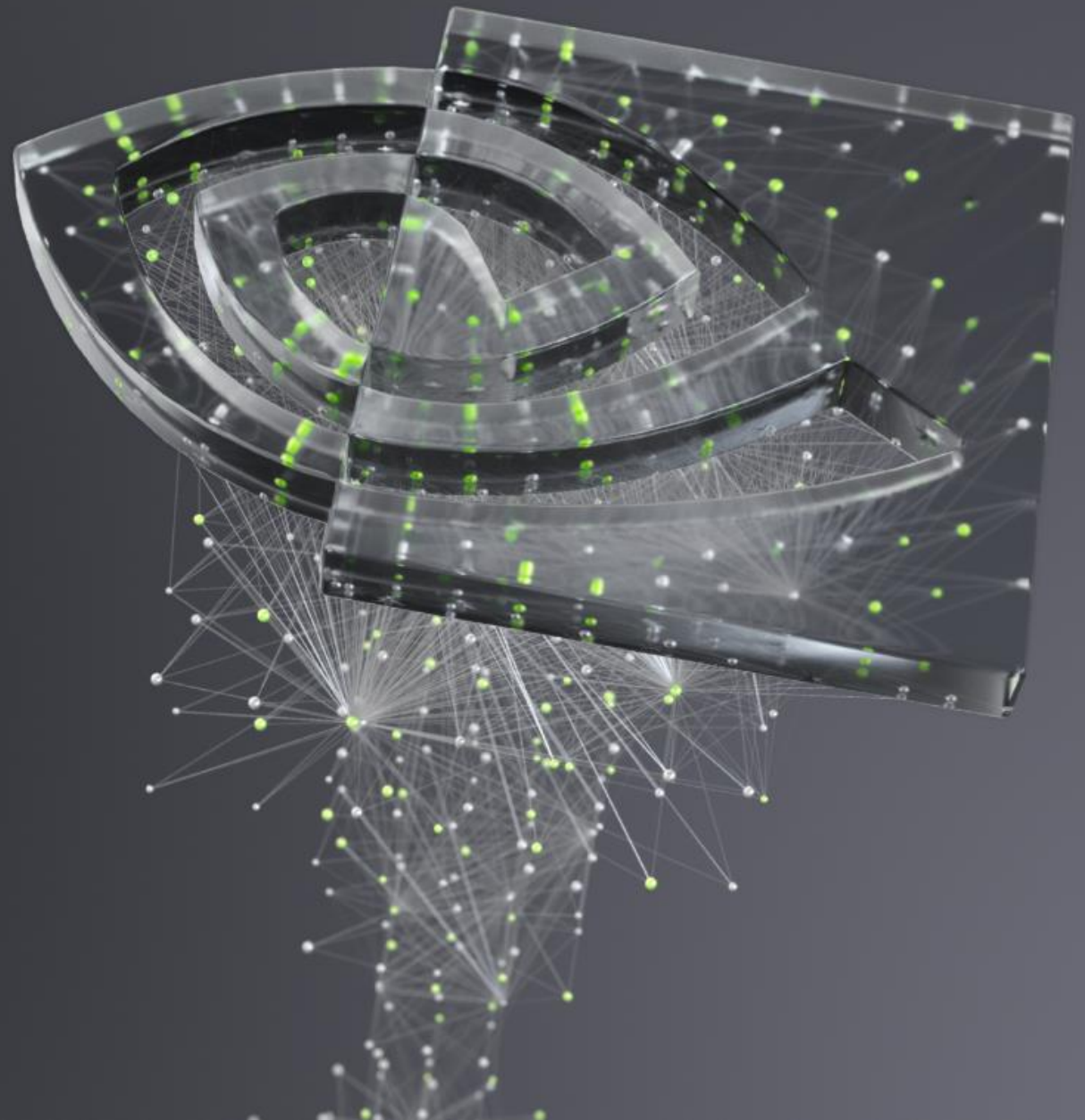
Back To Basics
Concurrency

DAVID OLSEN

Alxndrul, CC BY-SA 4.0, via Wikimedia Commons



NVIDIA

Back to Basics: Concurrency

David Olsen, Software Engineer, NVIDIA

CppCon, October 5, 2023

STD::THREAD

Hello, world!

```
int main() {
  std::thread my_thread{[](int z){
      std::cout << "Hello from thread: " << z << "\n";
  }, 42};
  my_thread.join();
}
```

https://godbolt.org/z/vqnfvh9Wx

STD::COUNTING_SEMAPHORE

Lightweight synchronization primitive that can control access to a shared resource

Maintains an internal counter

- Calls to `release()` increment the counter
- Calls to `acquire()` decrement the counter, or block if counter == 0

Calls to `acquire()` and `release()` can happen on different threads

https://en.cppreference.com/w/cpp/thread/counting_semaphore

CONCURRENCY

Definition

"Multiple logical threads of execution with unknown inter-task dependencies"

– Daisy Hollman, "A Unifying Abstraction for Async," CppCon 2019

CONCURRENCY

Definition

"Multiple logical threads of execution with unknown inter-task dependencies"

– Daisy Hollman, "A Unifying Abstraction for Async," CppCon 2019

CONCURRENCY

Definition

"Multiple logical threads of execution with [some] inter-task dependencies"

CONCURRENCY

Definition

"Multiple logical threads of execution with [some] inter-task dependencies"

Doing things at the same time

CONCURRENCY

Definition

"Multiple logical threads of execution with [some] inter-task dependencies"

Doing things at the same time

Some things need to happen before other things

CONCURRENCY

Definition

"Multiple logical threads of execution with [some] inter-task dependencies"

Doing things at the same time

Some things need to happen before other things

Some things can't happen at the same time

CONCURRENCY

Definition

"Multiple logical threads of execution with [some] inter-task dependencies"

Doing things at the same time

Some things need to happen before other things

Some things can't happen at the same time

Concurrency

Definition

"Multiple logical threads of execution with [some] inter-task dependencies"

Doing things at the same time

Some things need to happen before other things

Some things can't happen at the same time

synchronization

Parallelism

Definition

"Multiple logical threads of execution with no inter-task dependencies"

– Daisy Hollman, "A Unifying Abstraction for Async," CppCon 2019

Concurrency Made Easy

Concurrency Made Easy

Not easy, but a little less hard

Concurrency Made Easy

Not easy, but a little less hard

And it’s really parallelism, not concurrency

Concurrency Made Easy

Well, less hard. And it's parallelism, not concurrency

C++ parallel algorithms

Let the compiler do all the hard work for you

Bryce Adelstein Lelbach, "The C++17 Parallel Algorithms Library and Beyond," CppCon 2016

David Olsen, "Faster Code Through Parallelism or CPUs and GPUs," CppCon 2019

C++ Parallel Algorithms

Execution policies

Add execution policy as first argument to algorithm call

`std::execution::seq`

Run sequentially, no parallelism

`std::execution::par`, `std::execution::par_unseq`

Request to compiler to run in parallel

Promise by user that code is safe to run in parallel; no data races

`std::execution::unseq`, `std::execution::par_unseq`

Request to compiler to vectorize

Promise by user that code is safe to vectorize; no data races or locks

C++ Parallel Algorithms

Existing algorithms

If using standard algorithms already, add execution policy argument

```
std::sort(items.begin(), items.end(), compare_by_price{});
```

```
std::fill(v.begin(), v.end(), -1);
```

C++ Parallel Algorithms

Existing algorithms

If using standard algorithms already, add execution policy argument

```
std::sort(std::execution::par,
          items.begin(), items.end(), compare_by_price{});
```

```
std::fill(std::execution::par_unseq,
          v.begin(), v.end(), -1);
```

C++ Parallel Algorithms

Loops to algorithms

Some raw loops can be converted into parallel algorithms

```
std::int64_t sum = 0;
for (std::int64_t x : v) {
  sum += x;
}
```

https://godbolt.org/z/eTE9qPjq5

C++ Parallel Algorithms

Loops to algorithms

Some raw loops can be converted into parallel algorithms

```
std::int64_t sum = 0;
for (std::int64_t x : v) {
  sum += x;
}

std::int64_t sum = 0;
std::for_each(std::execution::par_unseq, v.begin(), v.end(),
    [&sum](std::int64_t x){ sum += x; });
```

https://godbolt.org/z/3PWxd79rT

C++ Parallel Algorithms

Loops to algorithms

Some raw loops can be converted into parallel algorithms

```
std::int64_t sum = 0;
for (std::int64_t x : v) {
  sum += x;
}

std::int64_t sum = 0;
std::for_each(std::execution::par_unseq, v.begin(), v.end(),
    [&sum](std::int64_t x){ sum += x; });
```

Data race!
Don't do this!

https://godbolt.org/z/3PWxd79rT

C++ Parallel Algorithms

Loops to algorithms

Some raw loops can be converted into parallel algorithms

```
std::int64_t sum = 0;
for (std::int64_t x : v) {
  sum += x;
}

std::int64_t sum = 0;
std::mutex mtx;
std::for_each(std::execution::par, v.begin(), v.end(),
    [&sum, &mtx](std::int64_t x){
        std::scoped_lock lock(mtx);
        sum += x;
    });
```

https://godbolt.org/z/on4Koejqx

C++ Parallel Algorithms

Loops to algorithms

Some raw loops can be converted into parallel algorithms

```
std::int64_t sum = 0;
for (std::int64_t x : v) {
  sum += x;
}
```

```
std::int64_t sum = 0;
std::mutex mtx;
std::for_each(std::execution::par, v.begin(), v.end(),
    [&sum, &mtx](std::int64_t x){
        std::scoped_lock lock(mtx);
        sum += x;
    });
```

Abysmal performance!
Don't do this!

https://godbolt.org/z/on4Koejqx

C++ Parallel Algorithms

Loops to algorithms

Some raw loops can be converted into parallel algorithms

```
std::int64_t sum = 0;
for (std::int64_t x : v) {
  sum += x;
}

std::int64_t sum = std::reduce(std::execution::par_unseq,
    v.begin(), v.end(), std::int64_t(0), std::plus<>{});
```

https://godbolt.org/z/KaEejaaKz

C++ Parallel Algorithms

Some of the algorithms

```
for_each  for_each_n  transform

find  find_if  find_end  search  count  any_of  adjacent_find

copy  copy_if  move  fill  replace  generate  rotate

sort  stable_sort  partial_sort  nth_element  is_sorted

reduce  transform_reduce  inclusive_scan  exclusive_scan
```

C++ Parallel Algorithms

Some of the algorithms

for_each for_each_n transform

find find_if find_end search count any_of adjacent_find

copy copy_if move fill replace generate rotate

sort stable_sort partial_sort nth_element is_sorted

reduce transform_reduce inclusive_scan exclusive_scan

C++ Parallel Algorithms

Some of the algorithms

```
for_each  for_each_n  transform
find  find_if  find_end  search  count  any_of  adjacent_find
copy  copy_if  move  fill  replace  generate  rotate
sort  stable_sort  partial_sort  nth_element  is_sorted
reduce  transform_reduce  inclusive_scan  exclusive_scan
```

C++ Parallel Algorithms

Some of the algorithms

```
for_each  for_each_n  transform
find  find_if  find_end  search  count  any_of  adjacent_find
copy  copy_if  move  fill  replace  generate  rotate
sort  stable_sort  partial_sort  nth_element  is_sorted
reduce  transform_reduce  inclusive_scan  exclusive_scan
```

C++ Parallel Algorithms

Some of the algorithms

```
for_each  for_each_n  transform

find  find_if  find_end  search  count  any_of  adjacent_find

copy  copy_if  move  fill  replace  generate  rotate

sort  stable_sort  partial_sort  nth_element  is_sorted

reduce  transform_reduce  inclusive_scan  exclusive_scan
```

C++ Parallel Algorithms

Some of the algorithms

```
for_each  for_each_n  transform
find  find_if  find_end  search  count  any_of  adjacent_find
copy  copy_if  move  fill  replace  generate  rotate
sort  stable_sort  partial_sort  nth_element  is_sorted
reduce  transform_reduce  inclusive_scan  exclusive_scan
```

C++ Parallel Algorithms

Some of the algorithms

```
for_each  for_each_n  transform
find  find_if  find_end  search  count  any_of  adjacent_find
copy  copy_if  move  fill  replace  generate  rotate
sort  stable_sort  partial_sort  nth_element  is_sorted
reduce  transform_reduce  inclusive_scan  exclusive_scan
```

STD::TRANSFORM_REDUCE

Word count

```
std::size_t word_count(std::string_view s) {
  if (s.empty()) return 0;
  return std::transform_reduce(std::execution::par_unseq,
    begin(s), end(s) - 1, begin(s) + 1,
    std::size_t(!std::isspace(s.front()) ? 1 : 0),
    std::plus<>{},
    [] (char l, char r) {
        return std::isspace(l) && !std::isspace(r);
    });
}
```

Bryce Adelstein Lelbach, "C++17 Parallel Algorithms and Beyond," CppCon 2016

STD::TRANSFORM_REDUCE

Traveling salesman

```
route_cost find_best_route(int const* distances, int N) {
  return std::transform_reduce(std::execution::par,
      counting_iterator<long>(0L), counting_iterator<long>(factorial(N)),
      route_cost(),
      route_cost::min,
      [=](long i) {
        int cost = 0;
        route_iterator it(i, N);
        int from = it.first();
        while (!it.done()) {
          int to = it.next();
          cost += distances[from*N + to];
          from = to;
        }
        return route_cost(i, cost);
      });
}
```

David Olsen, "Faster Code Through Parallelism or CPUs and GPUs," CppCon 2019

C++ Parallel Algorithms

Summary

Use C++ standard parallel algorithms when appropriate

Use the right algorithm for the task

Make sure your code is thread safe with no data races

STD::THREAD

https://en.cppreference.com/w/cpp/thread/thread

STD::THREAD

Hello, world!

```
int main() {
  std::thread my_thread{[](int z){
      std::cout << "Hello from thread: " << z << "\n";
  }, 42};
  my_thread.join();
}
```

https://godbolt.org/z/vqnfvh9Wx

STD::THREAD

Hello, world!

```
int main() {
  std::thread my_thread{[](int z){
      std::cout << "Hello from thread: " << z << "\n";
  }, 42};
  my_thread.join();
}
```

`std::thread` variable is a handle, not the thread

Exists in the main thread, not in the created thread

STD::THREAD

Hello, world!

```
int main() {
  std::thread my_thread{[](int z){
      std::cout << "Hello from thread: " << z << "\n";
  }, 42};
  my_thread.join();
}
```

Pass a callable to `std::thread` constructor

Can be lambda, pointer to function, `std::function`, object with call operator

STD::THREAD

Hello, world!

```
int main() {
  std::thread my_thread{[](int z){
      std::cout << "Hello from thread: " << z << "\n";
  }, 42};
  my_thread.join();
}
```

Arguments for the callable are also passed to the `std::thread` constructor

STD::THREAD

Hello, world!

```
int main() {
  std::thread my_thread{[](int z){
      std::cout << "Hello from thread: " << z << "\n";
  }, 42};
  my_thread.join();
}
```

std::thread constructor creates a new thread and runs the callable on that thread

STD::THREAD

Hello, world!

```
int main() {
  std::thread my_thread{[](int z){
      std::cout << "Hello from thread: " << z << "\n";
  }, 42};
  my_thread.join();
}
```

`join()` waits for the thread to complete, blocking if necessary

STD::THREAD

Hello, world!

```
int main() {
  std::thread my_thread{[](int z){
      std::cout << "Hello from thread: " << z << "\n";
  }, 42};
  my_thread.join();
}
```

`join()` waits for the thread to complete, blocking if necessary

Must call `join()` or `detach()`, or program will `std::terminate()`

STD::THREAD

Low-level tool

Don’t use `std::thread` directly

Use a threading library or other task management system

Similar characteristics to `std::thread`:

- A way to start a thread or start work concurrently
- A way to wait for the task to complete, or at least check for completion

Might have a way to return a value

Data Races

Race Conditions

Data Races

Race Conditions

Conflict over a resource without coordination

- Bad things happen as a result

Data Races

Examples

Counting people in a room while people are going in and out

Two people trying to walk through a door at the same time

Elsa/Getty Images

Data Races

Examples

Counting people in a room while people are going in and out

Two people trying to walk through a door at the same time

Outfielders both trying to catch a fly ball

Elsa/Getty Images

Data Races

Definition

Data race:

1. Two or more threads access the same memory
2. At least one access is a write
3. The threads do not synchronize with each other

A data race is undefined behavior

http://eel.is/c++draft/intro.races

Data Races

Race Conditions

It’s all about visibility of memory changes

When threads synchronize, changes made by one thread are guaranteed to be visible in the other thread

Data Race ?

Is this a data race?

```
auto assets = list_all_assets();
auto liabilities = list_all_liabilities();
currency_t net_worth;
currency_t durable_goods;
currency_t long_term_debts;
std::thread goods_thread{[&durable_goods, &assets](){
  durable_goods = calculate_durable_goods(assets);
}};
std::thread debts_thread{[&long_term_debts, &liabilities](){
  long_term_debts = calculate_long_term_debts(liabilities);
}};
net_worth = calculate_net_worth(assets, liabilities);
goods_thread.join();
debts_thread.join();
create_report(net_worth, durable_goods, long_term_debts);
```

Data Race ?

Is this a data race?

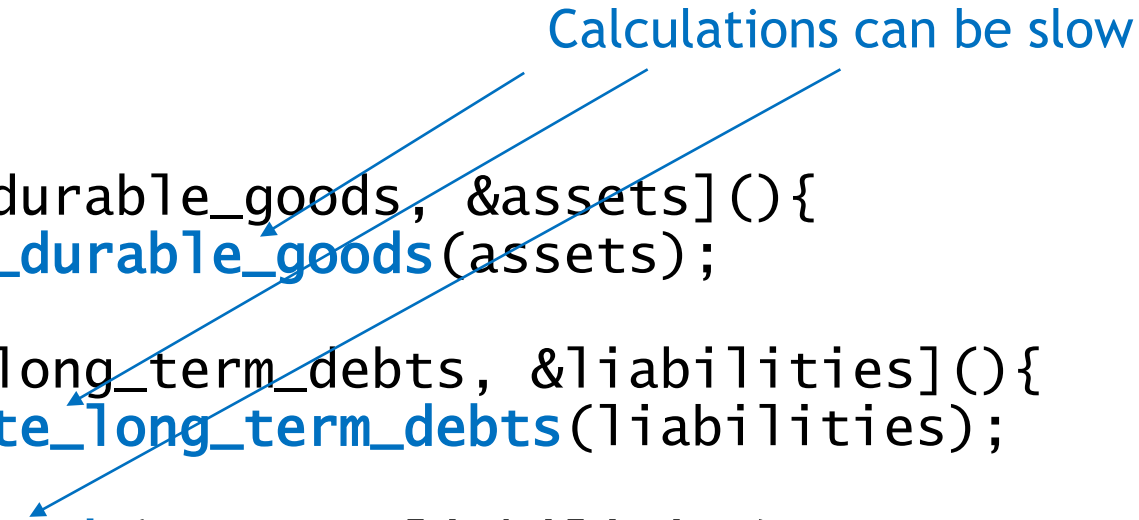
```
auto assets = list_all_assets();
auto liabilities = list_all_liabilities();
currency_t net_worth;
currency_t durable_goods;
currency_t long_term_debts;
std::thread goods_thread{[&durable_goods, &assets](){
  durable_goods = calculate_durable_goods(assets);
}};
std::thread debts_thread{[&long_term_debts, &liabilities](){
  long_term_debts = calculate_long_term_debts(liabilities);
}};
net_worth = calculate_net_worth(assets, liabilities);
goods_thread.join();
debts_thread.join();
create_report(net_worth, durable_goods, long_term_debts);
```

Goal: financial report for a company

Data Race ?

Is this a data race?

```
auto assets = list_all_assets();
auto liabilities = list_all_liabilities();
currency_t net_worth;
currency_t durable_goods;
currency_t long_term_debts;
std::thread goods_thread{[&durable_goods, &assets](){
  durable_goods = calculate_durable_goods(assets);
}};
std::thread debts_thread{[&long_term_debts, &liabilities](){
  long_term_debts = calculate_long_term_debts(liabilities);
}};
net_worth = calculate_net_worth(assets, liabilities);
goods_thread.join();
debts_thread.join();
create_report(net_worth, durable_goods, long_term_debts);
```



Data Race ?

Is this a data race?

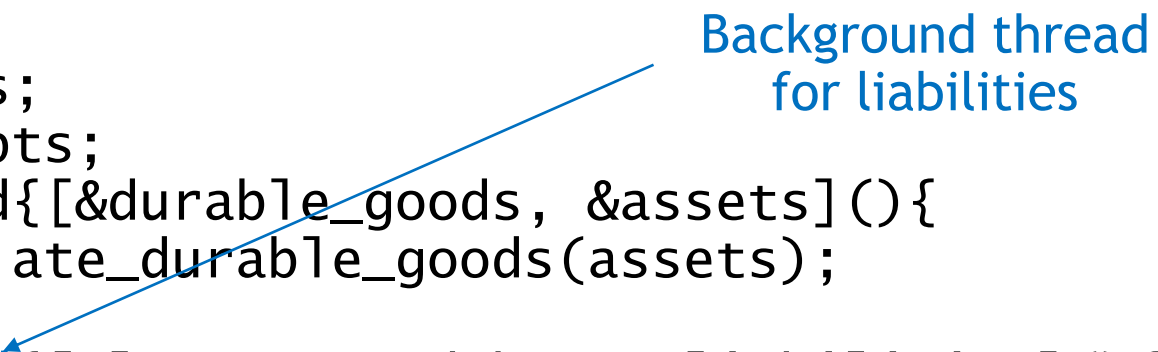
```
auto assets = list_all_assets();
auto liabilities = list_all_liabilities();
currency_t net_worth;
currency_t durable_goods;
currency_t long_term_debts;
std::thread goods_thread{[&durable_goods, &assets](){
  durable_goods = calculate_durable_goods(assets);
}};
std::thread debts_thread{[&long_term_debts, &liabilities](){
  long_term_debts = calculate_long_term_debts(liabilities);
}};
net_worth = calculate_net_worth(assets, liabilities);
goods_thread.join();
debts_thread.join();
create_report(net_worth, durable_goods, long_term_debts);
```

Background thread for durable goods

Data Race ?

Is this a data race?

```
auto assets = list_all_assets();
auto liabilities = list_all_liabilities();
currency_t net_worth;
currency_t durable_goods;
currency_t long_term_debts;
std::thread goods_thread{[&durable_goods, &assets](){
  durable_goods = calculate_durable_goods(assets);
}};
std::thread debts_thread{[&long_term_debts, &liabilities](){
  long_term_debts = calculate_long_term_debts(liabilities);
}};
net_worth = calculate_net_worth(assets, liabilities);
goods_thread.join();
debts_thread.join();
create_report(net_worth, durable_goods, long_term_debts);
```



Data Race ?

Is this a data race?

```
auto assets = list_all_assets();
auto liabilities = list_all_liabilities();
currency_t net_worth;
currency_t durable_goods;
currency_t long_term_debts;
std::thread goods_thread{[&durable_goods, &assets](){
  durable_goods = calculate_durable_goods(assets);
}};
std::thread debts_thread{[&long_term_debts, &liabilities](){
  long_term_debts = calculate_long_term_debts(liabilities);
}};
net_worth = calculate_net_worth(assets, liabilities);
goods_thread.join();
debts_thread.join();
create_report(net_worth, durable_goods, long_term_debts);
```

Net worth on main thread

Data Race ?

Is this a data race?

```
auto assets = list_all_assets();
auto liabilities = list_all_liabilities();
currency_t net_worth;
currency_t durable_goods;
currency_t long_term_debts;
std::thread goods_thread{[&durable_goods, &assets](){
  durable_goods = calculate_durable_goods(assets);
}};
std::thread debts_thread{[&long_term_debts, &liabilities](){
  long_term_debts = calculate_long_term_debts(liabilities);
}};
net_worth = calculate_net_worth(assets, liabilities);
goods_thread.join();
debts_thread.join();
create_report(net_worth, durable_goods, long_term_debts);
```

Wait for threads to complete

Data Race ?

Is this a data race?

Parent thread write

Child thread read

```
auto assets = list_all_assets();
auto liabilities = list_all_liabilities();
currency_t net_worth;
currency_t durable_goods;
currency_t long_term_debts;
std::thread goods_thread{[&durable_goods, &assets](){
  durable_goods = calculate_durable_goods(assets);
}};
std::thread debts_thread{[&long_term_debts, &liabilities](){
  long_term_debts = calculate_long_term_debts(liabilities);
}};
net_worth = calculate_net_worth(assets, liabilities);
goods_thread.join();
debts_thread.join();
create_report(net_worth, durable_goods, long_term_debts);
```

Synchronization: Thread Creation

Creating a thread synchronizes the parent and child threads

All memory changes in parent thread before thread creation are visible in the child thread

Data Race ?

Is this a data race?

Parent thread write

Synchronization

Child thread read

```
auto assets = list_all_assets();
auto liabilities = list_all_liabilities();
currency_t net_worth;
currency_t durable_goods;
currency_t long_term_debts;
std::thread goods_thread{[&durable_goods, &assets](){
  durable_goods = calculate_durable_goods(assets);
}};
std::thread debts_thread{[&long_term_debts, &liabilities](){
  long_term_debts = calculate_long_term_debts(liabilities);
}};
net_worth = calculate_net_worth(assets, liabilities);
goods_thread.join();
debts_thread.join();
create_report(net_worth, durable_goods, long_term_debts);
```

Data Race ?

Is this a data race?

Parent thread write

Synchronization

Child thread read

```
auto assets = list_all_assets();
auto liabilities = list_all_liabilities();
currency_t net_worth;
currency_t durable_goods;
currency_t long_term_debts;
std::thread goods_thread{[&durable_goods, &assets](){
  durable_goods = calculate_durable_goods(assets);
}};
std::thread debts_thread{[&long_term_debts, &liabilities](){
  long_term_debts = calculate_long_term_debts(liabilities);
}};
net_worth = calculate_net_worth(assets, liabilities);
goods_thread.join();
debts_thread.join();
create_report(net_worth, durable_goods, long_term_debts);
```

Data Race ?

Is this a data race?

```
auto assets = list_all_assets();
auto liabilities = list_all_liabilities();
currency_t net_worth;
currency_t durable_goods;
currency_t long_term_debts;
std::thread goods_thread{[&durable_goods, &assets](){
  durable_goods = calculate_durable_goods(assets);
}};
std::thread debts_thread{[&long_term_debts, &liabilities](){
  long_term_debts = calculate_long_term_debts(liabilities);
}};
net_worth = calculate_net_worth(assets, liabilities);
goods_thread.join();
debts_thread.join();
create_report(net_worth, durable_goods, long_term_debts);
```

Child thread write

Parent thread read

Synchronization: Thread Join

Joining a thread synchronizes the parent and child threads

All memory changes in child thread are visible in parent thread after call to join()

Data Race ?

Is this a data race?

```
auto assets = list_all_assets();
auto liabilities = list_all_liabilities();
currency_t net_worth;
currency_t durable_goods;
currency_t long_term_debts;
std::thread goods_thread{[&durable_goods, &assets](){
  durable_goods = calculate_durable_goods(assets);
}};
std::thread debts_thread{[&long_term_debts, &liabilities](){
  long_term_debts = calculate_long_term_debts(liabilities);
}};
net_worth = calculate_net_worth(assets, liabilities);
goods_thread.join();
debts_thread.join();
create_report(net_worth, durable_goods, long_term_debts);
```

Child thread write

Synchronization

Parent thread read

Data Race ?

Is this a data race?

```
auto assets = list_all_assets();
auto liabilities = list_all_liabilities();
currency_t net_worth;
currency_t durable_goods;
currency_t long_term_debts;
std::thread goods_thread{[&durable_goods, &assets](){
  durable_goods = calculate_durable_goods(assets);
}};
std::thread debts_thread{[&long_term_debts, &liabilities](){
  long_term_debts = calculate_long_term_debts(liabilities);
}};
net_worth = calculate_net_worth(assets, liabilities);
goods_thread.join();
debts_thread.join();
create_report(net_worth, durable_goods, long_term_debts);
```

Child thread

Parent thread

Data Race ?

Is this a data race?

All reads,
no synchronization needed

```
auto assets = list_all_assets();
auto liabilities = list_all_liabilities();
currency_t net_worth;
currency_t durable_goods;
currency_t long_term_debts;
std::thread goods_thread{[&durable_goods, &assets](){
  durable_goods = calculate_durable_goods(assets);
}};
std::thread debts_thread{[&long_term_debts, &liabilities](){
  long_term_debts = calculate_long_term_debts(liabilities);
}};
net_worth = calculate_net_worth(assets, liabilities);
goods_thread.join();
debts_thread.join();
create_report(net_worth, durable_goods, long_term_debts);
```

Child thread

Parent thread

Not a Data Race

```
auto assets = list_all_assets();
auto liabilities = list_all_liabilities();
currency_t net_worth;
currency_t durable_goods;
currency_t long_term_debts;
std::thread goods_thread{[&durable_goods, &assets](){
  durable_goods = calculate_durable_goods(assets);
}};
std::thread debts_thread{[&long_term_debts, &liabilities](){
  long_term_debts = calculate_long_term_debts(liabilities);
}};
net_worth = calculate_net_worth(assets, liabilities);
goods_thread.join();
debts_thread.join();
create_report(net_worth, durable_goods, long_term_debts);
```

MUTEXES

MUTual EXclusion

Most basic form of synchronization

Only one thread at a time can lock or acquire the mutex

Mutexes

Example

An airplane lavatory

- Anyone can go in and lock the door
- Has exclusive use of lavatory while the door is locked
- Unlocking the door and leaving makes it available for the next person

Mutexes

Example

An airplane lavatory

- Anyone can go in and lock the door
- Has exclusive use of lavatory while the door is locked
- Unlocking the door and leaving makes it available for the next person

Mutexes

Example

A talking stick in a group discussion

Only the person holding the stick is supposed to talk

Mutexes

Example

Air traffic control is the mutex

Pilot gets permission to use runway

STD::MUTEX

Member functions

https://en.cppreference.com/w/cpp/thread/mutex

STD::MUTEX

Member functions

```
void lock();
```

Blocks until the lock is acquired

https://en.cppreference.com/w/cpp/thread/mutex

STD::MUTEX

Member functions

`void lock();`

Blocks until the lock is acquired

`bool try_lock();`

Returns immediately

`true` if lock acquired; `false` if not

https://en.cppreference.com/w/cpp/thread/mutex

STD::MUTEX

Member functions

`void lock();`

Blocks until the lock is acquired

`bool try_lock();`

Returns immediately

`true` if lock acquired; `false` if not

`void unlock();`

Releases the lock

Undefined behavior if current thread doesn't own the lock https://en.cppreference.com/w/cpp/thread/mutex

STD::MUTEX

Synchronization

A call to `lock()` synchronizes with any previous call to `unlock()` on the same mutex

Anything that happened on thread 1 before it called `unlock()` is visible to thread 2 after its call to `lock()` returns

STD::MUTEX

Synchronization

Thread 1

```
// Section 1A
m.lock();
// Section 1B
m.unlock();
// Section 1C
```

Thread 2

```
// Section 2A
m.lock();
// Section 2B
m.unlock();
// Section 2C
```

STD::MUTEX

Synchronization

Thread 1

```
// Section 1A
m.lock();
// Section 1B
m.unlock();
// Section 1C
```

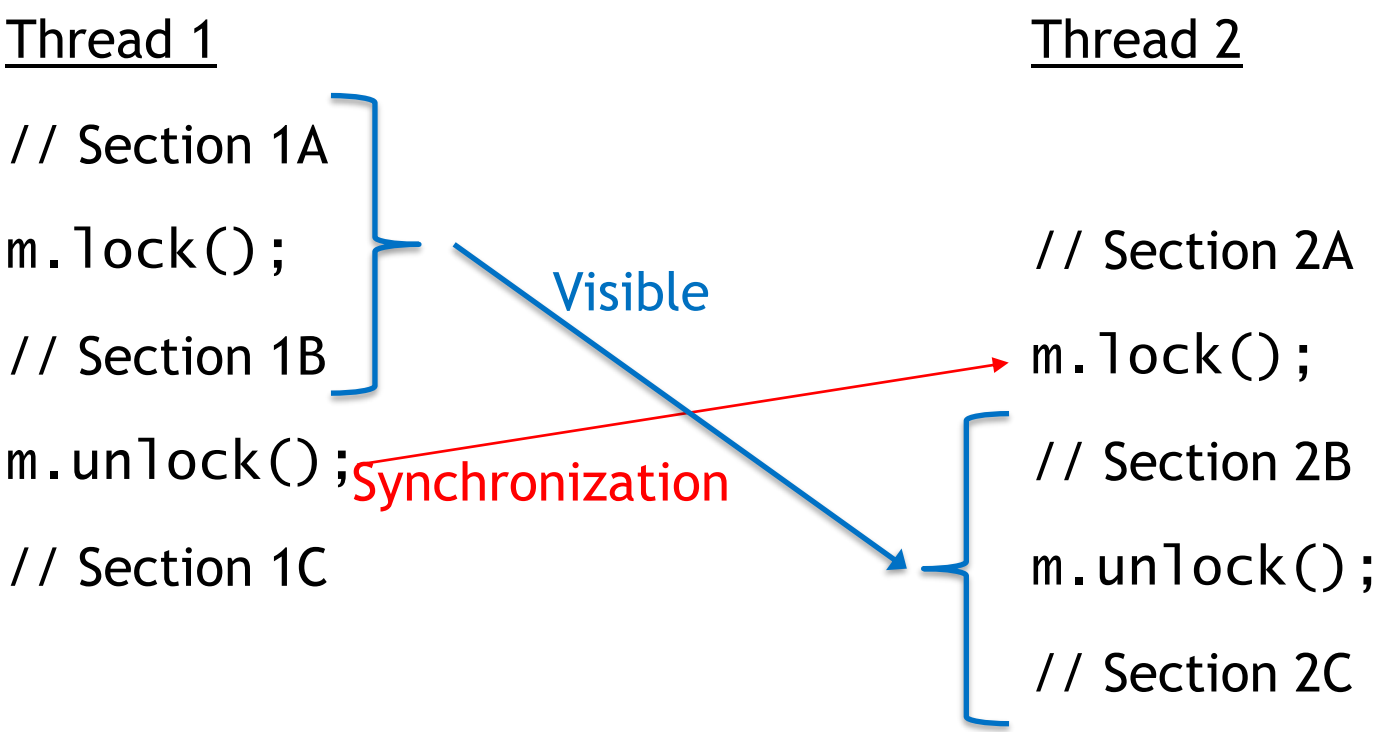
Synchronization

Thread 2

```
// Section 2A
m.lock();
// Section 2B
m.unlock();
// Section 2C
```

STD::MUTEX

Synchronization



STD::MUTEX

Member functions

Don’t call `lock()`, `try_lock()`, or `unlock()` on a mutex

STD::MUTEX

Member functions

Don’t call `lock()`, `try_lock()`, or `unlock()` on a mutex

Always use a lock guard instead

Lock Guards

RAII wrapper around mutexes

- Constructor calls lock()
- Destructor calls unlock()

Guarantee that the mutex is always released

STD::SCOPED_LOCK

Constructor takes one or more mutexes

    Calls lock() on each of the mutexes

Destructor calls unlock() on each of the mutexes

Not copyable or movable

No member functions or other operations

C++17

Uses CTAD to deduce class template arguments

```
std::scoped_lock lock(mutex_a, mutex_b);
```

https://en.cppreference.com/w/cpp/thread/scoped_lock

STD::LOCK_GUARD

Constructor takes one mutex

- Calls lock() on the mutex

Destructor calls unlock() on the mutex

Not copyable or movable

No member functions or other operations

Can’t assume CTAD is available

```
std::lock_guard<std::mutex> lock(mutex_a);
```

https://en.cppreference.com/w/cpp/thread/lock_guard

STD::UNIQUE_LOCK

Owns a mutex

Destructor calls unlock() on the mutex if the mutex is locked

Has lock(), unlock(), and several other member functions

Movable, but not copyable

Useful when you need more control over when the mutex is locked

https://en.cppreference.com/w/cpp/thread/unique_lock

Mutex and Lock Guard Example

Needs synchronization

```
if (transaction.amount < account.balance) {
  account.balance -= transaction.amount;
  record(account, transaction);
  accept_transaction(transaction);
} else {
  account.balance -= overdraft_charge;
  record(account, overdraft_of(transaction));
  notify_user(account, overdraft_message);
  reject_transaction(transaction);
}
```

https://godbolt.org/z/YhdcjncYo

Mutex and Lock Guard Example

Needs synchronization

```
if (transaction.amount < account.balance) {
  account.balance -= transaction.amount;
  record(account, transaction);
  accept_transaction(transaction);
} else {
  account.balance -= overdraft_charge;
  record(account, overdraft_of(transaction));
  notify_user(account, overdraft_message);
  reject_transaction(transaction);
}
```

https://godbolt.org/z/YhdcjncYo

Mutex and Lock Guard Example

Incomplete synchronization

```
if (transaction.amount < account.balance) {
  std::scoped_lock lock(account.mutex);
  account.balance -= transaction.amount;
  record(account, transaction);
  accept_transaction(transaction);
} else {
  std::scoped_lock lock(account.mutex);
  account.balance -= overdraft_charge;
  record(account, overdraft_of(transaction));
  notify_user(account, overdraft_message);
  reject_transaction(transaction);
}
```

https://godbolt.org/z/fnovozn8a

Mutex and Lock Guard Example

Incomplete synchronization

```
if (transaction.amount < account.balance) {
  std::scoped_lock lock(account.mutex);
  account.balance -= transaction.amount;
  record(account, transaction);
  accept_transaction(transaction);
} else {
  std::scoped_lock lock(account.mutex);
  account.balance -= overdraft_charge;
  record(account, overdraft_of(transaction));
  notify_user(account, overdraft_message);
  reject_transaction(transaction);
}
```

https://godbolt.org/z/fnovozn8a

Mutex and Lock Guard Example

Incomplete synchronization

```
if (transaction.amount < account.balance) {
  std::scoped_lock lock(account.mutex);
  account.balance -= transaction.amount;
  record(account, transaction);
  accept_transaction(transaction);
} else {
  std::scoped_lock lock(account.mutex);
  account.balance -= overdraft_charge;
  record(account, overdraft_of(transaction));
  notify_user(account, overdraft_message);
  reject_transaction(transaction);
}
```

Lock guards must be **named** variables!

https://godbolt.org/z/fnovozn8a

Mutex and Lock Guard Example

Incomplete synchronization

```
if (transaction.amount < account.balance) {
  std::scoped_lock lock(account.mutex);
  account.balance -= transaction.amount;
  record(account, transaction);
  accept_transaction(transaction);
} else {
  std::scoped_lock lock(account.mutex);
  account.balance -= overdraft_charge;
  record(account, overdraft_of(transaction));
  notify_user(account, overdraft_message);
  reject_transaction(transaction);
}
```

Mutex locked

Mutex locked

https://godbolt.org/z/fnovozn8a

Mutex and Lock Guard Example

Incomplete synchronization

No data race

```
if (transaction.amount < account.balance) {
    std::scoped_lock lock(account.mutex);
    account.balance -= transaction.amount;
    record(account, transaction);
    accept_transaction(transaction);
} else {
    std::scoped_lock lock(account.mutex);
    account.balance -= overdraft_charge;
    record(account, overdraft_of(transaction));
    notify_user(account, overdraft_message);
    reject_transaction(transaction);
}
```

Mutex locked

Mutex locked

https://godbolt.org/z/fnovozn8a

Mutex and Lock Guard Example

Incomplete synchronization

```
if (transaction.amount < account.balance) {
  std::scoped_lock lock(account.mutex);
  account.balance -= transaction.amount;
  record(account, transaction);
  accept_transaction(transaction);
} else {
  std::scoped_lock lock(account.mutex);
  account.balance -= overdraft_charge;
  record(account, overdraft_of(transaction));
  notify_user(account, overdraft_message);
  reject_transaction(transaction);
}
```

Data race!

https://godbolt.org/z/fnovozn8a

Mutex and Lock Guard Example

Incomplete synchronization

```
if (transaction.amount < account.balance) {
  std::scoped_lock lock(account.mutex);
  account.balance -= transaction.amount;
  record(account, transaction);
  accept_transaction(transaction);
} else {
  std::scoped_lock lock(account.mutex);
  account.balance -= overdraft_charge;
  record(account, overdraft_of(transaction));
  notify_user(account, overdraft_message);
  reject_transaction(transaction);
}
```

Don’t need locked mutex

https://godbolt.org/z/fnovozn8a

Mutex and Lock Guard Example

```
bool transaction_ok;
{ std::scoped_lock lock(account.mutex);
  transaction_ok = transaction.amount < account.balance;
  if (transaction_ok) {
    account.balance -= transaction.amount;
    record(account, transaction);
  } else {
    account.balance -= overdraft_charge;
    record(account, overdraft_of(transaction));
    notify_user(account, overdraft_message);
  }
}
if (transaction_ok) {
  accept_transaction(transaction);
} else {
  reject_transaction(transaction);
}
```

https://godbolt.org/z/aPjejjWxP

Mutex and Lock Guard Example

```
bool transaction_ok;
{ std::scoped_lock lock(account.mutex);
  transaction_ok = transaction.amount < account.balance;
  if (transaction_ok) {
    account.balance -= transaction.amount;
    record(account, transaction);
  } else {
    account.balance -= overdraft_charge;
    record(account, overdraft_of(transaction));
    notify_user(account, overdraft_message);
  }
}
if (transaction_ok) {
  accept_transaction(transaction);
} else {
  reject_transaction(transaction);
}
```

https://godbolt.org/z/aPjejjWxP

Mutex and Lock Guard Example

```
bool transaction_ok;
{ std::scoped_lock lock(account.mutex);
  transaction_ok = transaction.amount < account.balance;
  if (transaction_ok) {
    account.balance -= transaction.amount;
    record(account, transaction);
  } else {
    account.balance -= overdraft_charge;
    record(account, overdraft_of(transaction));
    notify_user(account, overdraft_message);
  }
}
if (transaction_ok) {
  accept_transaction(transaction);
} else {
  reject_transaction(transaction);
}
```

https://godbolt.org/z/aPjejjWxP

Mutex and Lock Guard Example

```
bool transaction_ok;
{ std::scoped_lock lock(account.mutex);
  transaction_ok = transaction.amount < account.balance;
  if (transaction_ok) {
    account.balance -= transaction.amount;
    record(account, transaction);
  } else {
    account.balance -= overdraft_charge;
    record(account, overdraft_of(transaction));
    notify_user(account, overdraft_message);
  }
}
if (transaction_ok) {
  accept_transaction(transaction);
} else {
  reject_transaction(transaction);
}
```

https://godbolt.org/z/aPjejjWxP

Mutex Gotchas

Deadlock

When multiple mutexes are locked at the same time, they must always be locked in the same order

If not, deadlock will happen

Mutex Gotchas

Deadlock

Thread 1

```
{ std::scoped_lock la1(mutex_a);
  change_data(data_a);
  { std::scoped_lock lb1(mutex_b);
    change_data(data_a, data_b);
  }
}
```

Thread 2

```
{ std::scoped_lock lb2(mutex_b);
  change_data(data_b);
  { std::scoped_lock la2(mutex_a);
    change_data(data_a, data_b);
  }
}
```

Mutex Gotchas

Deadlock

Thread 1

```
{ std::scoped_lock la1(mutex_a);
  change_data(data_a);
  { std::scoped_lock lb1(mutex_b);
    change_data(data_a, data_b);
  }
}
```

Thread 2

```
{ std::scoped_lock lb2(mutex_b);
  change_data(data_b);
  { std::scoped_lock la2(mutex_a);
    change_data(data_a, data_b);
  }
}
```

Mutex Gotchas

Deadlock

Thread 1

```
{ std::scoped_lock la1(mutex_a);
  change_data(data_a);
  { std::scoped_lock lb1(mutex_b);
    change_data(data_a, data_b);
  }
}
```

Thread 2

```
{ std::scoped_lock lb2(mutex_b);
  change_data(data_b);
  { std::scoped_lock la2(mutex_a);
    change_data(data_a, data_b);
  }
}
```

Mutex Gotchas

Deadlock

Thread 1

```
{ std::scoped_lock la1(mutex_a);
  change_data(data_a);
  { std::scoped_lock lb1(mutex_b);
    change_data(data_a, data_b);
  }
}
```

Thread 2

```
{ std::scoped_lock lb2(mutex_b);
  change_data(data_b);
  { std::scoped_lock la2(mutex_a);
    change_data(data_a, data_b);
  }
}
```

Mutex Gotchas

Deadlock

Thread 1

```
{ std::scoped_lock la1(mutex_a);
  change_data(data_a);
  { std::scoped_lock lb1(mutex_b);
    change_data(data_a, data_b);
  }
}
```

Thread 2

```
{ std::scoped_lock lb2(mutex_b);
  change_data(data_b);
  { std::scoped_lock la2(mutex_a);
    change_data(data_a, data_b);
  }
}
```

Mutex Gotchas

Deadlock

Thread 1

```
{ std::scoped_lock la1(mutex_a);
  change_data(data_a);
  { std::scoped_lock lb1(mutex_b);
    change_data(data_a, data_b);
  }
}
```

Thread 2

```
{ std::scoped_lock lb2(mutex_b);
  change_data(data_b);
  { std::scoped_lock la2(mutex_a);
    change_data(data_a, data_b);
  }
}
```

Mutex Gotchas

Deadlock

Thread 1

```
{ std::scoped_lock la1(mutex_a);
  change_data(data_a);
  { std::scoped_lock lb1(mutex_b);
    change_data(data_a, data_b);
  }
}
```

Thread 2

```
{ std::scoped_lock lb2(mutex_b);
  change_data(data_b);
  { std::scoped_lock la2(mutex_a);
    change_data(data_a, data_b);
  }
}
```

Mutex Gotchas

Deadlock

Thread 1

```
{ std::scoped_lock la1(mutex_a);
  change_data(data_a);
  { std::scoped_lock lb1(mutex_b);
    change_data(data_a, data_b);
  }
}
```

Thread 2

```
{ std::scoped_lock lb2(mutex_b);
  change_data(data_b);
  { std::scoped_lock la2(mutex_a);
    change_data(data_a, data_b);
  }
}
```

Mutex Gotchas

Deadlock

Thread 1

```
{ std::scoped_lock la1(mutex_a);
  change_data(data_a);
  { std::scoped_lock lb1(mutex_b);
    change_data(data_a, data_b);
  }
}
```

Thread 2

```
{ std::scoped_lock lb2(mutex_b);
  change_data(data_b);
  { std::scoped_lock la2(mutex_a);
    change_data(data_a, data_b);
  }
}
```

Mutex Gotchas

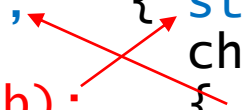
Deadlock

Thread 1

```
{ std::scoped_lock la1(mutex_a);
  change_data(data_a);
  { std::scoped_lock lb1(mutex_b);
    change_data(data_a, data_b);
  }
}
```

Thread 2

```
{ std::scoped_lock lb2(mutex_b);
  change_data(data_b);
  { std::scoped_lock la2(mutex_a);
    change_data(data_a, data_b);
  }
}
```



Mutex Gotchas

Deadlock

std::scoped_lock is useful for avoiding deadlock

If given multiple mutexes, always locks them in the same order

```
std::scoped_lock lockA{mutexA, mutexB, mutexC);
std::scoped_lock lockB(mutexC, mutexA, mutexB);
```

No deadlock

Mutex Gotchas

Deadlock

Thread 1

```
{ std::scoped_lock l1(
      mutex_a, mutex_b);
  change_data(data_a);
  change_data(data_a, data_b);
}
```

Thread 2

```
{ std::scoped_lock l2(
      mutex_b, mutex_a);
  change_data(data_b);
  change_data(data_a, data_b);
}
```

Mutex Gotchas

Deadlock

Thread 1

```
{ std::scoped_lock l1(
      mutex_a, mutex_b);
  change_data(data_a);
  change_data(data_a, data_b);
}
```

Thread 2

```
{ std::scoped_lock l2(
      mutex_b, mutex_a);
  change_data(data_b);
  change_data(data_a, data_b);
}
```

STD::ATOMIC

https://en.cppreference.com/w/cpp/atomic/atomic

STD::ATOMIC

Atomic operations happen as a unit

Things can't change in the middle

```
std::atomic<int> x;
```

Thread 1	Thread 2
`x += 5;`	`x += 7;`

STD::ATOMIC

Atomic operations happen as a unit

Things can't change in the middle

```
std::atomic<int> x;
```

Thread 1
```
x += 5;
```

Thread 2
```
x += 7;
```

No interference.
No data race.

STD::ATOMIC

Synchronization

Synchronizes between threads that access the same atomic object

```
std::atomic<int> x;
```

Thread 1

```
// Section 1A

x.store(5);

// Section 1B

// Section 1C
```

Thread 2

```
// Section 2A

// Section 2B

y = x.load();

// Section 2C
```

STD::ATOMIC

Synchronization

Synchronizes between threads that access the same atomic object

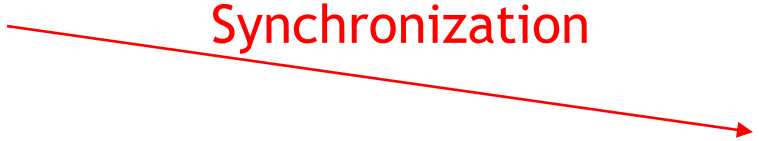
```
std::atomic<int> x;
```

<u>Thread 1</u>

```
// Section 1A
x.store(5);
// Section 1B
// Section 1C
```

<u>Thread 2</u>

```
// Section 2A
// Section 2B
y = x.load();
// Section 2C
```

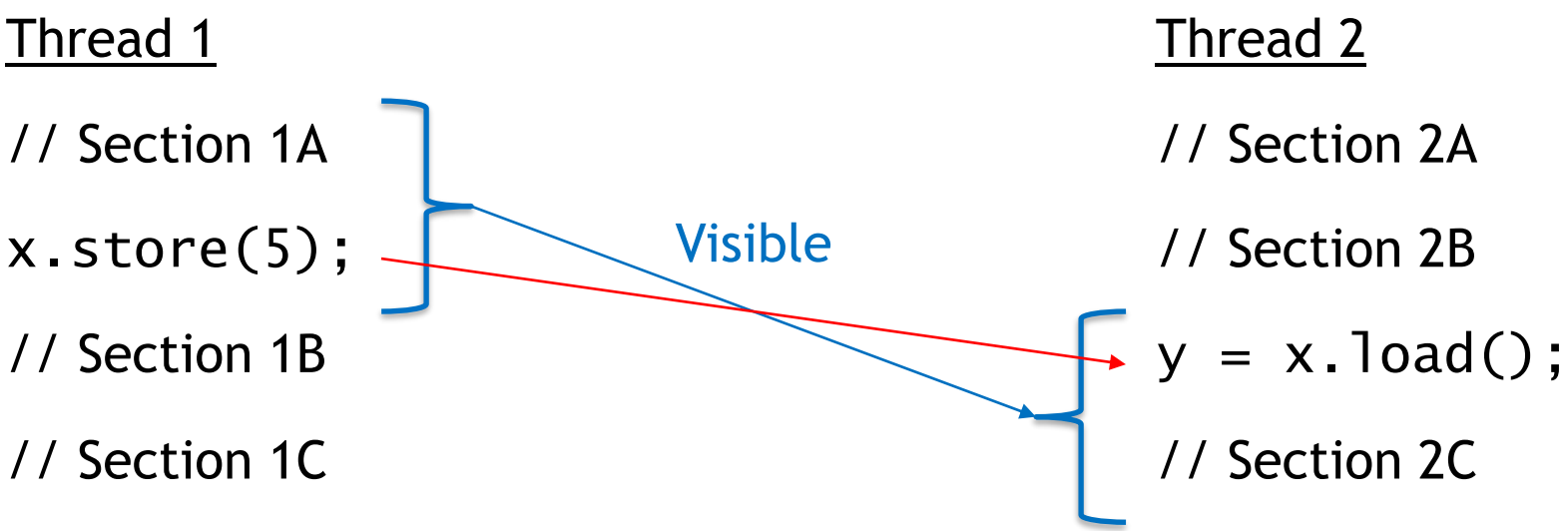


STD::ATOMIC

Synchronization

Synchronizes between threads that access the same atomic object

```
std::atomic<int> x;
```



Data Races

Examples

Data Races

Loop example – has data race

```
bool flag = false;
int main() {
  std::thread t([](){
    std::printf("Waiting...\n");
    while (not flag) { }
    std::printf("Flag changed\n");
  });
  std::this_thread::sleep_for(500ms);
  std::printf("Setting flag\n");
  flag = true;
  t.join();
  std::printf("Done\n");
}
```

https://godbolt.org/z/Gj8dsq9n6

Data Races

Loop example – has data race

```
bool flag = false;
int main() {
  std::thread t([](){
    std::printf("Waiting...\n");
    while (not flag) { }
    std::printf("Flag changed\n");
  });
  std::this_thread::sleep_for(500ms);
  std::printf("Setting flag\n");
  flag = true;
  t.join();
  std::printf("Done\n");
}
```

Read

No synchronization

Write

https://godbolt.org/z/Gj8dsq9n6

Data Races

Loop example – has data race

```
bool flag = false;
int main() {
  std::thread t([](){
    std::printf("Waiting...\n");
    while (not flag) { }
    std::printf("Flag changed\n");
  });
  std::this_thread::sleep_for(500ms);
  std::printf("Setting flag\n");
  flag = true;
  t.join();
  std::printf("Done\n");
}
```

Compiler assumes infinite loop

https://godbolt.org/z/Gj8dsq9n6

Data Races

Loop example – deadlock

```
bool flag = false;
std::mutex flag_mutex;
int main() {
  std::thread t([](){
    std::printf("Waiting...\n");
    { std::scoped_lock lock(flag_mutex);
      while (not flag) { }
    }
    std::printf("Flag changed\n");
  });
  std::this_thread::sleep_for(500ms);
  std::printf("Setting flag\n");
  { std::scoped_lock lock(flag_mutex);
    flag = true;
  }
  t.join();
  std::printf("Done\n");
}
```

https://godbolt.org/z/hb5TTbeaK

Data Races

Loop example – deadlock

```
bool flag = false;
std::mutex flag_mutex;
int main() {
  std::thread t([](){
    std::printf("Waiting...\n");
    { std::scoped_lock lock(flag_mutex);
      while (not flag) { }
    }
    std::printf("Flag changed\n");
  });
  std::this_thread::sleep_for(500ms);
  std::printf("Setting flag\n");
  { std::scoped_lock lock(flag_mutex);
    flag = true;
  }
  t.join();
  std::printf("Done\n");
}
```

Holds lock indefinitely

Can’t acquire lock

https://godbolt.org/z/hb5TTbeaK

Data Races

Loop example - mutex

```
bool flag = false;
std::mutex flag_mutex;
int main() {
  std::thread t([](){
    std::printf("Waiting...\n");
    bool local_flag;
    do {
      std::scoped_lock lock{flag_mutex};
      local_flag = flag;
    } while (not local_flag);
    std::printf("Flag changed\n");
  });
  std::this_thread::sleep_for(500ms);
  std::printf("Setting flag\n");
  { std::scoped_lock lock(flag_mutex);
    flag = true;
  }
  t.join();  std::printf("Done\n");
}
```

https://godbolt.org/z/b5ProbeMa

Data Races

Loop example - mutex

```
bool flag = false;
std::mutex flag_mutex;
int main() {
  std::thread t([](){
    std::printf("Waiting...\n");
    bool local_flag;
    do {
      std::scoped_lock lock{flag_mutex};
      local_flag = flag;
    } while (not local_flag);
    std::printf("Flag changed\n");
  });
  std::this_thread::sleep_for(500ms);
  std::printf("Setting flag\n");
  { std::scoped_lock lock(flag_mutex);
    flag = true;
  }
  t.join();  std::printf("Done\n");
}
```

https://godbolt.org/z/b5ProbeMa

Data Races

Loop example - mutex

```
bool flag = false;
std::mutex flag_mutex;
int main() {
  std::thread t([](){
    std::printf("Waiting...\n");
    bool local_flag;
    do {
      std::scoped_lock lock{flag_mutex};
      local_flag = flag;
    } while (not local_flag);
    std::printf("Flag changed\n");
  });
  std::this_thread::sleep_for(500ms);
  std::printf("Setting flag\n");
  { std::scoped_lock lock(flag_mutex);
    flag = true;
  }
  t.join();  std::printf("Done\n");
}
```

https://godbolt.org/z/b5ProbeMa

Data Races

Loop example - atomic

```
std::atomic<bool> flag{false};
int main() {
  std::thread t([](){
    std::printf("Waiting...\n");
    while (not flag) { }
    std::printf("Flag changed\n");
  });
  std::this_thread::sleep_for(500ms);
  std::printf("Setting flag\n");
  flag = true;
  t.join();
  std::printf("Done\n");
}
```

https://godbolt.org/z/f4Tv967KP

Data Races

Increment example – has data race

```
int counter = 0;
int main() {
  std::vector<std::thread> threads;
  threads.reserve(100);
  auto increment = [](){
    for (int i = 0; i < 50; ++i) {
      ++counter;
    }
  };
  for (int i = 0; i < 100; ++i) {
    threads.push_back(std::thread(increment));
  }
  for (std::thread& t : threads) { t.join(); }
  std::printf("%d\n", counter);
}
```

https://godbolt.org/z/Pqxd9adG5

Data Races

Increment example – has data race

```
int counter = 0;
int main() {
  std::vector<std::thread> threads;
  threads.reserve(100);
  auto increment = [](){
    for (int i = 0; i < 50; ++i) {
      ++counter;
    }
  };
  for (int i = 0; i < 100; ++i) {
    threads.push_back(std::thread(increment));
  }
  for (std::thread& t : threads) { t.join(); }
  std::printf("%d\n", counter);
}
```

https://godbolt.org/z/Pqxd9adG5

Data Races

Increment example – has data race

```
int counter = 0;
int main() {
  std::vector<std::thread> threads;
  threads.reserve(100);
  auto increment = [](){
    for (int i = 0; i < 50; ++i) {
      ++counter;
    }
  };
  for (int i = 0; i < 100; ++i) {
    threads.push_back(std::thread(increment));
  }
  for (std::thread& t : threads) { t.join(); }
  std::printf("%d\n", counter);
}
```

https://godbolt.org/z/Pqxd9adG5

Data Races

Increment example – has data race

```
int counter = 0;
int main() {
  std::vector<std::thread> threads;
  threads.reserve(100);
  auto increment = [](){
    for (int i = 0; i < 50; ++i) {
      ++counter;
    }
  };
  for (int i = 0; i < 100; ++i) {
    threads.push_back(std::thread(increment));
  }
  for (std::thread& t : threads) { t.join(); }
  std::printf("%d\n", counter);
}
```

https://godbolt.org/z/Pqxd9adG5

Data Races

Increment example - mutex

```
int counter = 0;
std::mutex counter_mutex;
int main() {
  std::vector<std::thread> threads;
  threads.reserve(100);
  auto increment = [](){
    for (int i = 0; i < 50; ++i) {
      std::scoped_lock lock{counter_mutex};
      ++counter;
    }
  };
  for (int i = 0; i < 100; ++i) {
    threads.push_back(std::thread(increment));
  }
  for (std::thread& t : threads) { t.join(); }
  std::printf("%d\n", counter);
}
```

Data Races

Increment example - atomic

```
std::atomic<int> counter{0};
int main() {
  std::vector<std::thread> threads;
  threads.reserve(100);
  auto increment = [](){
    for (int i = 0; i < 50; ++i) {
      ++counter;
    }
  };
  for (int i = 0; i < 100; ++i) {
    threads.push_back(std::thread(increment));
  }
  for (std::thread& t : threads) { t.join(); }
  std::printf("%d\n", counter.load());
}
```

https://godbolt.org/z/evMacb314

Data Races

Increment example – fewer updates

```
std::atomic<int> counter{0};
int main() {
  std::vector<std::thread> threads;
  threads.reserve(100);
  auto increment = [](){
    int local_counter = 0;
    for (int i = 0; i < 50; ++i) {
      ++local_counter;
    }
    counter += local_counter;
  };
  for (int i = 0; i < 100; ++i) {
    threads.push_back(std::thread(increment));
  }
  for (std::thread& t : threads) { t.join(); }
  std::printf("%d\n", counter.load());
}
```

https://godbolt.org/z/qbnYv7qbY

STD::RECURSIVE_MUTEX

Same thread can lock the mutex multiple times

Number of calls to unlock() must match number of calls to lock()

Useful when some thread-safe APIs call other thread-safe APIs

https://en.cppreference.com/w/cpp/thread/recursive_mutex

STD::TIMED_MUTEX

Additional locking functions:

`try_lock_for(`*duration*`)`

`try_lock_until(`*time*`)`

https://en.cppreference.com/w/cpp/thread/timed_mutex

STD::RECURSIVE_TIMED_MUTEX

recursive_mutex + timed_mutex

https://en.cppreference.com/w/cpp/thread/recursive_timed_mutex

STD::SHARED_MUTEX

a.k.a. read-write mutex

Many threads can get shared ownership, or read access

```
sm.lock_shared();
```

Calls to lock_shared() from other threads will succeed; calls to lock() will block

Only one thread can have exclusive ownership, or write access

```
sm.lock();
```

Calls to lock_shared() or lock() from other threads will block

Useful when there are lots of readers, but few writers

https://en.cppreference.com/w/cpp/thread/shared_mutex

STD::SHARED_TIMED_MUTEX

`shared_mutex` with additional operations:

- `try_lock_for(`*duration*`)`
- `try_lock_until(`*time*`)`
- `try_lock_shared_for(`*duration*`)`
- `try_lock_shared_until(`*time*`)`

https://en.cppreference.com/w/cpp/thread/shared_timed_mutex

STD::SHARED_LOCK

Constructor takes one shared_mutex

- Calls lock_shared() on the shared_mutex

Destructor calls unlock_shared() on the shared_mutex

API is similar to unique_lock

For shared ownership (read access) of a shared mutex, use shared_lock

For exclusive ownership (write access) of a shared mutex, use scoped_lock

https://en.cppreference.com/w/cpp/thread/shared_lock

STD::CONDITION_VARIABLE

Complicated to use correctly

Useful when some threads are waiting for a condition and other threads make that condition true

https://en.cppreference.com/w/cpp/thread/condition_variable

STD::COUNTING_SEMAPHORE

Lightweight synchronization primitive that can control access to a shared resource

Maintains an internal counter

- Calls to `release()` increment the counter
- Calls to `acquire()` decrement the counter, or block if counter == 0

Calls to `acquire()` and `release()` can happen on different threads

https://en.cppreference.com/w/cpp/thread/counting_semaphore

STD::LATCH

Synchronize the completion of a shared task

Each thread calls `arrive_and_wait()`

Blocks until all threads have called `arrive_and_wait()`

https://en.cppreference.com/w/cpp/thread/latch

STD::BARRIER

Reusable latch

Once all threads have called `arrive_and_wait()` and been unblocked, the process starts over

https://en.cppreference.com/w/cpp/thread/barrier

Things Other Than Threads

Processes – same system

Processes on the same system working together

- Communicate through file system
- How do they synchronize?

Things Other Than Threads

Processes – different systems

Processes working together across a network

- How do they communicate?
- Which process has which information?

Things Other Than Threads

GPU threads

GPU threads have different synchronization rules than CPU threads

Synchronization rules depend on type of memory and relationship between threads

Conclusion

Concurrency is hard

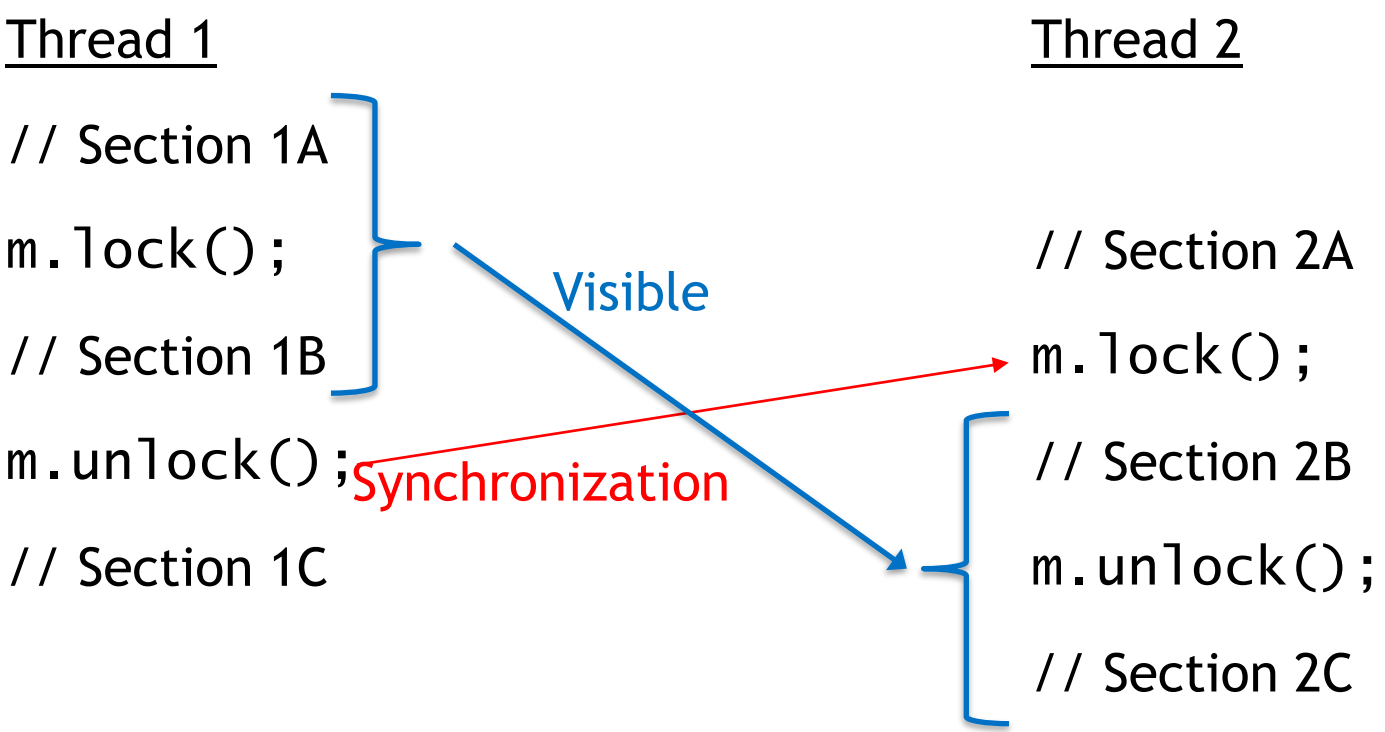
Use parallel algorithms when appropriate

Avoid data races at all costs

- Share less data
- Mutexes and lock guards

STD::MUTEX

Synchronization



Conclusion

Concurrency is hard

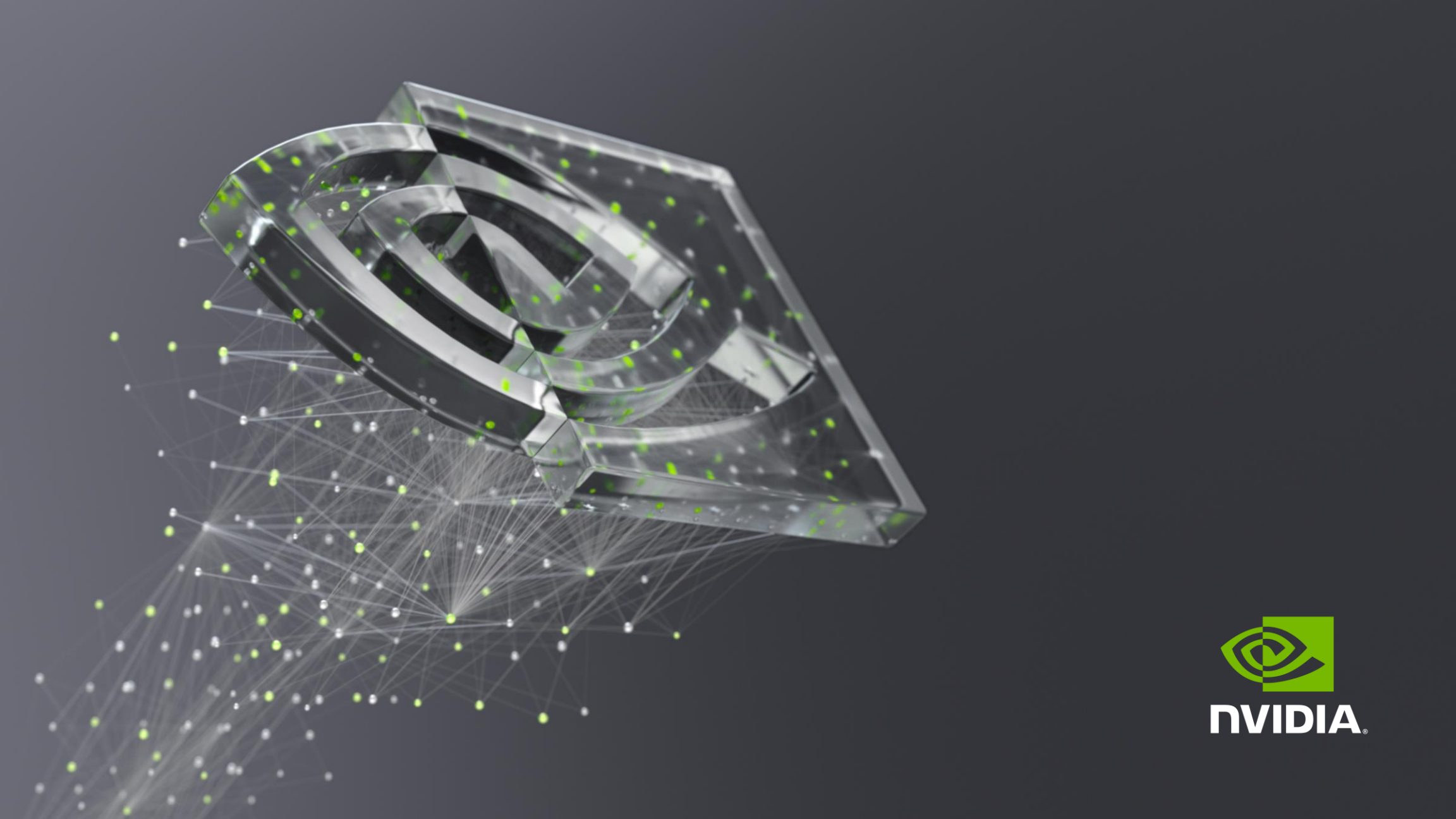
Use parallel algorithms when appropriate

Avoid data races at all costs

- Share less data
- Mutexes and lock guards


3 4


Jetstar.com


NVIDIA