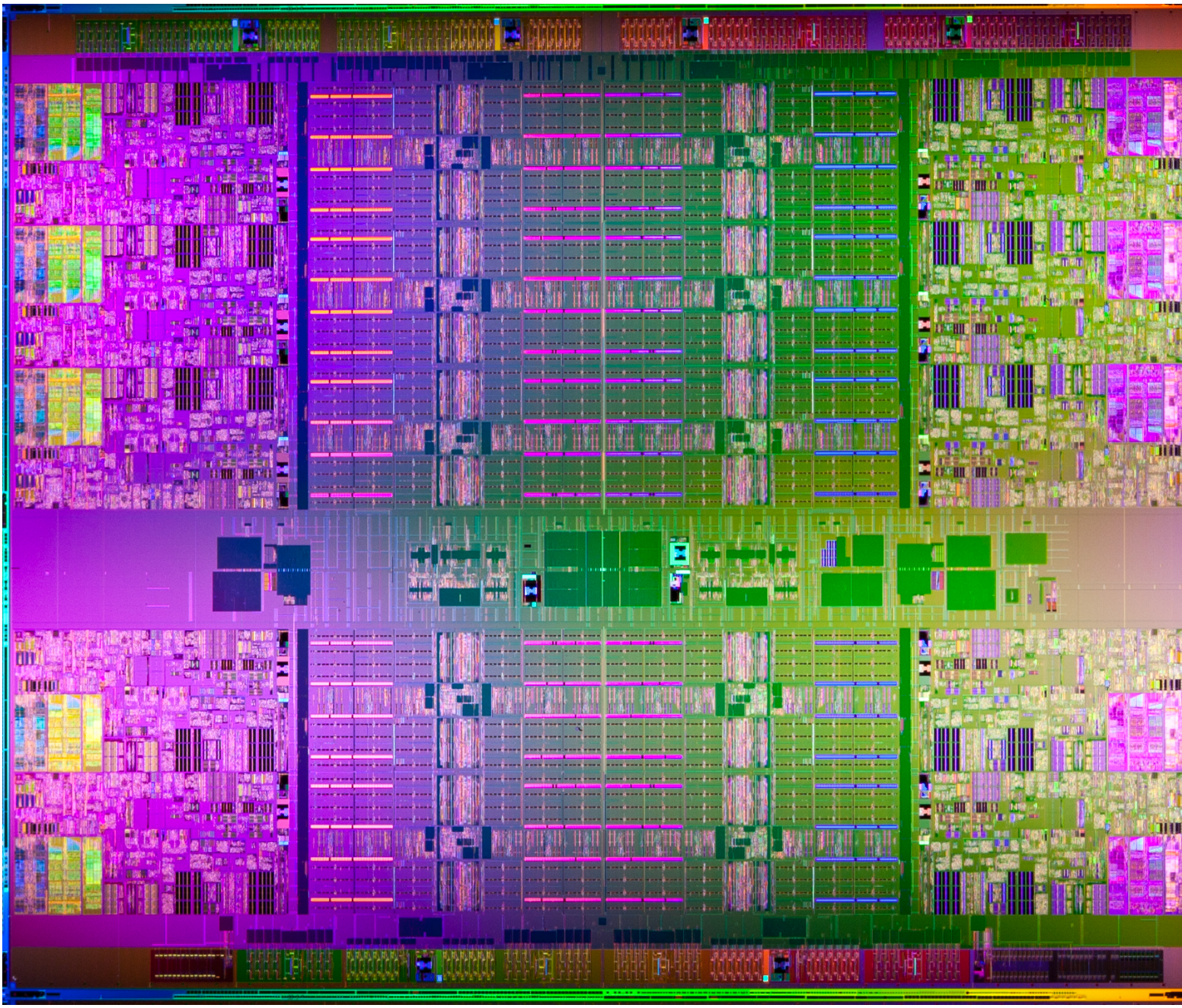


60007

Theory and Practice of Concurrent Programming

Imperial College London

Contents

Chapter 1

Introduction

1.1 Course Structure & Logistics

1.1.1 Structure

Dr Azalea Raad

Prof Alastair Donaldson

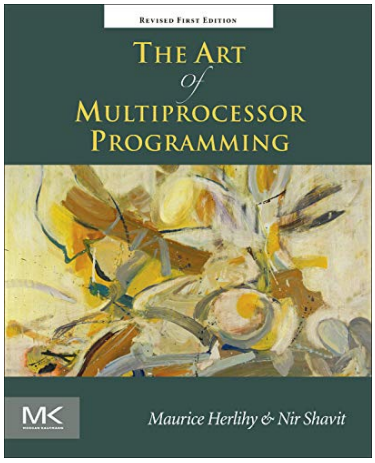
Theory For weeks 2 → 5:

- Intro to synchronisation paradigms (mutual exclusion, readers-writers, producer-consumer)
- Low-level concurrent semantics (sequential consistency, Intel-x86)
- High-level concurrent semantics (concurrent objects, linearisability)
- Transactional memory (serialisability)

Practical For weeks 5 → 8:

- Threads and locks in C++
- Implementing locks
- Concurrency in Haskell
- Race-free concurrency in Rust
- Dynamic data-race detection

1.1.2 Extra Materials



The Art of Multiprocessor Programming
About 65% of the theory course.

1.2 Preface for Concurrency

1.2.1 Moore's Law

Moore's Law — **Definition 1.2.1**

An empirical (supported by observation) law that states the density of transistors in an integrated circuit will double approximately every two years.

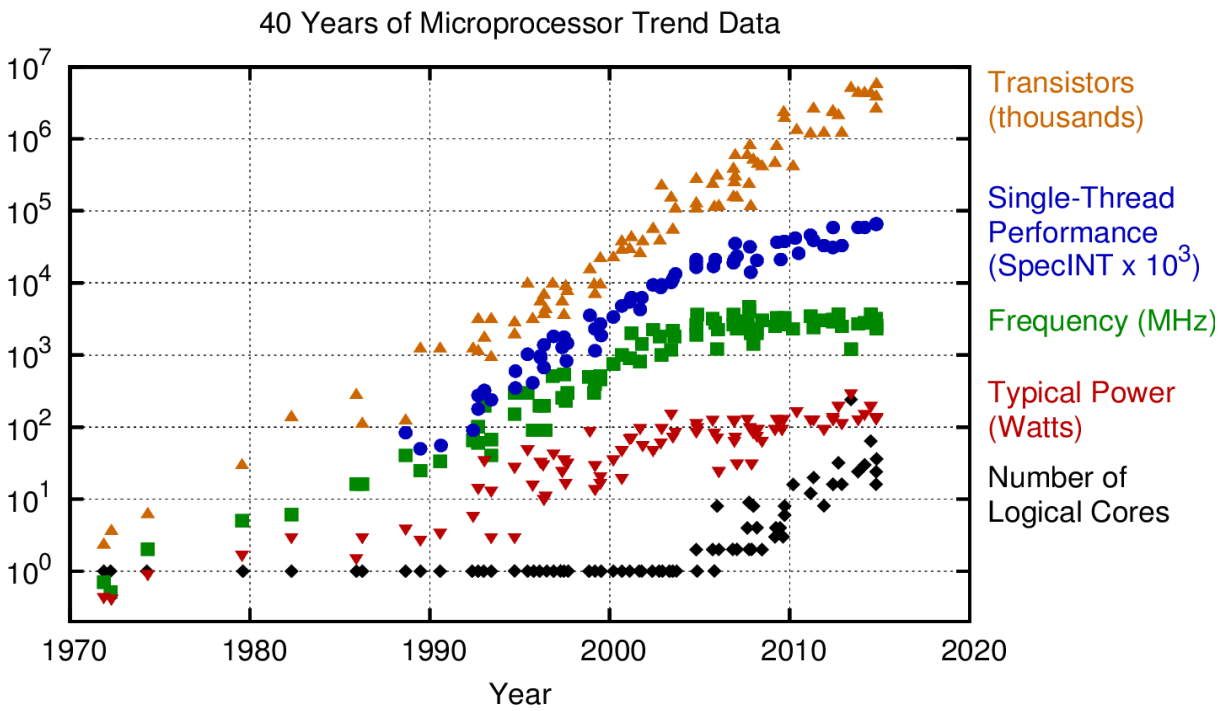
- The observation is named after Gordon Moore (co-founder and later CEO of Intel).
- This law no longer holds, and sequential performance improvements have declined.

Dennard/MOSFET Scaling Definition 1.2.2

$$\text{Power} \propto \text{Transistor Size}$$

A scaling law stating that as transistor density increases, the power requirements stay constant.

- Increasing transistor density results in power staying constant (less power per transistor) and lower circuit delay.
- This allows for higher switching frequency ⇒ higher clocks frequencies ⇒ better sequential performance).



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

The performance improvements typically expected yearly (moore's law and Dennard scaling) no longer apply.

- Sequential (Single-Thread) performance improvements have declined.

- Parallelism is being exploited to improve performance (uniprocessors are virtually extinct).
- Shared-memory multiprocessor systems have lost out to multicore processors.

Amdahl's Law — **Definition 1.2.3**

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}} \text{ where } p = \text{parallel portion and } s = \text{threads}$$

Amdahl's law describes the speedup of a program, associated with the number of threads.

- Can be applied to other resources.
- Versions of the equation exist for different proportions using different numbers of threads.
- As the number of threads increases the sequential part of the program becomes a bottleneck.

1.2.2 Concurrency Difficulties

Writing correct, concurrent code is difficult.

race Condition — **Definition 1.2.4**

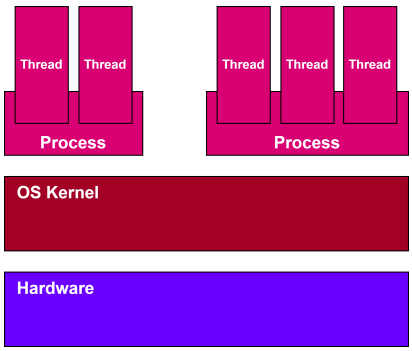
A potential for a situation where the result of a program depends on the non-deterministic timing or interleaving of threads.

- Where multiple threads access data (non-atomically) and at least one writes.
- Where a lack of enforced ordering on some events causes differing results (e.g output to user)

- A process can have multiple threads executing in parallel.
- Cannot determine at compile time the relative speed of execution of threads (many delays are unpredictable; cache misses, page faults, interrupts).
- Cannot predict how long threads will be blocked (e.g I/O) or when threads will be scheduled (or use up their time quantum).

Hence we must use synchronisation mechanisms to regulate accesses to shared data that can result in a race condition.

1.2.3 OS Concepts



- Operating system provides process and thread abstractions.
- A process contains one or more threads (streams of instructions being executed).
- A process has its own address space, all threads in the process share this address space.
- The OS kernel contains a scheduler which schedules processes & their threads.

Chapter 2

Concurrency In C++

2.1 Threads

To interact with threads the `thread` header must be included.

- It provides a standard, implementation independent, interface for handling threads.
- Provides the `std::thread` class

```
#include <thread>

namespace std {
  class thread {
  public:
    // types
    class id;
    using native_handle_type = /* implementation-defined */;

    // construct/copy/destroy
    thread() noexcept;

    // Constructor takes a function to start from, and its arguments (all type checked)
    template<class F, class... Args> explicit thread(F&& f, Args&&... args);

    // Destructor (terminates current thread if the thread has not been joined)
    ~thread();

    // Attempting to copy a thread is not allowed. Hence delete ensures no compile.
    thread(const thread&) = delete;

    // Can create thread from a thread r-value (copy)
    thread(thread&&) noexcept;

    // Attempting to copy a thread (via an immutable reference).
    // This is not allowed, so if this operator is used it will not compile.
    thread& operator=(const thread&) = delete;

    // Assign a thread from an (r value - e.g expression, literal) reference
    thread& operator=(thread&&) noexcept;

    // members
    void swap(thread&) noexcept;
    bool joinable() const noexcept;

    // Wait for this thread to terminate.
    void join();
```

```
    // Allow the thread to continue executing after the thread handler (this)
    // is destroyed
    void detach();

    // Get the unique id of the thread
    id get_id() const noexcept;

    native_handle_type native_handle();

    // static members
    static unsigned int hardware_concurrency() noexcept;
  };
}
```

Lambda — **Definition 2.1.1**

A lambda is a small function that can be defined in an expression, capture values in its scope (and above), and be passed as a value.

```
// [captures] (arguments) {body}

// a basic lambda with no captures
auto my_lambda = [] (int a, int b) -> int {return a + b;}

// using the lambda
int c = my_lambda(1, 2);

auto another_lambda = [c&] (int d) {return c + d;}
```

We can construct using `std::thread`'s constructors.

```
// idiomatic constructor
std::thread my_thread(StartFunction, arg1, arg2, ...)

// call constructor and assign
std::thread my_thread = std::thread(StartFunction, arg1, arg2, ...)
auto my_thread = std::thread(StartFunction, arg1, arg2, ...)

// pass lambda as function
std::thread my_thread(StartFunction, arg1, arg2, ...)
```

When passing arguments to the thread, if these are by reference, a `std::ref` or `std::cref` must be used.

Reference this! — **Example Question 2.1.1**

Given some function `static void some_func(const int& a)` create a thread to take a reference to the number 42.

```
int a = 42;
std::thread my_thread(some_func, std::cref(42));
my_thread.join();
```

2.1.1 Vectors of Threads

When adding an object to a container (e.g a vector) we want to avoid allocating the object, and then moving it into the container.

- Some objects may not be movable/copyable.
- The object should be allocated within the container.

For this we can use emplacement.

```
template< class... Args >
void emplace_back( Args&&... args );
```

Emplace — **Example Question 2.1.2**

Given some function `static void some_func()` create 10 threads and append to the vector using `std::vector::push_back` and another 10 with `std::vector::emplace`.

```
std::vector<std::thread> threads;

for (int i; i < 10; i++) {
  threads.push_back(std::thread(some_func));
}

for (int i; i < 10; i++) {
  threads.emplace_back(some_func);
}

for (auto& t : threads) {
  t.join();
}
```

2.1.2 This Thread

The threads header also provides functionality for interacting with the current thread.

```
#include <compare>

namespace std {
  class thread;

  void swap(thread& x, thread& y) noexcept;

  // class jthread
  class jthread;

  // methods for interacting with the current thread
  namespace this_thread {
    thread::id get_id() noexcept;

    // indicates another thread should be scheduled (e.g long wait expected)
    void yield() noexcept;

    // Sleeping, generic for
    template<class Clock, class Duration>
    void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);

    //
    template<class Rep, class Period>
    void sleep_for(const chrono::duration<Rep, Period>& rel_time);
  }
}
```

Clock watching — **Example Question 2.1.3**

Create program that prints the thread id, and sleeps.

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

```
int main() {
  using namespace std::chrono_literals; // to use the ms syntax

  std::cout << std::this_thread::get_id() << " will sleep now!" << std::endl;
  std::this_thread::sleep_for(200ms);

  std::cout << std::this_thread::get_id() << " has awoken!" << std::endl;
}
```

2.2 Locks

RAII **Definition 2.2.1**

Resource Acquisition Is Initialization (also called Scope-Bound Resource Management and Constructor Acquires, Destructor Release) is where a resource's allocation and release is bound to the lifetime of an object.

- a resource may be the memory allocated to an object, or resources such as os provided file handlers.
- When the object goes out of scope (e.g the variable owning the object is destroyed) the resource is released.
- In C++, when a variable goes out of scope, the destructor of the contained object is called, so the destructor must release the resources.
- This concept is heavily embedded in Rust. Lifetimes are a major part of the type system, and ownership rules are enforced by the compiler.
- RAII is used for smart pointers such as `Rc` in rust or `std::shared_ptr`.

```
static void my_scope() {
    MyClass my_object;  // initialised, default constructor called

    // ... do some stuff ...

    return; // destructor my_object.~MyClass() called.
}
```

The `mutex` header contains locks for synchronisation.

```
namespace std {
  class mutex;                  // A regular lock
  class recursive_mutex;        // reentrant/recursive lock
  class timed_mutex;            // A mutex with timeout
  class recursive_timed_mutex;  // A recursive mutex with timeout

  /* used to set the locking strategy when using lock guards
   * e.g create guard (that releases lock on destruction) assuming
   * lock is held.
   */
  struct defer_lock_t { explicit defer_lock_t() = default; };   // do not acquire ownership
  struct try_to_lock_t { explicit try_to_lock_t() = default; }; // try to acquire ownership (no block)
  struct adopt_lock_t { explicit adopt_lock_t() = default; };   // assume calling thread has ownership

  inline constexpr defer_lock_t  defer_lock { };
  inline constexpr try_to_lock_t try_to_lock { };
  inline constexpr adopt_lock_t  adopt_lock { };

  // A RAII like mechanism that releases the lock it guards when destroyed.
  template<class Mutex> class lock_guard;

  // A RAII style lock guard, when taking ownership of multiple locks it attempts
  // deadlock avoidance.
```

```
  template<class... MutexTypes> class scoped_lock;

  // A movable lock guard.
  template<class Mutex> class unique_lock;

  template<class Mutex>
    void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y) noexcept;

  // attempts to acquire locks from references provided, returns index (in args) of lock
  // that could not be acquired.
  template<class L1, class L2, class... L3> int try_lock(L1&, L2&, L3&...);

  // Acquire one or more locks (blocking) and use deadlock avoidance.
  template<class L1, class L2, class... L3> void lock(L1&, L2&, L3&...);

  struct once_flag;

  template<class Callable, class... Args>
    void call_once(once_flag& flag, Callable&& func, Args&&... args);
}
```

2.2.1 Using Mutexes

```
namespace std {
  class mutex {
    public:

      // Constructor initialises mutex as unlocked. It is a constexpr
      // as can determine all fields at compile time.
      constexpr mutex() noexcept;

      // Destructor, undefined behaviour if the mutex is held by a thread.
      ~mutex();

      // Cannot create mutex from another, or use assignment to move a mutex.
      mutex(const mutex&) = delete;
      mutex& operator=(const mutex&) = delete;

      void lock();
      bool try_lock();
      void unlock();

      using native_handle_type = /* implementation-defined */;
      native_handle_type native_handle();
  };
}
```

Locked Out — **Example Question 2.2.1**

Create a mutex to protect a counter, and use 100 threads to increment the counter 10 times each. Add a wait of 1ms between each increment and only lock for each increment.

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

int cnt;
std::mutex cnt_lock;
```

```
static void increment_cnt() {
  for (int i = 0; i < 10; i++) {
    std::this_thread::sleep_for(std::chrono::milliseconds(1));
    cnt_lock.lock();
    cnt++;
    cnt_lock.unlock();
  }
}

int main() {
  std::vector<std::thread> threads;

  for (int i = 0; i < 100; i++) {
    threads.emplace_back(increment_cnt);
  }

  for (auto& t : threads) {
    t.join();
  }

  std::cout << "The counter is: " << cnt << std::endl;
}
```

Scoped Locks

We can use `scoped_lock`, `unique_lock` or `lock_guard` to link the time the lock is held to the lifetime of the lock guard object.

- Each has slight differences, separate implementations are provided rather than using complex template magic.
- Deadlock avoidance is used to ensure all threads acquire and release locks in the same order.

```
namespace std {
  template<class... MutexTypes>
  class scoped_lock {
  public:
    using mutex_type = Mutex;   // If MutexTypes... consists of the single type Mutex

    explicit scoped_lock(MutexTypes&... m);
    explicit scoped_lock(adopt_lock_t, MutexTypes&... m);
    ~scoped_lock();

    scoped_lock(const scoped_lock&) = delete;
    scoped_lock& operator=(const scoped_lock&) = delete;

  private:
    tuple<MutexTypes&...> pm;   // exposition only
  };
}
```

Scoped out **Example Question 2.2.2**

Create a basic implementation of defer that can be used for a scoped lock.

```
#include <mutex>
#include <iostream>
#include <functional>

class Defer {
  private:
```

```
    std::function<void(void)> function_;

  public:
    Defer(std::function<void(void)> fun) : function_(fun) {}
    ~Defer() {
      function_();
    }
};


int main() {
  std::mutex m;

  Defer lock([&m] () {
    m.unlock();
    std::cout << "Unlocking" << std::endl;}
  );

  std::cout << "lets do some racey stuff here" << std::endl;
}
```

We could also implement this pattern in rust. As mutexes already work this way in rust, we create a dummy mutex struct to use.

```
#![feature(fn_traits)]
struct Defer<F: FnMut()>(F);

impl<F: FnMut()> Drop for Defer<F> {
    fn drop(&mut self) {
        self.0()
    }
}

fn main() {
    let mut m = Mutex();
    m.lock();
    let _d = Defer(|| m.unlock());
    println!("lets do some racey stuff here")
}
```