Concurrency

N.B.

- All this requires C++11 (or more recent)
- Much more detail e.g. in “Concurrency in Action” (also source of examples)

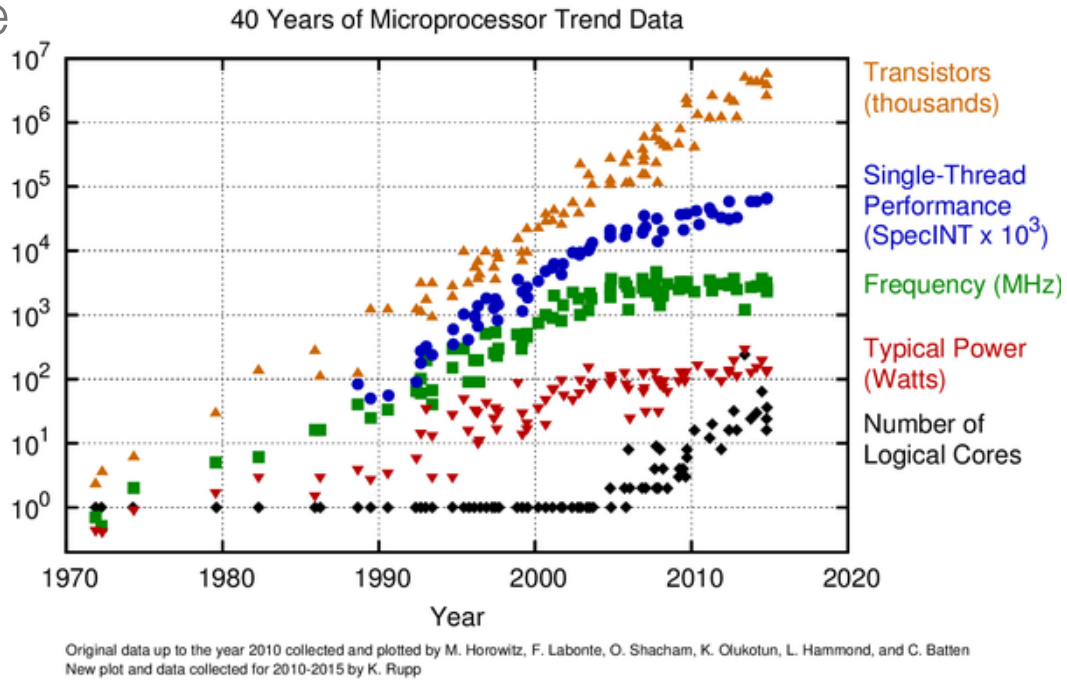
Need for concurrency

Dictionary: “the fact of having two or more events or circumstances happening or existing at the same time”

- in computing context: simultaneous execution of multiple tasks
- not to be confused with seemingly simultaneous processes on a single CPU core: these are in fact switched between (scheduling done by kernel)

Nowadays, many computers come with multi-core CPUs: allows for distribution of work over multiple cores

- relevant as the most obvious way to improve computing performance is to use more CPU cores
  - breakdown of Moore’s law



Thread basics

System level representation of a “task” within a program

```
#include <thread>
#include <string>
#include <iostream>
using namespace std;

void f1() {
  cout << "Hello ";
}

void f2(const std::string& s) {
  cout << s << endl;
}

int main() {
  thread t1(f1);
  thread t2{f2, "Parallel World!"};
  t1.join();
  t2.join();
  return 0;
}
```

function w/o arguments

- NB alternatively a function object could be used (see next slide)
- generally not efficient to use more threads than there are cores (or twice that, if hyper-threading is used on the machine)

Thread basics

System level representation of a “task” within a program

```
#include <thread>
#include <string>
#include <iostream>
using namespace std;

void f1() {
  cout << "Hello ";
}

void f2(const std::string& s) {
  cout << s << endl;
}

int main() {
  thread t1(f1);
  thread t2{f2, "Parallel World!"};
  t1.join();
  t2.join();
  return 0;
}
```

function with arguments

- NB alternatively a function object could be used (see next slide)
- generally not efficient to use more threads than there are cores (or twice that, if hyper-threading is used on the machine)

Thread basics

System level representation of a “task” within a program

```
#include <thread>
#include <string>
#include <iostream>
using namespace std;

void f1() {
  cout << "Hello ";
}

void f2(const std::string& s) {
  cout << s << endl;
}

int main() {
  thread t1(f1);
  thread t2{f2, "Parallel World!"};
  t1.join();
  t2.join();
  return 0;
}
```

aggregate initialisation (uses “variadic template constructor”: free arguments)

- NB alternatively a function object could be used (see next slide)
- generally not efficient to use more threads than there are cores (or twice that, if hyper-threading is used on the machine)

Thread basics

System level representation of a “task” within a program

```
#include <thread>
#include <string>
#include <iostream>
using namespace std;

void f1() {
  cout << "Hello ";
}

void f2(const std::string& s) {
  cout << s << endl;
}

int main() {
  thread t1(f1);
  thread t2{f2, "Parallel World!"};
  t1.join();
  t2.join();
  return 0;
}
```

wait for threads to finish; omitting join() will lead to run-time error

- NB alternatively a function object could be used (see next slide)
- generally not efficient to use more threads than there are cores (or twice that, if hyper-threading is used on the machine)

Thread basics: modifying data

To do something useful with threads, they need data

- following example: two ways of passing a non-const reference
  - function object also makes it easy to store any results

```
#include <functional>

void manipulate(vector<double>& v) {
    …
}

struct my_f {
    vector<double>& v;
    my_f(vector<double>& vv): v(vv) {}
    void operator()();
    void my_function();
}

vector<double> v {0., 1., 2., 3.14159};
thread t {manipulate, std::ref(v)};

vector<double> v2(v), v3(v);
thread t2 {my_f(v2)};
my_f f;
thread t3 (&my_f::my_function, f);
```

struct storing a reference (!)

use std::ref() to indicate explicitly the use of a reference (otherwise the assumption is that arguments will be passed by value — and compilation will fail)

Thread ownership and “move semantics”

Threads cannot be copied (what would this mean anyway, given that they start executing a “function” as soon as they are instantiated?)

- but it is possible to transfer ownership of a thread using its move constructor: `thread (thread&& t)`
  - this uses the concept of rvalue reference (denoted by &&), which we will not elaborate on more (except to say that one can typically use it as the object itself, and it is typically used for temporary objects)
  - move constructors swap resources; they may be compiler generated
  - move semantics: may be used to prevent unnecessary copying
    - to be compared with ex 2.1
    - especially useful in case of pointer data members

```
template<class T>
swap(T& a, T& b) {
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}
```

std::move() effectively turns its argument into an rvalue reference

Thread ownership (continued)

Effect of the preceding discussion: threads can be returned to a calling function, can be entered in containers etc.

- example:

```
void do_work(unsigned int i) {
    …
}

vector<thread> threads;
for (unsigned int i = 0; i < N; ++i) {
    threads.push_back(thread(do_work, i));
}
for (unsigned int i = 0; i < N; ++i) {
    threads[i].join();
}
```

under the hood: move constructor called

- N.B. individual threads can still be identified by ID:

```
thread t(do_work, 0);
thread::id id = t.get_id();
…
if (this_thread::get_id() == id) {
    …
}
```

within a thread, and assuming it is informed of id

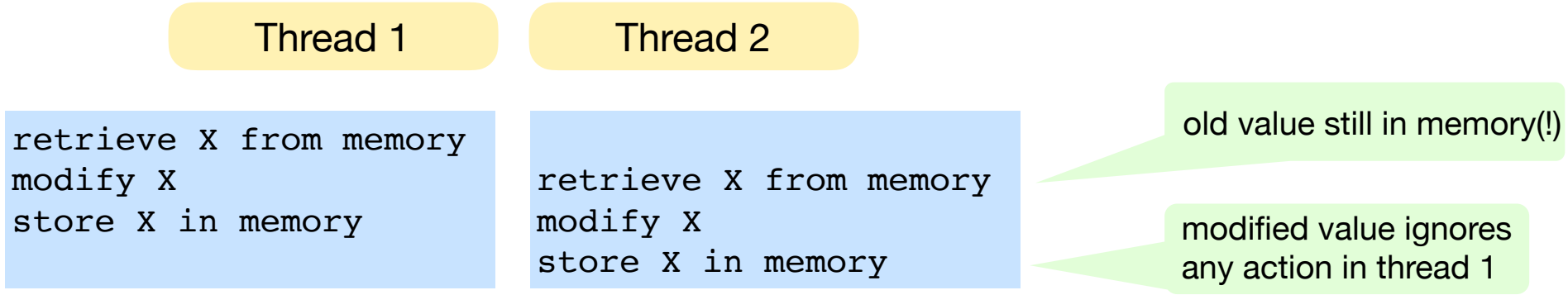
Shared use of resources

Running the program on p11 will lead to ill-defined results: shared use of std::cout

- order of print statements not well defined, or could even be garbled

Example of a general problem when resources are shared between threads, if at least one of them has write access

- even a single operation on a basic type involves multiple steps (loading into cache, from there into register, actual manipulation, then moving back through cache to memory), allowing for data race



Mutual Exclusion object locking: mutex

std::mutex: basic locking mechanism in C++

- imagine that functions below are executed in different threads

```
#include <mutex>
#include <algorithm>

list<int> some_list;
mutex some_mutex;

void add_to_list(int new_value) {
    lock_guard<mutex> guard(some_mutex);
    some_list.push_back(new_value);
}

bool list_contains(int value_to_find) {
    lock_guard<mutex> guard(some_mutex);
    return (std::find(some_list.begin(),some_list.end(),value_to_find)
        != some_list.end());
}
```

mutex object intended to "protect" list object

lock_guard also defined in <mutex>: calls mutex::lock() when instantiated, mutex::unlock() when going out of scope

lock here prevents list from being read while being modified

- use of lock_guard mainly offers code robustness: no need to worry about unlocking mutex object

Giving a mutex object a proper place

Extracted from the preceding page:

```
list<int> some_list;
mutex some_mutex;
```

- association of mutex object with data object is entirely by convention!

In an object oriented world, better coherence can be achieved by embedding both the data object and the mutex in a wrapper class:

```
class my_data {
    …
public:
    manipulate();
};

class my_wrapper {
    my_data d; mutex m;
public:
    void process() {
        lock_guard<mutex> g(m); d.manipulate();
    }
    my_data& retrieve() {
        lock_guard<mutex> g(m); return d;
    }
};
```

danger! reference (or pointer) to data object is again unprotected (despite the mutex lock)

Deadlock

Locking does not come without its own pitfalls..

- it may happen that access is required by multiple threads to two (or more) resources, which both need to be protected by a mutex lock
- this may lead to the opposite problem compared to a data race: deadlock
  - if the threads in this example have acquired their mutex lock at the same time, they will wait indefinitely for the other to release its lock

Thread 1	Thread 2

`lock mutex 1`
`(some code)`
`lock mutex 2`
	`lock mutex 2`
	`(some code)`
	`lock mutex 1`

Deadlock: how to avoid it

Recommendation: do not attempt to acquire a lock if you already have one (i.e., if multiple ones are needed, acquire them simultaneously)

```
mutex m1, m2;
…
std::lock(m1, m2);
lock_guard g1(m1, std::adopt_lock);
lock_guard g2(m2, std::adopt_lock);
```

simultaneous acquisition of both locks

to deal with unlocking without attempting to lock once again

- in this case the mutex could alternatively be used for both resources; in general, too coarse granularity can slow code down unduly

If it is not possible to acquire the locks simultaneously, it will help to always acquire them in the same order

- may be easier said than done

Re-entrant functions and deadlock

With a standard (non-recursive) mutex, the following (variadic!) re-entrant function would block itself

```
#include <mutex>
recursive_mutex rm;

template<typename Arg,typename… Args>
void write(Arg a, Args tail…) {
    lock_guard<recursive_mutex> g(rm);
    cout << a;
    write(tail…);
}
```

- shown here for completeness; often other alternatives to re-entrant code are possible
  - yet more complete: timed_mutex (with timeout), unique_lock (with additional functionality, e.g., can be copied)

Finally: locking is non-trivial to get right (and also takes time)

- useful to try to avoid or minimise its use

Task synchronisation between threads

It may be useful for threads to be able to communicate (one thread waiting for something in another thread to happen); mutex locks can be used for this, but “raw” use generally isn’t efficient

- continuous checking of mutex keeps CPU busy
- alternative of sleeping / waking up periodically may be better.. but how to determine the period?

std::condition_variable deals with this in a CPU-friendly way

```
#include <condition_variable>
…
bool some_condition;
mutex m;
condition_variable cv;
…
void some_consumer() {
    unique_lock<mutex> l(m);
    cv.wait(l, []{ return condition; });
    …
}

void some_producer() {
    lock_guard l(m);
    condition = true;
    cv.notify_one();
}
```

`some_condition`: condition of interest

use unique_lock rather than lock_guard as is copied in call to wait()

“lambda expression” may be evaluated repeatedly until true

notify thread waiting for release of lock; NB also a version notifying all threads waiting for this lock release exists