TECHNISCHE UNIVERSITÄT DRESDEN

DEPARTMENT OF COMPUTER SCIENCE
INSTITUTE OF COMPUTER ENGINEERING
CHAIR OF COMPUTER ARCHITECTURE
PROF. DR. WOLFGANG E. NAGEL

Hauptseminar
“Rechnerarchitektur und Programmierung”

Innovations in C++11, 14 and 17 in the Context of Performance Analysis of HPC Applications

Jan Stephan
(Mat.-No.: 3755136)

Professor: Prof. Dr. Wolfgang E. Nagel
Tutor: Ronny Brendel

Dresden, 30th of September 2016

Contents

1 Introduction

With the release of the C++11 standard in 2011 the C++ community was introduced to several new language and library features [3]. Together with a new programming philosophy (see Section 1.2) these changes were so extensive that the original creator of the C++ programming language, Bjarne Stroustrup, felt like it was a “new language” [13].

Because of the newly added concurrency and synchronisation primitives modern C++ is particularly interesting for High-Performance Computing (HPC) and performance analysis tools like Score-P [9] and Vampir [1]. Hence, the contributions of this report are

- the presentation of modern C++ features relevant in a context of HPC and performance analysis,
- the evaluation and analysis of these feature’s influence on the aforementioned context,
- the implementation of example programs making use of these features and their analysis with Score-P and Vampir and finally
- recommendations for enabling Score-P and Vampir to support those features they currently can not instrument correctly.

1.1 C++11 and C++14

The C++11 standard [3] was published in 2011, thirteen years after the last major standard C++98 and eight years after the last minor standard C++03. While it maintained near-perfect backward compatibility [6] it introduced a lot of new features to the core language and the standard library.

The current C++14 standard [4] brought fewer changes to both the language and the library compared to its predecessor and can be considered a “bugfix” to C++11. Because of this the contents of the sections below are focused on C++11, mentioning C++14 where changes to the C++11 standard can be applied.

1.2 The Modern C++ Philosophy

Apart from language standardisation the Standard C++ Foundation also advertises a new programming philosophy by suggesting more abstraction, using the Resource Acquisition is Initialization (RAII) pattern and relying on the Standard Template Library (STL) in almost all situations. In support of this cause, Bjarne Stroustrup (creator of C++) and Herb Sutter (president of the Standard C++ Foundation) are working on the detailed *C++ Core Guidelines*, the current draft being publicly available [14]. The code examples in this document are following these guidelines.

2 Modern C++ by example

This section introduces some features of modern C++ by example. It focuses on the features most important in the context of HPC and performance analysis, i.e. concurrency, synchronisation and memory management. The first example (Section 2.1) shows the implementation of a simple CUDA kernel and its surroundings in modern C++, the second example (Section 2.2) a thread-safe producer/consumer queue.

2.1 CUDA Vector Addition

Using the CUDA toolkit, this subsection introduces some of the features supported by the C++11 and C++14 standards. It focuses on additions to the core language which improve maintainability and the stating of the programmer's intent as well as dynamic memory management.

2.1.1 The Kernel

Support by the CUDA runtime for a subset of C++11 has been available since CUDA 7. A kernel which implements a vector addition looks like the following code:

```
__global__ void vector_add(const std::int32_t* a, const std::int32_t*
 ↪  b, std::int32_t* c, std::size_t size)
{
    auto i = threadIdx.x + blockIdx.x * blockDim.x;
    if(i < size) c[i] = a[i] + b[i];
}
```

There are two notable differences to traditional code: firstly the usage of `std::int32_t` instead of plain `int` and secondly automatic type deduction with the keyword `auto`. Both are presented in the following paragraphs.

Fixed-Width Integer Types Instead of `int` the kernel parameters are of the type `std::int32_t`. The latter is called a *fixed-width integer type* and was introduced with the C++11 standard [3](§18.4.1) which in turn based this inclusion on the C99 standard [2](§7.18). These types (ranging from 8-bit to 64-bit types, both signed and unsigned), along with counterparts optimized for size or performance, can be found in the header file `<cstdint>`.

The `auto` Keyword In C++11 the already existing keyword `auto` changed its meaning. Prior to C++11 it denoted the storage class of a variable, a property inherited from C [2](§6.7.1):

```
void f()
{
    auto int i; // local variable, lives until end of scope
    static int j; // static variable, permanent duration
}
```

Because the usage of this keyword was largely redundant, its meaning was changed in C++11 [3](§7.1.6.4) to deduce the type of an expression:

```
auto i = 0; // i is deduced to be an int
auto f1 = float(0); // committing to a specific type
auto f2 = 0.f; // committing to a specific type, shorthand
auto two = std::sqrt(4); // type deduction from return value
```

`auto` can also be used in combination with functions:

```
auto f() -> void;

template <class T, class U>
auto add(T a, U b) -> decltype(a + b)
{
    return a + b;
}
```

The *trailing return type* syntax is especially needed when used with templates, as the second example shows. `decltype` is the counterpart to `auto` which returns the type of an expression (must be evaluable at compile-time).

Since C++14 the return type is no longer needed as long as the function body is visible to the compiler and the function itself is not recursive:

```
template <class T, class U>
auto add(T a, U b)
{
    return a + b;
}
```

According to Herb Sutter [15] one should follow an *Almost Always Auto (AAA)* pattern, i.e. use `auto` wherever possible. It can also be shown that the adoption of this practice leads to better performing and more secure code [16].

2.1.2 Host Memory Management

Vector Creation Before the kernel can be executed the input vectors need to be initialised, which is typically done on the host. Before the adoption of C++11 the usual approach would have been

- the definition of a size constant using the preprocessor,
- memory allocation with `new` or `malloc`,
- memory initialisation with a `for`-loop,
- after kernel execution, freeing memory with `delete` or `free`.

However, this is not idiomatic and contradicts the design philosophy mentioned in Section 1.2. A more modern approach looks like the following code:

```
constexpr auto size = 1000;

auto host_a = std::make_unique<std::int32_t[]>(size);
auto host_b = std::make_unique<std::int32_t[]>(size);

std::generate(a.get(), a.get() + size, std::rand);
std::generate(b.get(), b.get() + size, std::rand);

// ...
```

This approach offers several advantages: the need for the preprocessor is eliminated (see the `constexpr` paragraph in this section), the memory is automatically `free`'d once the pointers leave their scope (see the "Smart Pointers" paragraph in this section) and the usage of `std::generate` expresses intent much better than a C-style `for`-loop.

The `constexpr` Keyword `constexpr` is a new keyword introduced in C++11 [3](§20.7). Its purpose is to provide a hint to the compiler that an expression or function can be evaluated at compile-time:

```
constexpr auto i = 0;

constexpr auto max(std::int32_t a, std::int32_t b)
{
    return (a > b) ? a : b;
}

constexpr auto i = max(2, 3);
auto j = max(x, y);
```

Note that the last expression might be executed at runtime, unless `x` and `y` are known at compile-time. However, as `constexpr` functions are usually small the compiler is likely to inline the function or optimise the call out anyway.

The benefits are not obvious when comparing `constexpr` constants to preprocessor constants but they become clearer when looking at `constexpr` functions. Like a lot of other novelties introduced with C++11 `constexpr`'s primary purpose is to state *intent*. Assume that the following function was provided by developer A:

```
auto sum(std::uint32_t n) -> std::uint32_t
{
    return n > 0 ? n + sum(n - 1) : n;
}
```

If `n` is known at compile-time the compiler can easily prove that the function as a whole can be evaluated at compile-time as well, leading to the following:

```
float arr[sum(10)]; // works
```

Somewhat later developer B decides to change the function in order to make it print the result:

```
auto sum(std::uint32_t n) -> std::uint32_t
{
    auto res = n > 0 ? n + sum(n - 1) : n;
    std::cout << res << std::endl;
    return res;
}
```

Now every context in which `sum` was evaluated at compile-time is broken[1]. However, this is the fault of developer A as he never promised that the function was usable in such a context. With `constexpr` he could enforce his intent and prevent other developers from breaking the code base because the compilation would fail immediately.

Smart Pointers Smart pointers are an addition to the standard library introduced in C++11 [3](§20.7). Currently there are three variants:

- `unique_ptr`
- `shared_ptr`
- `weak_ptr`

At the same time the predecessor `auto_ptr` was marked as deprecated [3](§D.10).

- `unique_ptr` is used when there is only one owner of the pointer, for example in a function-local context, as a class member variable or as nodes of a tree.
- `shared_ptr` is the counterpart for situations in which shared ownership is needed, for example nodes in a graph where each node can have multiple parents.
- `weak_ptr` is a “pointer to pointer”-like construct as it has to be converted to a `shared_ptr` before being able to access the memory.

Memory is allocated in the following way:

```
auto unique_obj = std::unique_ptr<my_class>{new my_class(args)};
auto unique_arr = std::unique_ptr<std::int32_t[]>{new
 ↪  std::int32_t[size]};
// make_unique was introduced in C++14
auto unique_obj = std::make_unique<my_class>(args);
auto unique_arr = std::make_unique<std::int32_t[]>(size);

auto shared_obj = std::make_shared<my_class>(args);
auto shared_arr = std::make_shared<std::int32_t[]>(size);

auto weak_obj = std::weak_ptr<my_class>(shared_obj);
auto weak_arr = std::weak_ptr<std::int32_t[]>(shared_arr);
```

[1]GCC 6.2.1 and clang 3.8.1 accept this code. Note that this behaviour is not conforming to the C++ standard. When activating the `pedantic` flag both compilers will warn about the usage of this extension.

The big benefit of smart pointers becomes clear when looking at how memory is free'd: This is done automatically once the pointer leaves its scope (unique_ptr) or the internal reference counter is set to zero (shared_ptr). This essentially makes new and delete obsolete with no (unique_ptr) to low (shared_ptr because of reference counting) overhead.

2.1.3 Device Memory Management

After the host memory has been initialised the vectors have to be copied to device memory in order to be processed by the kernel. Traditionally one would

- allocate device memory using cudaMalloc,
- copy the memory from host to device and back to the host after kernel execution using cudaMemcpy and
- finally free the memory using cudaFree.

As this is plain C style it does not correspond to the modern design philosophy mentioned in Section 1.2. The process is more idiomatic if the calls to cudaMalloc and cudaFree are wrapped in order to use them in a fashion similar to the host memory functions described in Section 2.1.2:

```
auto dev_a = make_device_ptr<std::int32_t>(size);
auto dev_b = make_device_ptr<std::int32_t>(size);
```

This approach also removes the explicit call to cudaFree which would be needed in multiple locations otherwise in order to catch CUDA runtime errors and behave accordingly. The make_device_ptr function in this example is implemented as follows:

```
struct device_deleter {
    auto operator()(void* p) -> void { cudaFree(p); }
};

template <class T>
using device_ptr = std::unique_ptr<T[], device_deleter>;

template <class T>
auto make_device_ptr(std::size_t size) -> device_ptr<T>
{
    auto p = static_cast<T*>(nullptr);
    cudaMalloc(reinterpret_cast<void**>(&p), size * sizeof(T));
    return device_ptr<T>(p);
}
```

Note the using directive above which is a new way of declaring typedefs. This keyword is presented later in this section.

Once the device memory is allocated, it can be copied from the host to the device. Because the smart pointer itself cannot be passed to cudaMemcpy, the "raw" pointer needs to be accessed which can be done by using the get member function:

```
auto bytes = size * sizeof(std::int32_t);
cudaMemcpy(dev_a.get(), host_a.get(), bytes, cudaMemcpyHostToDevice);
cudaMemcpy(dev_b.get(), host_b.get(), bytes, cudaMemcpyHostToDevice);
// ...
cudaMemcpy(host_c.get(), dev_c.get(), bytes, cudaMemcpyDeviceToHost);
```

For the sake of clarity, error handling is omitted. However, in a real-world application one would need to catch all CUDA runtime errors which are returned by calls to all CUDA functions. If an error occurs the allocated memory (both host and device) needs to be `free`'d before returning, or else memory is leaked. This adds a lot of "boilerplate" code which can be avoided by relying on smart pointers.

The `using` Keyword The `using` keyword was adapted to serve as another form of `typedef` in C++11 [3](§7.1.3/2). It provides some benefits over the classic `typedef` as the following examples show:

```
typedef int my_type;
using my_type = int;

typedef void (*func_ptr)(double);
using func_ptr = void (*)(double);

// not possible with typedef
template <class T>
using my_vec = std::vector<T, my_allocator>;
```

While the first example does not change much, the second example shows that function pointer definitions can be expressed a lot clearer because the type name is now separated from the definition. Additionally, the ability to define templated `typedef`s is a big advantage of its own as this was not possible before and adds a lot more flexibility to the language.

This is also coherent with the new `auto` style of variable definitions, i.e. the separation of the variable or type name and the corresponding definition. This is becoming a pattern of itself and is increasingly advertised by C++ experts [16].

```
auto var = 0;
using type = int;
```

2.1.4 Launching the Kernel

Once all the preparations are done the kernel can be executed. Usually one would call the kernel, specify the grid and block dimensions and pass the parameters. Because there are small kernels for which a manual configuration of the dimensions are useless (as it would not provide a performance benefit), it would be much better to have a simple `cuda_launch` function. This function would use mechanisms provided by the CUDA runtime to calculate these dimensions. In this example `cuda_launch` is implemented in the following way:

```
template <class... Args>
auto cuda_launch(void (*kernel)(Args...), Args... args) -> void
{
    // omitted: calculate grid_size and block_size
    kernel<<<grid_size, block_size>>>(args...);
}
```

By making use of *variadic templates* we can pass any kernel with an arbitrary number of arguments to `cuda_launch` which will determine the dimension configuration and then execute the kernel accordingly.

Variadic Templates Variadic templates are a language extensions introduced in C++11 [3](§14.2.15). They are somewhat similar to variadic functions known from C (see `printf` for an example), except that their *parameter packs* are unpacked at compile-time which leads to different function signatures for different parameters. Additionally, they are usable outside of functions as well, for example in template meta programming.

2.1.5 Summary

The following program is the result of combining the modernisations mentioned in the previous sections:

```
constexpr auto size = 1000;

auto host_a = make_unique<std::int32_t[]>(size);
auto host_b = make_unique<std::int32_t[]>(size);
auto host_c = make_unique<std::int32_t[]>(size);

auto dev_a = make_device_ptr<std::int32_t>(size);
auto dev_b = make_device_ptr<std::int32_t>(size);
auto dev_c = make_device_ptr<std::int32_t>(size);

std::generate(host_a.get(), host_a.get() + size, std::rand);
std::generate(host_b.get(), host_b.get() + size, std::rand);

auto bytes = size * sizeof(std::int32_t);
cudaMemcpy(dev_a.get(), host_a.get(), bytes, cudaMemcpyHostToDevice);
cudaMemcpy(dev_b.get(), host_b.get(), bytes, cudaMemcpyHostToDevice);

cuda_launch(vec_add, dev_a.get(), dev_b.get(), dev_c.get(), size);

cudaMemcpy(host_c.get(), dev_c.get(), bytes, cudaMemcpyDeviceToHost);
```

This is more idiomatic and cleaner than its C equivalent while providing the benefits of RAII with regard to host and CUDA device memory. By providing own versions of `unique_ptr` with additional metadata it would be easy to wrap the calls to `cudaMemcpy` and remove the need for size calculations and `enum` values for the copy direction.

2.2 Producer/Consumer Queue

A simple producer/consumer queue is implemented in the following sections to showcase the modern C++ concurrency features.

2.2.1 C++11 Thread Support

Before the new C++ standard was published in 2011 there was no standardised way to create and manage threads in a platform independent manner. The programmer had to resort to mechanisms specific to the targetted operating system, e.g. Pthreads [8] or Win32 threads [12], or third-party libraries like Qt [10]. Furthermore these mechanisms are not idiomatic as they are usually plain C interfaces. This means there was no native support for RAII, making wrappers around these interfaces necessary. Additionally, there was no platform-independent support for atomic operations.

With the adoption of C++11 the situation changed: the new standard introduced a thread support library [3](§30) and an atomic operations library [3](§29). Both will be briefly introduced in the following sections.

2.2.2 High-level Thread Creation

The easiest way to spawn concurrent threads in C++11 is to not do it by oneself. Instead, the highest-level approach C++11 offers is a call to `std::async` [3](§30.6.8), leaving thread creation and management to the library. The result (if any) of the concurrent computation can be collected later by using the `std::future` construct. With this approach a program relying on a shared queue (which will be implemented in the Sections 2.2.4 and 2.2.5) can be implemented as follows:

```cpp
#include <future>
using namespace std;

auto main() -> int {
    auto q = queue{};
    // async returns a future which will contain the result
    auto f1 = async(launch::async, [&q](){ q.push(create_object()); });
    auto f2 = async(launch::async, [&q](){ return q.pop(); });

    // f1 does not return anything
    f1.get();

    // f2 returns an object
    auto o = f2.get();

    return 0;
}
```

std::async Note the explicit specification of `std::launch::async`. By default, one passes a function and an arbitrary number of arguments to `std::async`. This is equivalent to the following call:

```
using namespace std;
auto f = async(launch::async | launch::deferred, func, params...);
```

- `std::launch::async` explicitly tells the library to run the specified function in a separate thread. If `std::future::get` or `std::future::wait` are called before the task is completed, the calling thread is blocked until the result becomes ready.
- `std::launch::deferred` is the opposite: The function is to be executed on the first call to either `std::future::get` or `std::future::wait` and in the same thread as its caller.
- `std::launch::async | std::launch::deferred` shows implementation-defined behaviour, i.e. may either spawn a new thread or execute in the same thread as the caller.

In either case the computation's result is stored in the associated `std::future` and can be accessed by a call to `std::future::get`.

Lambda functions The second parameter to `std::async` in the example is called a lambda function. This language extension [3](§5.1.2) provides an easy way to create simple functions in-place. It has the following syntax:

`[capture clause](parameters){ body }`

The *capture clause* tells the compiler how to make names outside of the lambda's body visible on the inside. An empty clause (`[]`) prevents any access to names outside of the lambda. A reference clause (`[&]`) captures all names by reference, a copy clause (`[=]`) copies them. These operations are also applicable to single names: `[&foo]` captures only `foo` by reference, `[=bar]` copies only `bar`. The latter is equivalent to `[bar]`. Additionally, these operations are combinable: `[&foo, bar]` captures `foo` and copies `bar`.

The parameters and the body work mostly like those of a normal function, except that the parameters can be generic (declared as `auto`) as well since C++14 [4](§5.1.2).

2.2.3 Low-level Thread Creation

A manual approach to thread management is offered by `std::thread` [3](§30.3.1). It works in a very similar way to the Pthread library:

```cpp
#include <thread>

auto main() -> int {
    auto q = queue{};
    auto obj = object{};
    auto p = std::thread([&q](){ q.push(create_object()); });
    auto c = std::thread([&q, &obj](){ obj = q.pop(); });

    p.join();
    c.join();

    return 0;
}
```

In contrast to `std::async` there is no direct way to obtain the result of the computation; instead it is the programmer's responsibility to obtain the result in a way of his choosing, e.g. by passing an empty object by reference or by handcrafting a solution around `std::future` and its companion `std::promise`.

2.2.4 High-level Synchronisation

Whenever multiple threads are accessing the same data synchronisation becomes an important issue. For this purpose the C++11 standard includes synchronisation constructs, namely `std::mutex`, its derivates and locking mechanisms [3](§30.4) as well as `std::condition_variable` [3](§30.5). The latter offers a more fine-grained control over blocking or unblocking threads and is always bound to a `std::mutex`.

A shared queue utilizing `std::mutex` and `std::condition_variable` could be implemented as follows:

```
#include <condition_variable>
#include <mutex>
#include <queue>

template <class T>
class queue {
    public:
        auto push(T t) -> void {
            auto lock = std::unique_lock<std::mutex>{m_};

            queue_.push(std::move(t));
            cv_.notify_one();
        }

        auto pop() -> T {
            auto lock = std::unique_lock<std::mutex>{m_};
            while(queue_.empty())
                cv_.wait(lock);

            auto ret = std::move(queue_.front());
            queue_.pop();

            return ret;
        }

    private:
        std::condition_variable cv_;
        std::mutex m_;
        std::queue<T> queue_;
};
```

Once an object is pushed to the `queue`, the `std::mutex` is locked. Instead of calling `std::mutex`'s member functions `lock` and `unlock` directly, RAII is utilized by constructing a `std::unique_lock` object. The object is then pushed to the internal `std::queue`. Afterwards, another thread that is waiting to access the (previously empty) `queue` is notified.

Popping an object works similarly: first, the lock is obtained. Then, if the internal `std::queue` happens to be empty, the lock is released temporarily until another thread calls `notify_one`, a member function of `std::condition_variable`. Afterwards the first object in the queue is obtained and returned to the caller.

2.2.5 Low-level Synchronisation

Besides the high-level constructs presented in Section 2.2.4, C++11 also introduced atomic operations to the standard library. These offer another way to synchronise threads and leave more control (and responsibility) to the programmer. `std::atomic_flag` is especially useful for synchronisation constructs and can replace `std::mutex` in the shared queue implemented above:

```
#include <atomic>
#include <thread>
#include <queue>

template <class T>
class queue {
    public:
        auto push(T t) -> void {
            while(lock_.test_and_set())
                std::this_thread::yield();

            queue_.push(std::move(t));
            lock_.clear();
        }

        auto pop() -> T {
            while(queue_.empty())
                std::this_thread::yield();

            while(lock_.test_and_set())
                std::this_thread::yield();

            auto ret = std::move(queue_.front());
            queue_.pop();

            lock_.clear();
            return ret;
        }

        private:
            std::atomic_flag lock_ = ATOMIC_FLAG_INIT;
            std::queue<T> queue_;
};
```

The disadvantage of this approach is that it is not intuitive. `std::atomic_flag::test_and_set` atomically sets the flag to `true` and returns the previous value. In this context `true` means “locked”. The `while` loop will only break if another thread has cleared the flag, i.e. set it to `false`. In this case the flag is set to `true` by the current thread (thus blocking other threads), the current thread performs its work and then clears the flag (thus unblocking other threads).

From an outside perspective both implemented queues behave the same: There can be only one thread accessing the data, all other threads have to wait for it to complete its work.

3 Support in Score-P and Vampir

The following sections present the level of support for modern C++ by Score-P and Vampir. Most features are already supported by both tools; the focus therefore lies on missing capabilities. This is achieved by implementing several small example programs that make use of these features. The source code of these examples can be found in Appendix A. The code is compiled with GCC 6.2.1, using the compiler plugin from Score-P 3.0. The results are examined using Vampir 9.0.

3.1 Concurrency

In this section the various concurrency constructs introduced in C++11 is discussed.

3.1.1 `std::async`

As mentioned in Section 2.2.2 `std::async` is a high-level construct to asynchronously run tasks. The underlying management of threads is not exposed to the library user; instead, it is guaranteed that the result is returned upon a call to `std::future::get` (`std::async` returns a `std::future`). This may lead to blocking on the caller's side if the asynchronous task has not been completed yet. This potential bottleneck makes any application using `std::async` a natural target for performance analysis, even more so as its default behaviour with regard to thread spawning is implementation specific.

There are three test cases in this document: One for the default behaviour (Appendix A.1.1), another for explicit asynchronous evaluation (Appendix A.1.2) and a third for explicit lazy evaluation (Appendix A.1.3).

The first two test cases fail with the following error message (when compiled with GCC 6.2.1):

```
$ SCOREP_ENABLE_TRACING=1 ./future_default
[Score-P] src/measurement/thread/create_wait/scorep_thread_create_wait
          _pthread.c:84: Fatal: Bug 'tpd == 0': Invalid Pthread thread
          specific data object. Please ensure that all pthread_create
          calls are instrumented.
[Score-P] Please report this to support@score-p.org. Thank you.
[Score-P] Try also to preserve any generated core dumps.
[Score-P] src/measurement/thread/create_wait/scorep_thread_create_wait
          _pthread.c:84: Fatal: Bug 'tpd == 0': Invalid Pthread thread
          specific data object. Please ensure that all pthread_create
          calls are instrumented.[1]    8405 abort (core dumped)
```

Unfortunately, imitating `std::async`'s behaviour with the Pthread library for the sake of this document is impossible as the thread management behind the call is implementation-defined: the library may spawn operating system specific threads internally, rely on a thread pool or do something entirely different.

The lazy evaluation test case executes successfully; however, because the code in question is not executed in a separate thread the inspection of Score-P's trace file shows nothing noteworthy.

3.1.2 `std::thread`

When compared to `std::async`, `std::thread` follows a more “low level” approach as thread execution is now entirely in the hands of the user. Naturally C++11 threads are profiling targets, too.

There are two test cases for `std::thread`: the first spawns a thread and joins it afterwards (Appendix A.1.4), the second spawns a thread and detaches it, making it unjoinable (Appendix A.1.5).

Both test cases fail to execute and produce the same error message already known from the previous section. Because `std::thread` simply wraps a Pthread (at least in libstdc++) its behaviour can be emulated. Figures 1 and 2 are created by profiling such an emulation.

Figure 1 shows the creation of a thread by the master thread (the first blue block). Afterwards the master thread tries to join the new thread (the second blue block). This call blocks the master thread as indicated by the black lines connected to the thread execution (lower green block). The master thread can only resume once the spawned thread has completed its work, indicated by the second black line.

Figure 2 show the creation of a thread by the master thread (blue block). This thread is then detached, meaning it is left in an unjoinable state. The master thread continues its own work (upper green block) while the detached thread runs in parallel (lower green block).

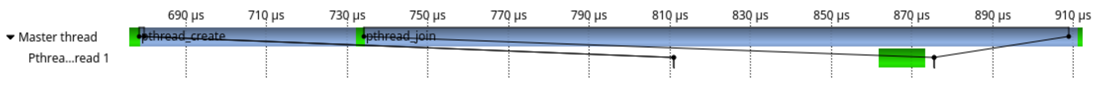

Figure 1: Emulated master timeline for a joined `std::thread`

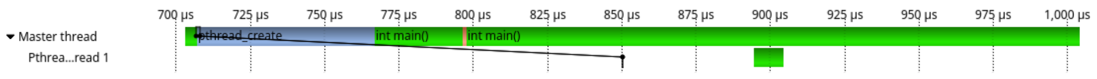

Figure 2: Emulated master timeline for a detached `std::thread`

3.1.3 Summary

At the time of writing, Score-P’s support for C++11 multithreading is nonexistent as the error messages from the sections above show. The likely cause for these errors lies in the internals of the GNU C++ compiler’s standard library because the threading support library is compiled into the shared library object. Score-P is unable to instrument this already compiled code and crashes upon execution once it encounters an unknown thread.

The solution for this problem would be a more graceful handling of “unknown threads”: Instead of crashing, Score-P should simply register them on the fly. As of September 2016 there is a new Score-P development branch adressing this issue.

The visualisation in Vampir should offer the same look and feel as the visualisation of Pthreads. That means:

- Vampir should draw a line between the call to `std::future::get` or `std::future::wait` and the actual end of the related function and from there to the end of the call. Like a Pthread thread the asynchronous task should be visible as a separate block on the master timeline. If the task was launched with the `std::launch::deferred` parameter no visualisation is needed as it shared the same thread of execution with its caller.

- When using `std::thread` the Pthread interface can be widely reused. It should not show the underlying native threads but the calls to the `std::thread` interface, e.g. `std::thread::join` instead of `pthread_join`. The visualisation of thread interdependencies should be identical, i.e. black lines from the call to the thread and back to the end of the call.

3.2 Synchronisation

With the introduction of the thread support library synchronisation primitives were added to the standard library as well. Because such primitives are able to block thread execution, profiling their behaviour is especially interesting in a HPC performance analysis context. In this section the mutual exclusion constructs (`std::mutex`) and condition variables (`std::condition_variable`) are evaluated. Due to the problems shown in Section 3.1 the spawned threads in the testcases are emulated by using the Pthread library.

3.2.1 `std::mutex`

`std::mutex` and its more specialised siblings `std::timed_mutex`, `std::recursive_mutex` and `std::recursive_timed_mutex` provide simple mechanisms for mutual exclusion between threads, namely locking and unlocking. However, directly accessing these mechanisms is not exception-safe which is why they are usually managed by `std::unique_lock` or `std::lock_guard`. Relying on RAII these primitives will automatically unlock the managed `std::mutex` once they leave their scope.

The test case (see Appendix A.2.1) utilizes them for synchronisation between two threads instead of locking the `std::mutex` manually. Upon execution the test case itself works as expected: threads are spawned and `std::mutex` blocks one of them while being locked by the other.

Score-P on the other hand is not able to fully parse the instruction flow correctly: instead of instrumenting the calls to the standard library facilities it detects and instruments the underlying Pthread mechanisms utilised by the library implementation, shown by Figures 3 and 4. This can be explained by the compiler's inlining of the STL functions which prevents Score-P from instrumenting the actual function call. The black lines in Figure 3 connecting the threads are known from Section 3.1.2 and indicate thread joining. The red block seen in one thread symbolises a blocking call to the mutex's locking routine.

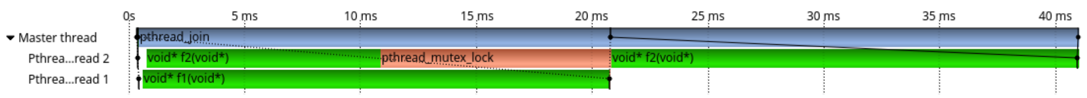

Figure 3: Master timeline for thread synchronisation with `std::mutex`

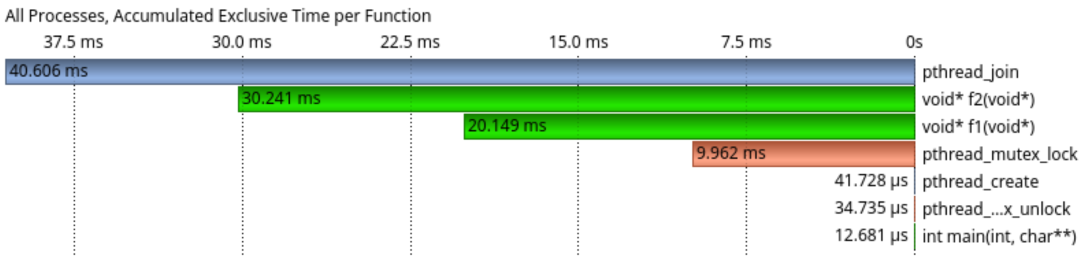

Figure 4: Function summary for thread synchronisation with `std::mutex`

3.2.2 `std::condition_variable`

`std::condition_variable` is the other synchronisation primitive added to the standard library with C++11, allowing threads to communicate with each other. A `std::condition_variable` is always associated with a `std::mutex` but offers more fine-grained control via its notification mechanisms – a thread can either wake up one other waiting thread or all of them.

The test case (see Appendix A.2.2) spawns four threads, of which three are waiting for the fourth to signal them. Once they receive the notification they execute their instructions in parallel.

Like the simpler `std::mutex` test case (see Section 3.2.1) this program executes successfully and suffers from the same problems with inlining (see Figures 5 and 6). Again, the red blocks indicate mutex locking or unlocking; the lines connecting the individual threads with the master thread represent thread joining operations.

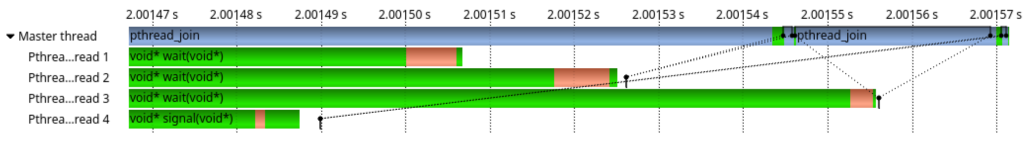

Figure 5: Master timeline for thread synchronisation with `std::condition_variable`

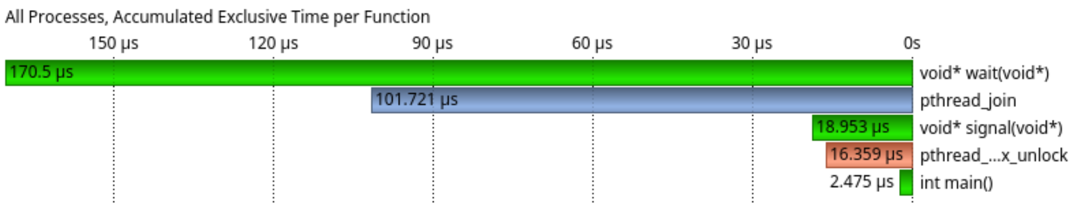

Figure 6: Function summary for thread synchronisation with `std::condition_variable`

3.2.3 Summary

While Score-P and Vampir generally support the instrumentation of mutex operations they fail to do this correctly for the STL because of inlining. In order to change this, Score-P would need a mechanism to instrument the STL's interface without automatically instrumenting all the inlined implementation details behind the interface.

The visual representation by Vampir could be adapted to work with the STL interface. Drawing connection lines between related mutex locking and unlocking operations would improve the user experience even further; these are not needed if threads are not actually waiting on each other despite using the same mutex (i.e. thread 1 unlocks the mutex before thread 2 tries to lock it – thread 2 is not blocked in this case and a connecting line does not make sense).

3.3 File I/O

While file I/O is not a new concept and has not changed with C++11 its performance penalties are still an important issue to consider in an HPC context. This section represents the evaluation of file I/O using C++ file streams as well as the C API.

3.3.1 C++ Filestreams

The first test case (see Appendix A.3.1) generates a `std::vector` with random integer values and writes them to a file. The file itself is created and modified by a `std::ofstream` opened in binary mode. It is then closed and opened again by a `std::ifstream` which then proceeds to read its contents into another `std::vector`.

At the time of writing all file operations using C++ streams are ignored by Score-P and Vampir.

3.3.2 C-style I/O

The result of Section 3.3.1 may lead to the conclusion that the file operations are inlined by the compiler and thus not visible to Score-P. The second test case implements the functionality from Section 3.3.1 using the API inherited from the C standard library (see Appendix A.3.2).

Again, Score-P and Vampir ignore all file operations. This shows that the issue is not related to inlining but to Score-P not instrumenting standard library and/or system calls.

3.3.3 Summary

As of September 2016 Score-P does not support the instrumentation of file I/O. There is, however, a development branch targeting these issues for the C API, representing file operations as triangles on the master timeline. From a usability perspective it might be more intuitive to represent them as a range rather than points (similar to Pthread library calls).

The visualisation of C++ file I/O should look similar to the proposed C API visualisation; a call to `std::fstream::write` is usually interchangeable with a call to `std::fwrite` since both operate on the level of individual bytes.

3.4 Lambdas

Lambda functions are especially useful when used in conjunction with the functions found in the header file `<algorithm>`. This naturally makes them a profiling target. Usually they are inlined because of their small code size which renders them invisible for Score-P.

In cases where inlining does not occur Score-P faces two minor naming issues. The first issue can be reproduced by the following code:

```
auto my_lambda = [](int i) { //... };
```

If this code does not get inlined for whatever reason[2] Score-P will show the following name:

```
main::{lambda(int)#2}::operator()(int) const
```

This naming is very unintuitive and does not help the user in quickly identifying hotspots. The situation becomes worse when using an `auto` parameter, a feature introduced in C++14:

```
auto my_lambda = [](auto i) { // ... };

_ZZ4mainENKUlTE_clIiEEDaS_ // called with "int" parameter
_ZZ4mainENKUlTE_clIdEEDaS_ // called with "double" parameter
```

Both issues are not Score-P's fault but originate in GCC's naming and mangling scheme. To work around this, Score-P would need to implement a custom naming mechanism which works around the one imposed by GCC. The easier alternative for the second issue is to wait for a GCC bugfix which improves generic lambda name mangling.

[2]Unfortunately, this behaviour was not reliably reproducible which is why there is no test case in the appendix. The lambda's symbols are present in the executable but they are still not visible to Score-P.

4 Outlook On C++17

This section provides a brief overview for the planned changes of the upcoming C++ standard [5]. This revision is likely to be released in 2017.

4.1 The Standardisation Process

When C++11 was released in 2011 thirteen years had passed since the last major C++ standard (C++98) and eight since the last minor standard (C++03). During this time technology has evolved and with it the needed programming models, for example the widespread adoption of multicore CPUs and thus concurrent programming. However, the C++ programming language and its standard library did not reflect this development, forcing developers to rely on third-party libraries such as Boost or Pthreads.

In order to prevent this issue in the future the C++ standard committee was reorganised into several subgroups in 2012, adopting a decentral work process. These subgroups are officially called “study groups” and each of them works on evolving a part of the language or the library. As of September 2016 there are 14 study groups [7], focusing on topics such as low latency, issues with the currently available concurrency features, databases and more.

The overall goal is to publish new standards much more frequently than before. The focus on faster and therefore “slimmer” standardisation also has the effect that compiler and standard library developers are able to fully implement the standard much faster [17].

4.2 `if`-Statement with Initializer

The next C++ standard will see a significant change to the core language: `if`-statements can initialise variables directly. The current way of initialising looks like this:

```cpp
std::map<int, std::string> m;

auto it = m.find(10);
if(it != m.end()) return it->size();
```

This approach has the disadvantage that `it` is now visible beyond the `if`-statement and can not be changed later to e.g. a `std::vector::iterator`. The upcoming standard enables the developer to change his code to the following [5](§6.4):

```cpp
std::map<int, std::string> m;

if(auto it = m.find(10); it != m.end()) return it->size();
```

Additionally, `it` is now only visible inside the scope of the `if`-statement. This means he can reuse the name later for another purpose.

4.3 Structured Bindings

Another important change to the core language are *structured bindings* [5](§8.5). These enable the developer to initialise multiple variables at once from a struct:

```
struct s {
    int x;
    volatile double y;
};

auto f() -> s { /* ... */ }
const auto [x, y] = f();

foo(x); // x is of type const int
bar(y); // y is of type const volatile double
```

In combination with the changes presented in Section 4.2 this leads to clean and elegant code:

```
if(auto [x, y, z] = foo(); x.valid())
    bar(y, z);
```

4.4 Library Extensions

Apart from changes to the core language the upcoming standard will see additions to the standard library as well. Some of those changes are:

- the switch to the C11 standard as foundation for C++17 [5](§1.2)
- additional mathematical functions (polynomials, beta function, elliptic integrals, etc.) [5](§26.9.5)
- functions for filesystem interaction [5](§27.10)
- new algorithms in the header file `<algorithm>` (`sample`, `reduce`, etc.) [5](§25)
- parallelisable algorithms (in the header `<algorithm>`) [5](§25)

The last point raises the question how parallelisable algorithms such as `std::find` are internally implemented (which makes them suitable for specific use cases or not). At the time of writing most of the commonly available standard libraries do not yet support parallel algorithms; however, Microsoft provided an open-source reference implementation [11].

Appendix B shows a Microsoft-inspired implementation of a parallelised `std::find`. In this example the work is distributed amongst an implementation-defined number of threads. Each thread then processes its part of the work. Once it finds the value in question it checks the shared state for the existence of a better find. Should there be no better result the state is set to the current value; otherwise the value is discarded. In either case the thread stops its execution at this point. If there are no fitting values the thread continues its execution until it reaches the end of its range.

5 Conclusion

Modern C++ changed the way of efficient C++ programming tremendously. It leads to a higher abstraction level without a decrease in performance, makes memory management (and thus safety) as well as concurrency a lot easier and offers several other advantages to the HPC community.

The increasing adoption of these changes by developers puts a lot of pressure on performance analysis tools such as Score-P and Vampir. Most of the features offered by C++ and its standard library are already supported or not relevant in a performance analysis context. As this report shows some important features – especially in the field of concurrency – are not supported yet. This is an important issue that needs to be addressed quickly if Score-P and Vampir should not become irrelevant in the medium term. Solutions for some of these problems are outlined in this paper; other issues are actively worked on at the time of writing.

Perhaps the most interesting additions (from an HPC perspective) to the standard library will be the parallelisable algorithms from the upcoming standard. Correctly instrumenting and displaying these algorithms at the time of their release would be a great improvement to both Score-P and Vampir and a serious advantage over comparable applications.

The test cases from the appendix, along with their traces and several other example programs, can be found at GitHub: `https://github.com/j-stephan/hs16`.

6 Additional Literature

The inclined reader may refer to one or more of the works listed below to follow a more in-depth introduction to modern C++:

- Stanley B. Lippmann, Josee Lajoie, Barbara E. Moo: *C++ Primer*, Addison-Wesley, 2012. The intended audience of this book are beginners, both in programming and C++. It follows a high-level approach; for example it starts with containers before explaining dynamic memory allocation.
- Bjarne Stroustrup: *A Tour of C++*, Addison-Wesley, 2014. This book targets people who have programmed before (not necessarily in C++) and provides them with a broad introduction to the features found in modern C++.
- Bjarne Stroustrup: *The C++ Programming Language*, Addison-Wesley, 2013. With this work the creator of the C++ programming language published an exhaustive introduction to modern C++, covering the language and its standard library in great detail.
- Scott Meyers: *Effective Modern C++*, O'Reilly, 2014. Targeting an experienced audience, Scott Meyers provides a lot of optimization hints with this work.
- Anthony Williams: *C++ Concurrency in Action*, Manning, 2012. As the title says this book focuses on modern C++ concurrency mechanisms.

Additionally, the following websites are valuable sources of information:

- Bjarne Stroustrup: *C++11 FAQ*, `http://www.stroustrup.com/C++11FAQ.html`. This page explains the frequently questioned design decisions of the C++11 standard.
- Herb Sutter: *Guru of the Week*, `https://www.herbsutter.com/gotw/`. In this weekly weblog series Herb Sutter explains C++ features in great depth, usually by providing a puzzle and explaining the solution a week later.

References

[1] Holger Brunst and Matthias Weber. Custom hot spot analysis of HPC software with the Vampir performance tool suite. In *Tools for High Performance Computing 2012*, pages 95–114. Springer, 2013.

[2] International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC). International Standard ISO/IEC 9899:1999(E) – programming language C, 1999.

[3] International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC). International Standard ISO/IEC 14882:2011(E) – programming language C++, 2011.

[4] International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC). International Standard ISO/IEC 14882:2014(E) – programming language C++, 2014.

[5] International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC). Working draft, standard for programming language C++ (N4606), July 2016.

[6] The Standard C++ Foundation. C++11 overview: Is C++11 hard to learn? `https://isocpp.org/wiki/faq/cpp11`. Online; accessed 27 September 2016.

[7] The Standard C++ Foundation. The committee. `https://isocpp.org/std/the-committee`, 2016. Online; accessed 28 September 2016.

[8] The Open Group. General concepts. `http://pubs.opengroup.org/onlinepubs/9699919799/`, April 2013. Online; accessed 29 September 2016.

[9] Andreas Knüpfer, Christian Rössel, Dieter Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, et al. Score-P: A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Tools for High Performance Computing 2011*, pages 79–91. Springer, 2012.

[10] The Qt Company Ltd. QThread class. `https://doc.qt.io/qt-5/qthread.html`, June 2016. Online; accessed 29 September 2016.

[11] Microsoft. Parallel STL. `https://parallelstl.codeplex.com/`, April 2014. Online; accessed 28 September 2016.

[12] Microsoft. Multithreading with C and Win32. `https://msdn.microsoft.com/en-us/library/y6h8hye8.aspx`, July 2015. Online; accessed 29 September 2016.

[13] Bjarne Stroustrup. *A Tour of C++*. C++ In-Depth. Addison-Wesley, 2014.

[14] Bjarne Stroustrup and Herb Sutter. C++ core guidelines. `https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines`, August 2016. Online; accessed 27 September 2016.

[15] Herb Sutter. GotW #94 solution: AAA style (almost always auto). `https://herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/`, August 2013. Online; accessed 14 September 2016.

[16] Herb Sutter. Back to basics: Modern C++ style. `https://github.com/CppCon/CppCon2014/tree/master/Presentations/Back%20to%20the%20Basics!%20Essentials%20of%20Modern%20C%2B%2B%20Style`, September 2014. Online; accessed 12 June 2016.

[17] Herb Sutter. We have C++14! `https://isocpp.org/blog/2014/08/we-have-cpp14`, August 2014. Online; accessed 28 September 2016.

A Score-P / Vampir Test Cases

A.1 Concurrency

A.1.1 `std::async` – Default Behaviour

```cpp
#include <algorithm>
#include <cstddef>
#include <cstdint>
#include <future>
#include <iterator>
#include <numeric>
#include <vector>

constexpr auto vec_size = std::size_t{1000000};

auto task1() -> void
{
    // fill a vector with values (vec_size, 0] and sort it
    auto vec = std::vector<std::int32_t>{};
    vec.resize(vec_size);
    std::iota(std::rbegin(vec), std::rend(vec), 0);
    std::sort(std::begin(vec), std::end(vec));
}

auto task2() -> std::int32_t
{
    // fill a vector with random values and return the maximum
    auto vec = std::vector<std::int32_t>{};
    vec.resize(vec_size);
    std::generate(std::begin(vec), std::end(vec), std::rand);
    auto it = std::max_element(std::begin(vec), std::end(vec));
    return *it;
}

auto main() -> int
{
    auto f1 = std::async(task1);
    auto f2 = std::async(task2);

    f1.get(); // task1 has return type void
    auto res = f2.get();

    return 0;
}
```

A.1.2 `std::async` – Asynchronous Evaluation

```cpp
#include <algorithm>
#include <cstddef>
#include <cstdint>
#include <future>
#include <iterator>
#include <numeric>
#include <vector>

constexpr auto vec_size = std::size_t{1000000};

auto task1() -> void
{
    // fill a vector with values (vec_size, 0] and sort it
    auto vec = std::vector<std::int32_t>{};
    vec.resize(vec_size);
    std::iota(std::rbegin(vec), std::rend(vec), 0);
    std::sort(std::begin(vec), std::end(vec));
}

auto task2() -> std::int32_t
{
    // fill a vector with random values and return the maximum
    auto vec = std::vector<std::int32_t>{};
    vec.resize(vec_size);
    std::generate(std::begin(vec), std::end(vec), std::rand);
    auto it = std::max_element(std::begin(vec), std::end(vec));
    return *it;
}

auto main() -> int
{
    auto f1 = std::async(std::launch::async, task1);
    auto f2 = std::async(std::launch::async, task2);

    f1.get(); // task1 has return type void
    auto res = f2.get();

    return 0;
}
```

A.1.3 `std::async` – Lazy Evaluation

```cpp
#include <algorithm>
#include <cstddef>
#include <cstdint>
#include <future>
#include <iterator>
#include <numeric>
#include <vector>

constexpr auto vec_size = std::size_t{1000000};

auto task1() -> void
{
    // fill a vector with values (vec_size, 0] and sort it
    auto vec = std::vector<std::int32_t>{};
    vec.resize(vec_size);
    std::iota(std::rbegin(vec), std::rend(vec), 0);
    std::sort(std::begin(vec), std::end(vec));
}

auto task2() -> std::int32_t
{
    // fill a vector with random values and return the maximum
    auto vec = std::vector<std::int32_t>{};
    vec.resize(vec_size);
    std::generate(std::begin(vec), std::end(vec), std::rand);
    auto it = std::max_element(std::begin(vec), std::end(vec));
    return *it;
}

auto main() -> int
{
    auto f1 = std::async(std::launch::deferred, task1);
    auto f2 = std::async(std::launch::deferred, task2);

    f1.get(); // task1 has return type void
    auto res = f2.get();

    return 0;
}
```

A.1.4 `std::thread` – Spawn and Join

```cpp
#include <algorithm>
#include <cstddef>
#include <cstdint>
#include <iterator>
#include <numeric>
#include <thread>
#include <vector>

constexpr auto vec_size = std::size_t{1000000};

auto f() -> void
{
    auto vec = std::vector<std::int32_t>{vec_size};
    std::iota(std::rbegin(vec), std::rend(vec), 0);
    std::sort(std::begin(vec), std::end(vec));
}

auto main() -> int
{
    auto t = std::thread{f};
    t.join();

    return 0;
}
```

A.1.5 `std::thread` – Spawn and Detach

```cpp
#include <algorithm>
#include <chrono>
#include <cstddef>
#include <cstdint>
#include <iterator>
#include <numeric>
#include <thread>
#include <vector>

constexpr auto vec_size = std::size_t{1000000};

auto f() -> void
{
    auto vec = std::vector<std::int32_t>{vec_size};
    std::iota(std::rbegin(vec), std::rend(vec), 0);
    std::sort(std::begin(vec), std::end(vec));
}

auto main() -> int
{
    auto t = std::thread{f};
    t.detach();

    using namespace std::chrono_literals;
    std::this_thread::sleep_for(5s);

    return 0;
}
```

A.2 Synchronisation

A.2.1 std::mutex

```cpp
#include <iostream>
#include <mutex>

#include <pthread.h>
#include <unistd.h>

std::mutex m;

auto f1(void*) -> void*
{
    std::lock_guard<std::mutex> l(m);
    std::cout << "Mutex locked in f1" << std::endl;
    usleep(20 * 1000);
    return nullptr;
}

auto f2(void*) -> void*
{
    usleep(10 * 1000);
    std::lock_guard<std::mutex> l(m);
    std::cout << "Mutex locked in f2" << std::endl;
    usleep(20 * 1000);
    return nullptr;
}

auto main() -> int
{
    pthread_t threads[2];

    pthread_create(&threads[0], nullptr, f1, 0);
    pthread_create(&threads[1], nullptr, f2, 0);

    pthread_join(threads[0], nullptr);
    pthread_join(threads[1], nullptr);

    return 0;
}
```

A.2.2 `std::condition_variable`

```
#include <algorithm>
#include <array>
#include <condition_variable>
#include <iostream>
#include <iterator>
#include <mutex>

#include <pthread.h>
#include <unistd.h>

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

auto done = false;

auto wait(void*) -> void*
{
    std::unique_lock<std::mutex> l(m);
    std::cout << "Waiting...\n";
    cv.wait(l, []{ return done; });
    std::cout << = true"...finished waiting. done =\n";
    return nullptr;
}

auto signal(void*) -> void*
{
}

auto main() -> int
{
    auto threads = std::array<pthread_t, 4>{};
    std::for_each(std::begin(threads), std::begin(threads) + 3,
     ↪  [](auto& t){ pthread_create(&t, nullptr, wait, 0); });
    pthread_create(&threads[3], nullptr, signal, 0);

    for(auto&& t : threads)
        pthread_join(t, nullptr);

    return 0;
}
```

A.3 File I/O

A.3.1 `std::fstream`

```
#include <algorithm>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <string>
#include <vector>

auto main() -> int
{
    auto path = std::string{"test.bin"};
    auto output = std::vector<int>{};
    auto input = std::vector<int>{};
    output.resize(1000);
    input.resize(1000);

    std::generate(std::begin(output), std::end(output), std::rand);

    {
        std::ofstream out{path.c_str(), std::ios::binary};
        out.write(reinterpret_cast<const char*>(output.data()),
          ↪  output.size() * sizeof(int));
    }
    {
        std::ifstream in{path.c_str(), std::ios::binary};
        in.read(reinterpret_cast<char*>(input.data()), input.size() *
          ↪  sizeof(int));
    }

    if(!std::equal(std::begin(output), std::end(output),
     ↪  std::begin(input)))
        std::cerr << "Error: Input does not match output." <<
          ↪  std::endl;

    return 0;
}
```

A.3.2 C-style I/O

```cpp
#include <algorithm>
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <string>
#include <vector>

auto main() -> int
{
    auto path = std::string{"test.bin"};
    auto output = std::vector<int>{};
    auto input = std::vector<int>{};
    output.resize(1000);
    input.resize(1000);

    std::generate(std::begin(output), std::end(output), std::rand);

    auto out = std::fopen(path.c_str(), "wb");
    std::fwrite(output.data(), sizeof(int), output.size(), out);
    std::fclose(out);

    auto in = std::fopen(path.c_str(), "rb");
    std::fread(input.data(), sizeof(int), input.size(), in);
    std::fclose(in);

    if(!std::equal(std::begin(output), std::end(output),
     ↪ std::begin(input)))
        std::cerr << "Error: Input does not match output." <<
         ↪ std::endl;

    return 0;
}
```

B Parallel `std::find`

```
using namespace std;

template <class It, class T>
It find_impl(execution::par, It first, It last, const T& value)
{
    // container size
    auto size = distance(first, last);

    // thread-shared state
    using diff_t = typename iterator_traits<It>::difference_type;
    atomic<diff_t> state(size);

    /* Omitted: thread spawning. count represents the thread-local
     * work size, begin the thread-local starting position. */
    auto dist = distance(first, begin);
    for(auto pos = 0; pos < count; ++pos, ++begin) {
        if(value == *begin) {
            auto cur = dist + pos;
            auto old = state.load(memory_order_relaxed);

            do {
                // do not replace if a closer match has been found
                if(old < cur)
                    break;
            } while(!state.compare_exchange_strong(old, cur,
             ↪  memory_order_relaxed));

            break;
        }
    }

    /* Omitted: thread joining. */
    auto pos = state.load(memory_order_relaxed);
    if(pos != size) {
        advance(first, pos);
        return first;
    }

    return last;
}
```