abstract

This paper proposes a minimal API that enables stackful context switching without the need for a scheduler. The API is suitable to act as building-block for high-level constructs such as stackful coroutines as well as cooperative multitasking (aka user-land/green threads that incorporate a scheduling facility).

This revision addresses concerns, questions and suggestions from the past meetings. The proposed API supersedes the former proposals N3985, P0099R1, P0534R3 and P0876R13.

Because of name clashes with coroutine from C++20, execution context from executor proposals and continuation used in the context of future::then(), the committee has indicated that fiber is preferable. However, given the foundational, low-level nature of this proposal, we choose fiber_context, leaving the term fiber for a higher-level facility built on top of this one.

Informally within this proposal, the term fiber is used to denote the lightweight thread of execution launched and represented by the first-class object fiber_context.
Revision History
This document supersedes P0876R13.

Changes since P0876R13

- At LEWG’s request, retracted changes to `uncaught_exceptions()` and `current_exception()`, instead clarifying that results may reflect exceptions on other fibers running on the current thread.
- Updated against draft standard N4958.
- Deleted “User-Mode” from new section title “Cooperative Threads” and removed the explanatory paragraph.
- Removed `explicit` from the explicit-stack constructor.
- Added `system_error: resource_unavailable_try_again` to the `Throws:` clause of the implicit-stack constructor.
- Changed `bad_alloc` to `system_error: resource_unavailable_try_again` in the `Throws:` clause of the explicit-stack constructor.
- Stated that the move constructor and move assignment operator empty the moved-from `fiber_context`.
- Removed the `empty()` precondition from assignment operator; instead added the same `(! empty())` effect as for the destructor.
- Removed `resume_with()` references to “execution context.” Existing section 7.6.1.3 Function call `[expr.call]` makes no mention of saving or restoring state.
- Removed bullets in `resume_with()`

  `Returns:` and

  `Throws:` clauses regarding `resume()`, since they can be inferred from `resume_with()` and the trivial-lambda equivalence described for `resume()`.
- Removed the

  `Remarks:` about concurrent calls from multiple threads from `can_resume()`, leaving in place the editorial note about the intentional absence of `const`.
- Changed exposition-only `state` member from unspecified-type to `void*`.
- Sanitized stable names.
- Moved feature-test macro to appropriate section.
- Cleaned up the header-file synopsis.
- Grouped class members with forward references.
- Added `std::swap()` specialization.
- Added obtrusive paragraph numbers.
- Streamlined single-item dash lists.
- Changed `Ensures` to `Postconditions`.
- Changed template parameters from `typename` to `class`.
- Tweaked constructor

  `Preconditions:`/
Mandates:

- Clarified that `entry_copy`, `stack_copy` and `deleter_copy` are not intended to be data members of `fiber_context`.
- Streamlined initialization of these exposition objects.
- “Instantiates a `fiber_context`” => “Initializes state”
- `empty()` returns `true` => `empty()` is `true`, et al.
- Removed explicit-stack constructor

Preconditions: for stack size and alignment, since

Throws: explicitly specifies exceptions for violations.

Effects: of move constructor.

Extracted “Let” statements from

Effects: to preceding paragraphs.

Changes since P0876R12

- Proposed that `uncaught_exceptions()` and `current_exception()` be specific to the current thread of execution.
- Specified that constructors `decay-copy` the entry-function.
- Changed `span<byte, N>` constructor param to simply `span<byte>`; also accepted deleter function, which it must `decay-copy`.
- Specified constructor exceptions.
- Specified that destroying a non-empty `fiber_context` calls `terminate()`.
- Clarified that when `resume_with()` is called, `empty()` becomes true immediately.
- Introduced exposition-only `fiber_context::state` member to streamline wording.
- Removed `concurrency_v2` namespace.
- Changed “Equivalent to” to “As-if”.
- Clarified Preconditions vs. Mandates.

Changes since P0876R11

- Removed `get_stop_source()`, `get_stop_token()`, `request_stop()` and exposition-only `ssource` members.
- Added a `fiber_context` constructor accepting a caller-provided uninitialized memory area for the new fiber’s function call stack.

Bundling a `stop_source` into `fiber_context` presented implementability concerns. Although each fiber (specifically, its function call stack) is itself a persistent entity, the `fiber_context` representing that fiber is not: a new `fiber_context` object is synthesized on every suspension. This presents a problem: how does the code that suspends a fiber find its associated `stop_source` shared state?

A consumer wishing to pass a `std::stop_token` to a new fiber can itself instantiate `std::stop_source`, obtain from it a `stop_token` and bind that `stop_token` in a lambda passed to the `fiber_context` constructor. Accordingly, the `fiber_context` API need not explicitly support that.
Changes since P0876R10

- Removed `cancel()` method and the cancellation-function constructor argument. Replaced with the `std::jthread` stop token handling API: `get_stop_source()`, `get_stop_token()` and `request_stop()`. This simplifies examples by eliminating `launch()` and `assert_on_cancel`.
- Added a section exploring the relationship of `fiber_context` to the larger C++ ecosystem.
- Reordered some sections to make the paper more accessible for new readers.

Changes since P0876R9

- Removed `resume_from_any_thread()`, `resume_from_any_thread_with()`, `cancel_from_any_thread()` and `can_resume_from_this_thread()`, along with stated support for resuming a suspended fiber on some thread other than the one on which it was launched.

In Belfast, EWG came down strongly against cross-thread fiber resumption. The most emphatic objection was that for a function referencing TLS, multiple compilers cache TLS pointers on the function’s stack frame. Resuming a fiber containing that stack frame on some other thread would cause problems. In the best case, the resumed function would merely reference TLS belonging to the wrong thread – but at some point the original thread will terminate, its TLS will be destroyed, and the cached pointers will be left dangling.

With `fiber_context`, any opaque function call might possibly suspend – but invalidating cached TLS pointers across every opaque function call is deemed unacceptable overhead.

Changes since P0876R8

- Reinstated cancellation function constructor argument.
- Added `cancel()` and `cancel_from_any_thread()` member functions.
- Re-removed `std::unwind_fiber()`.

SG1 directed P0876R9 to conform to the Cologne 2019 recommendations, with any other changes proposed in a separate paper.

Changes since D0876R7

- Cancellation function removed from `fiber_context` constructor.
- `std::unwind_fiber()` re-added, with implementation-defined behaviour.
- Added elaboration of `filament` example to bind cancellation function.

P0876R8 diverged from the recommendations of the second SG1 round in Cologne 2019. It did not introduce `cancel()` or `cancel_from_any_thread()` member functions. In fact it removed the cancellation-function constructor argument. `fiber_context` is intended as the lowest-level stackful context-switching API. Binding a cancellation-function on the fiber stack is a flourish rather than a necessity. It adds overhead in both space (on the fiber stack) and time (to traverse the stack to retrieve the cancellation-function). For this API, it should suffice to pass the desired cancellation-function to `resume_with()`. If it is important to associate a cancellation-function with a particular fiber earlier in the lifespan of the fiber, a struct serves.

A more compelling reason to avoid constructing an explicit fiber with a cancellation-function is that no implicit fiber has any such cancellation-function – and the consuming application cannot tell, a priori, whether a given `fiber_context` instance represents an explicit or an implicit fiber. If `*this` represents an implicit fiber, what should the proposed `cancel()` member function do?

Passing a specific cancellation-function to `resume_with()` avoids that problem.

P0876R8 follows SG1 recommendation in making it Undefined Behaviour to destroy (or assign to) a non-empty `fiber_context` instance.

`std::unwind_fiber()` was reintroduced with implementation-defined behaviour to allow fiber cleanup leveraging implementation internals. Its use was entirely optional (and auditable).
Changes since P0876R6

- Implicit stack unwinding (by non-C++ exception) removed.
- `std::unwind_fiber()` removed.
- Cancellation function added to `fiber_context` constructor.

In Cologne 2019, SG1 took the position that:

- The `fiber_context` facility is not the only C++ feature that requires “special” unwinding (special function exit path).
- Such functionality should be decoupled from `fiber_context`. It requires its own proposal that follows its own course through WG21 process.
- Depending on this (yet to be written) proposal would unduly delay the `fiber_context` facility.
- For now, the `fiber_context` facility should adopt a “less is more” approach, removing promises about implicit unwinding, placing the burden on the consumer of the facility instead.
- This leaves the way open for `fiber_context` to integrate with a new, improved unwind facility when such becomes available.

The idea of making `fiber_context`'s constructor accept a cancellation function was suggested to permit consumer opt-in to P0876R5 functionality where permissible, or convey to the fiber in question by any suitable means the need to clean up and terminate.

Requiring the cancellation function is partly because it remains unclear what the default should be. This could be one of the questions to be answered by a TS. Moreover, the absence of a default permits specifying later that the default engages the new, improved unwind facility.

Changes since P0876R5

- `std::unwind_exception` removed.
- `fiber_context::can_resume_from_any_thread()` renamed to `can_resume_from_this_thread()`.
- `fiber_context::valid()` renamed to `empty()` with inverted sense.
- Material has been added concerning the top-level wrapper logic governing each fiber.

`std::unwind_exception` was removed in response to deep discussions in Kona 2019 of the surprisingly numerous problems surfaced by using an ordinary C++ exception for that purpose.

Problems resolved by discarding `std::unwind_exception`:

- When unwinding a fiber stack, it is essential to know the subsequent fiber to resume. `std::unwind_exception` therefore bound a `fiber_context`. `fiber_context` is move-only. But C++ exceptions must be copyable.
- It was possible to catch and discard `std::unwind_exception`, with problematic consequences for its bound `fiber_context`.
- Similarly, it was possible to catch `std::unwind_exception` but not rethrow it.
- If we attempted to address the problem above by introducing a `std::unwind_exception` operation to extract the bound `fiber_context`, it became possible to rethrow the exception with an empty (moved-from) `fiber_context` instance.
- Throwing a C++ exception during C++ exception unwinding terminates the program. It was possible for an exception implementation based on `thread_local` to become confused by exceptions on different fibers on the same thread.
- It was possible to capture `std::unwind_exception` with `std::exception_ptr` and migrate it to a different fiber – or a different thread.
higher-level libraries fiber_context as building block for higher-level frameworks enumerates a number of higher-level abstraction libraries built upon the Boost.Context implementation of the API proposed in this paper. This is not an exhaustive list, but it suffices to illustrate that there is widespread interest in this functionality.

The most significant point about this proposal is that, given fiber_context, all those libraries can be written in standard C++. They need not themselves be integrated into the Standard.

Because it creates and switches between different function call stacks, though, the fiber_context facility cannot be written in portable C++. There is real value to integrating this library into the Standard.

Boost.Context is maintained by one individual to support the specific set of processors and operating systems to which he has access. The fiber_context facility will ensure support in every implementation of the C++ runtime, extending into the future.

Given the lively ecosystem of open-source libraries, it’s possible that standardizing fiber_context could suffice. It is not essential that WG21 must standardize additional higher-level libraries before the facility would become useful. The uptake of Boost.Context illustrates that the community can make good use of fiber_context.

However, the evolution of this proposal and the WG21 discussions thereof have surfaced a number of interesting adjacencies.

cancellation Given C++ support for concurrency, in various forms, within a program, cancellation of an asynchronous task remains a topic of widespread interest. It has been much discussed, e.g. in P1677R2, P1820R0 and P2175R0.

Previous revisions of this paper have proposed canceling a suspended fiber by injecting an exception, e.g. using fiber_context::resume_with(). A comparable approach was rejected for std::jthread although it’s worth noting that cooperative fibers differ in a very significant respect: every fiber suspends at a well-defined point, namely a call to resume_with().

Evolution of the exception mechanism itself may affect the viability of using exceptions for cancellation.

This paper simply notes that an invoker can use lambda binding to pass (e.g.) a std::stop_token from the Standard, to a fiber at launch time.

modules and optimizations Before modules, the only information the compiler could know about a function in an external translation unit was what a human coder stated in the relevant header file. But since the information in a module is prepared by the compiler itself, a subsequent compile of a translation unit that imports that module can know as much about each module function as it would if the function’s source code was found within the current translation unit.

This permits the compiler to infer and propagate attributes. If a function neither contains a throw statement nor calls other functions, the compiler can conclude that it doesn’t throw. It can encode this information in the module produced for that translation unit, so that subsequent compiles can make use of the knowledge. If another function contains no throw statement and calls only functions known not to throw, it too can be implicitly marked nothrow.

Similarly, when compiling a function that can never return, the compiler can so indicate in the output module. Any caller whose code path leads unconditionally to any such function can also be known never to return.

In much the same way, the module describing the library’s fiber_context::resume_with() method can mark it as can-suspend. Then any caller of resume_with() will also be marked can-suspend, and so forth. The compiler can use this to improve its optimization tactics around any call to a can-suspend function.

(The can-suspend characteristic of a co_await coroutine function is just as pervasive, but in that case the coder must manually propagate it.)

synchronization primitives The Standard provides an assortment of primitives for synchronizing work between threads, e.g. sections 33.6, 33.7, 33.8, 33.9, 33.10. An essential behaviour of many such synchronization primitives is to pause, or suspend, execution of the current thread until some external condition is satisfied.

Such suspension is very different from fiber suspension as proposed in this paper. This proposal neither requires nor implies a scheduler. A fiber suspends by explicitly designating the next fiber to resume, either by passing its fiber_context to resume_with() or by returning that fiber_context from its entry-function.

*Although exception-based cancellation is not implicitly supported, a consumer of fiber_context may still explicitly pass to resume_with() an invocable that raises an exception in the suspended fiber.
C++ threads, in contrast, assume a thread scheduler, usually provided by the operating system. Suspending a thread means passing control to the scheduler, which reallocates CPU resources to other pending threads. At some future time, the scheduler is responsible for directing some CPU core to resume the suspended thread.

Fiber suspension as implemented by fiber_context is independent of thread suspension. Suspending the running fiber simply means directing the thread to run a different fiber; the thread continues running. Conversely, suspending the host thread (e.g. by invoking a synchronization primitive) means that no fiber is running on that thread.

A higher-level fiber-based library that emulates the std::thread API, such as Boost.Fiber, necessarily implements a fiber scheduler, permitting implicit fiber suspension. Standardizing such a library would raise the interesting question of how to present fiber-aware synchronization primitives.

A straightforward approach is to present a suite of fiber-aware synchronization primitives distinct from, but analogous to, the thread-based synchronization primitives. A program running multiple fibers within a thread would use fiber-aware synchronization primitives rather than thread-based synchronization primitives. Evaluating a thread-based synchronization primitive would suspend the entire thread, as usual, halting all fibers within that thread.

It is tempting to contemplate modifying the semantics of the present suite of synchronization primitives to make them fiber-aware. Naturally this is a matter of some concern.

For purposes of this fiber_context proposal, though, it is entirely moot.

**Execution Agent Local Storage** A similar question arises concerning variable storage duration. Should the Standard introduce a fiber-specific storage duration, e.g. fiber_local, analogous to thread_local? (section 6.7.5.3 Thread storage duration)

The Standard defines the general term execution agent (section 33.2.5.1) to allow for multiple kinds of parallelism. It seems reasonable to assume that over time, new types of execution agents will be defined. Will we want the Standard to present a new xyz_local storage duration for each new “xyz” execution agent type?

P0772R1 notes that library code should not have to care what kind of execution agent is running it. Already it’s important to ensure that library code avoids static variables because any such variable prohibits calling that library from more than one thread. P0772R1 suggests a generalized variable storage duration dynamically local to the innermost current execution agent.

(The same consideration about library code impacts the above question about presenting fiber-aware synchronization primitives.)

It’s true that if:

- a particular function relies on a thread_local variable
- the function calls a function that resumes a different fiber
- the other fiber makes a different call to that same function, or to another function that modifies the same thread_local variable

then on resumption of the original fiber, the function will observe the new value for the thread_local variable.

This is analogous to use of a static variable by multiple threads in the same program – though not as bad, since it doesn’t produce race-related Undefined Behaviour on top of correctness problems.

std::thread was introduced despite this problem because it’s useful.

Multiple C++ implementations cache a pointer to thread-local storage in the stack frame of a function referencing TLS. If a suspended fiber were resumed by a thread other than the one on which it previously ran, such cached TLS pointers would point to TLS for the wrong thread. This is why such cross-thread resumption is forbidden.

(This is the only optimization that has yet been surfaced by implementers as a potentially problematic interaction with fibers.)

**tooling** One particularly valuable consequence of adding fiber_context to the Standard will be to add fiber awareness to debuggers, performance analyzers and other tools that inspect a running C++ program.

Such tools need only be aware of fiber_context. They would not need to be further adapted to support higher-level libraries built on the fiber_context facility.

---

*This is the approach taken by Boost.Fiber.*
control transfer mechanism

According to the literature, coroutine-like control-transfer operations can be distinguished into the concepts of symmetric and asymmetric operations.

**symmetric fiber** A symmetric fiber provides a single control-transfer operation. This single operation requires that the control is passed explicitly between the fibers.

```
1 fiber_context* pf1;
2 fiber_context f4{[&pf1] {
3     pf1->resume();
4 });
5 fiber_context f3{[&f4] {
6     f4.resume();
7 });
8 fiber_context f2{[&f3] {
9     f3.resume();
10 });
11 fiber_context f1{[&f2] {
12     f2.resume();
13 });
14 pf1=&f1;
15 f1.resume();
```

In the pseudo-code example above, a chain of fibers is created.

Control is transferred to fiber `f1` at line 15 and the lambda passed to constructor of `f1` is entered. Control is transferred from fiber `f1` to `f2` at line 12 and from `f2` to `f3` (line 9) and so on. Fiber `f4` itself transfers control directly back to fiber `f1` at line 3.

**asymmetric fiber** Two control-transfer operations are part of asymmetric fiber’s interface: one operation for resuming (`resume()`) and one for suspending (`suspend()`) the fiber. The suspending operation returns control back to the calling fiber.

```
1 // hypothetical API
2 fiber_context f4{[] {
3     self::suspend();
4 });
5 fiber_context f3{[&f4] {
6     f4.resume();
7     self::suspend();
8 });
9 fiber_context f2{[&f3] {
10     f3.resume();
11     self::suspend();
12 });
13 fiber_context f1{[&f2] {
14     f2.resume();
15     self::suspend();
16 });
```
In the pseudo code above execution control is transferred to fiber \( f_1 \) at line 16. Fiber \( f_1 \) resumes fiber \( f_2 \) at line 13 and so on. At line 2 fiber \( f_4 \) calls its suspend operation \( \text{self}::\text{suspend()} \). Fiber \( f_4 \) is suspended and \( f_3 \) resumed. Inside the lambda, \( f_3 \) returns from \( f_4.\text{resume()} \) and calls \( \text{self}::\text{suspend()} \) (line 6). Fiber \( f_3 \) gets suspended while \( f_2 \) will be resumed and so on ... The asymmetric version needs \( N-1 \) more fiber switches than the variant using symmetric fibers.

Symmetric fibers represent independent threads of execution, making symmetric fibers a suitable mechanism for concurrent programming. Additionally, constructs that produce sequences of values (generators) are easily constructed out of two symmetric fibers (one represents the caller, the other the callee).

Asymmetric fibers incorporate additional fiber switches as shown in the pseudo code above. It is obvious that for a broad range of use cases, asymmetric fibers are less efficient than their symmetric counterparts.

Additionally, the calling fiber must be kept alive until the called fiber terminates. Otherwise the call of \( \text{suspend()} \) will be undefined behaviour (where to transfer execution control to?).

Symmetric fibers are more efficient, have fewer restrictions (no caller-callee relationship) and can be used to create a wider set of applications (generators, cooperative multitasking, backtracking ...).

**fiber_context as a first-class object**

Because the symmetric control-transfer operation requires explicitly passing control between fibers, fibers must be expressed as first-class objects.

Fibers exposed as first-class objects can be passed to and returned from functions, assigned to variables or stored into containers. With fibers as first-class objects, a program can explicitly control the flow of execution by suspending and resuming fibers, enabling control to pass into a function at exactly the point where it previously suspended.

**encapsulating the stack**

Each fiber is associated with a function call stack and is responsible for managing the lifespan of its stack (allocation at construction, deallocation when fiber terminates). The RAII-pattern\(^*\) should apply.

Copying a fiber_context must not be permitted!

If a fiber_context were copyable, then its stack with all the objects allocated on it must be copied too. That presents two implementation choices.

- One approach would be to capture sufficient metadata to permit object-by-object copying of stack contents. That would require dramatically more runtime information than is presently available – and would take considerably more overhead than a coder might expect. Naturally, any one move-only object on the stack would prohibit copying the entire stack.

- The other approach would be a bytewise copy of the memory occupied by the stack. That would force undefined behaviour if any stack objects were RAII-classes (managing a resource via RAII pattern). When the first of the fiber copies terminates (unwinds its stack), the RAII class destructors will release their managed resources. When the second copy terminates, the same destructors will try to doubly-release the same resources, leading to undefined behaviour.

\(^*\)resource acquisition is initialisation
A fiber API must:
- encapsulate the stack
- manage lifespan of an explicitly_allocated stack: the stack gets deallocated when fiber_context goes out of scope
- prevent accidentally copying the stack
Class fiber_context must be move-only.

invalidation at resumption

The framework must prevent the resumption of an already running or terminated (computation has finished) fiber. Resuming an already running fiber will cause overwriting and corrupting the stack frames (note, the stack is not copyable). Resuming a terminated fiber will cause undefined behaviour because the stack might already be unwound (objects allocated on the stack were destroyed or the memory used as stack was already deallocated).

As a consequence each call of resume() will empty the fiber_context instance.

Whether or not a fiber_context is empty can be tested with member function operator bool().

To make this more explicit, functions resume() and resume_with() are rvalue-reference qualified.

The essential points:
- regardless of the number of fiber_context declarations, exactly one fiber_context instance represents each suspended fiber
- no fiber_context instance represents the currently_running fiber

Section solution: avoiding non-const global variables and undefined behaviour describes how an instance of fiber_context is synthesized from the active fiber that suspends.

A fiber API must:
- prevent accidentally resuming a running fiber
- prevent accidentally resuming a terminated fiber
- resume() and resume_with() are rvalue-reference qualified

problem: avoiding non-const global variables and undefined behaviour

According to C++ core guidelines, non-const global variables should be avoided: they hide dependencies and make the dependencies subject to unpredictable changes.

Global variables can be changed by assigning them indirectly using a pointer or by a function call. As a consequence, the compiler can’t cache the value of a global variable in a register, degrading performance (unnecessary loads and stores to global memory especially in performance critical loops).

Accessing a register is one to three orders of magnitude faster than accessing memory (depending on whether the cache line is in cache and not invalidated by another core; and depending on whether the page is in the TLB).

The order of initialisation (and thus destruction) of static global variables is not defined, introducing additional problems with static global variables.

A library designed to be used as building block by other higher-level frameworks should avoid introducing global variables. If this API were specified in terms of internal global variables, no higher level layer could undo that: it would be stuck with the global variables.
**switch back to main() by returning**  Switching back to `main()` by returning from the fiber function has two drawbacks: it requires an internal global variable pointing to the suspended `main()` and restricts the valid use cases.

```c
int main() {
    fiber_context f([]{
        ...
        // switch to 'main()' only by returning
    });
    f.resume(); // resume 'f'
    return 0;
}
```

For instance the generator pattern is impossible because the only way for a fiber to transfer execution control back to `main()` is to terminate. But this means that no way exists to transfer data (sequence of values) back and forth between a fiber and `main()`.

> Switching to `main()` only by returning is impractical because it limits the applicability of fibers and requires an internal global variable pointing to `main()`.

**static member function returns active fiber_context**  P0099R0[^0] introduced a static member function `fibert_context::current()` that returned an instance of the active fiber. This allows passing the active fiber `m` (for instance representing `main()`) into the fiber `f` via lambda capture. This mechanism enables switching back and forth between the fiber and `main()`, enabling a rich set of applications (for instance generators).

```c
int main(){
    int a;
    fiber_context m=fiber_context::current(); // get active fiber
    fiber_context f([&] {
        a=0;
        int b=1;
        for(;;) {
            m=m.resume(); // switch to 'main()'
            int next=a+b;
            a=b;
            b=next;
        }
    });
    for(int j=0; j<10; ++j) {
        f=f.resume(); // resume 'f'
        std::cout << a << " ";
    }
    return 0;
}
```

But this solution requires an internal global variable pointing to the active fiber and some kind of reference counting. Reference counting is needed because `fiber_context::current()` necessarily requires multiple instances of `fiber_context` for the active fiber. Only when the last reference goes out of scope can the fiber be destroyed and its stack deallocated.

```c
fiber_context f1=fiber_context::current();
fiber_context f2=fiber_context::current();
assert(f1==f2); // f1 and f2 point to the same (active) fiber
```

Additionally a static member function returning an instance of the active fiber would violate the protection requirements of sections encapsulating the stack and invalidation at resumption. For instance you could accidentally attempt to resume the active fiber by invoking `resume()`.

```c
fiber_context m=fiber_context::current();
m.resume(); // tries to resume active fiber == UB
```

> A static member function returning the active fiber requires a reference counted global variable and does not prevent accidentally attempting to resume the active fiber.
solution: avoiding non-const global variables and undefined behaviour

The *avoid non-const global variables* guideline has an important impact on the design of the *fiber_context* API!

**synthesizing the suspended fiber**  The problem of global variables or the need for a static member function returning the active fiber can be avoided by *synthesizing the suspended fiber* and passing it into the resumed fiber (as parameter when the fiber is first started, or returned from *resume()*).

```cpp
void foo()
{
    fiber_context f{[ ]}(fiber_context&& m)
    {  
        m=std::move(m).resume(); // switch to 'foo'
        m=std::move(m).resume(); // switch to 'foo'
        ...
    }
    f=std::move(f).resume(); // start 'f'
    f=std::move(f).resume(); // resume 'f'
    ...
}
```

In the pseudo-code above the fiber *f* is started by invoking its member function *resume()* at line 7. This operation suspends *foo*, empties instance *f* and synthesizes a new *fiber_context* *m* that is passed as parameter to the lambda of *f* (line 2).

Invoking *m.resume()* (line 3) suspends the lambda, empties *m* and synthesizes a *fiber_context* that is returned by *f.resume()* at line 7. The synthesized *fiber_context* is assigned to *f*. Instance *f* now represents the suspended fiber running the lambda (suspended at line 3). Control is transferred from line 3 (lambda) to line 7 (*foo()*).

Call *f.resume()* at line 8 empties *f* and suspends *foo* again. A *fiber_context* representing the suspended *foo* is synthesized, returned from *m.resume()* and assigned to *m* at line 3. Control is transferred back to the lambda and instance *m* represents the suspended *foo*. Function *foo()* is resumed at line 4 by executing *m.resume()* so that control returns at line 8 and so on ...

Class *symmetric_coroutine<>::yield_type* from N3985 is **not** equivalent to the synthesized *fiber_context*. *symmetric_coroutine<>::yield_type* does not represent the suspended context, instead it is a special representation of the same coroutine. Thus *main()* or the current thread’s entry-function can **not** be represented by *yield_type* (see next section representing *main()* and thread’s entry-function as fiber).

Because *symmetric_coroutine<>::yield_type()* yields back to the starting point, i.e. invocation of *symmetric_coroutine<>::call_type::operator()()*, both instances (*call_type* as well as *yield_type*) must be preserved. Additionally the caller must be kept alive until the called coroutine terminates or UB happens at resumption.

This API is specified in terms of passing the suspended *fiber_context*. A higher level layer can hide that by using private variables.

**representing main() and thread’s entry-function as fiber**  As shown in the previous section a synthesized instance of *fiber_context* is passed into the resumed fiber.

```cpp
int main()
{
    fiber_context f{[ ]}(fiber_context&& m)
    {  
        m=std::move(m).resume(); // switch to 'main'
        ...
    }
    f=std::move(f).resume(); // resume 'f'
    ...
    return 0;
}
```

The mechanism presented in this proposal describes switching between stacks: each fiber has its own stack. The stacks of main() and explicitly-launched threads are not excluded; these can be used as targets too.
Thus every program can be considered to consist of fibers – some created by the OS (main() stack; each thread’s initial stack) and some created explicitly by the code.

This is a nice feature because it allows (the stacks of) main() and each thread’s entry-function to be represented as fibers. A fiber_context representing main() or a thread’s entry-function can be handled like an explicitly created fiber_context: it can passed to and returned from functions or stored in a container.

In the code snippet above the suspended main() is represented by instance m and could be stored in containers or managed just like f by a scheduling algorithm.

The proposed fiber API allows representing and handling main() and the current thread’s entry-function by an instance of fiber_context in the same way as explicitly created fibers.

**fiber returns (terminates)** When a fiber returns (terminates), what should happen next? Which fiber should be resumed next? The only way to avoid internal global variables that point to main() is to explicitly return a non-empty fiber_context instance that will be resumed after the active fiber terminates.

```cpp
int main(){
    fiber_context f{[](fiber_context&& m){
        return std::move(m); // resume 'main()' by returning 'm'
    }};
    f = std::move(f).resume(); // resume 'f'
    assert(f.empty());
    return 0;
}
```

In line 5 the fiber is started by invoking resume() on instance f. main() is suspended and an instance of type fiber_context is synthesized and passed as parameter m to the lambda at line 2. The fiber terminates by returning m. Control is transferred to main() (returning from f.resume() at line 5) while fiber f is destroyed.

In a more advanced example another fiber_context is used as return value instead of the passed in synthesized fiber.

```cpp
int main(){
    fiber_context m;
    fiber_context f1{[](fiber_context&& f){
        std::cout << "f1: entered first time" << std::endl;
        assert(!f);
        return std::move(m); // resume (main-)fiber that has started 'f2'
    }};
    fiber_context f2{[](fiber_context&& f){
        std::cout << "f2: entered first time" << std::endl;
        m=std::move(f); // preserve 'f' (== suspended main())
        return std::move(f1);
    }};
    std::move(f2).resume();
    std::cout << "main: done" << std::endl;
    return 0;
}
```

At line 13 fiber f2 is resumed and the lambda is entered at line 8. The synthesized fiber_context f (representing suspended main()) is passed as a parameter f and stored in m (captured by the lambda) at line 10. This is necessary in order to prevent destructing f when the lambda returns. Fiber f2 uses f1, that was also captured by the lambda, as return value. Fiber f2 terminates while fiber f1 is resumed (entered the first time). The synthesized fiber_context f passed into the lambda at line 3 represents the terminated fiber f2 (e.g. the calling fiber). Thus instance f is empty as the assert statement verifies at line 5. Fiber f1 uses the captured fiber_context m as return value (line 6). Control is returned to main(), returning from f2.resume() at line 13.
The entry-function passed to `fiber_context`'s constructor must have signature `fiber_context(fiber_context&&)` . Using `fiber_context` as the return value from such a function avoids global variables.

**returning synthesized fiber_context instance from resume()** An instance of `fiber_context` remains empty after return from `resume()` or `resume_with()`; the synthesized fiber is returned, instead of implicitly updating the `fiber_context` instance on which `resume()` was called.

If the `fiber_context` object were implicitly updated, the fiber would change its identity because each fiber is associated with a stack. Each stack contains a chain of function calls (call stack). If this association were implicitly modified, unexpected behaviour happens.

The example below demonstrates the problem:

```cpp
int main(){
    fiber_context m, f1, f2, f3;
    f3 = fiber_context{[&](fiber_context&& f)->fiber_context{
        f2 = std::move(f);
        for(;;){
            std::cout << "f3 ";
            std::move(f1).resume();
        }
        return {};
    }};
    f2 = fiber_context{[&](fiber_context&& f)->fiber_context{
        f1 = std::move(f);
        for(;;){
            std::cout << "f2 ";
            std::move(f3).resume();
        }
        return {};
    }};
    f1 = fiber_context{[&](fiber_context&& f)->fiber_context{
        m = std::move(f);
        for(;;){
            std::cout << "f1 ";
            std::move(f2).resume();
        }
        return {};
    }};
    std::move(f1).resume();
    return 0;
}
output:
```
f1 f2 f3 f1 f3 f1 f3 f1 f3 ...  
```

In this pseudo-code the fiber_context object is implicitly updated.

The example creates a circle of fibers: each fiber prints its name and resumes the next fiber (f1 -> f2 -> f3 -> f1 -> ...).

Fiber f1 is started at line 27. The synthesized `fiber_context` main passed to the resumed fiber is stored but not used: control flow cycles through the three fibers. The for-loop prints the name f1 and resumes fiber f2. Inside f2's for-loop the name is printed and f3 is resumed. Fiber f3 resumes fiber f1 at line 7. Inside f1 control returns from f2.resume(). f1 loops, prints out the name and invokes f2.resume(). But this time fiber f3 instead of f2 is resumed. This is caused by the fact the instance f2 gets the synthesized fiber_context of f3 implicitly assigned. Remember that at line 7 fiber f3 gets suspended while f1 is resumed through f1.resume().

This problem can be solved by returning the synthesized fiber_context from `resume()` or `resume_with()`.

```cpp
int main(){
    fiber_context m, f1, f2, f3;
    f3 = fiber_context{[&](fiber_context&& f)->fiber_context{
        //...
    }};
    //...
}
```
f2 = std::move(f);
for(;;){
    std::cout << "f3 ";
    f2 = std::move(f1).resume();
}
return {};
}
}
f2 = fiber_context{[&]{fiber_context&& f} -> fiber_context{
    f1 = std::move(f);
    for(;;){
        std::cout << "f2 ";
        f1 = std::move(f3).resume();
    }
    return {};
}(
}
}
f1 = fiber_context{[&]{fiber_context&& f} -> fiber_context{
    m = std::move(f);
    for(;;){
        std::cout << "f1 ";
        f3 = std::move(f2).resume();
    }
    return {};
}();
std::move(f1).resume();
return 0;
}

output:
  f1  f2  f3  f1  f2  f3  f1  f2  f3  ...

In the example above the synthesized fiber_context returned by each resume() call is specifically move-assigned to a fiber_context instance other than the one on which resume() was called, to properly track the three fibers. (Of course this particular example depends on static knowledge of the overall control flow. But the API does not, in general, require that.)

The synthesized fiber_context must be returned from resume() and resume_with() in order to prevent changing the identity of the fiber.

If the overall control flow isn’t known, member function resume_with() (see section inject function into suspended fiber) can be used to assign the synthesized fiber_context to the correct fiber_context instance (held by the caller).

class filament{
private:
    fiber_context f_;

public:
    ...;
    void resume_next( filament& fila){
        std::move(fila.f_).resume_with([this](fiber_context&& f) -> fiber_context{
            f_ = std::move(f);
            return {};
        });
    }
};

Picture a higher-level framework in which every fiber can find its associated filament instance, as well as others. Every context switch must be mediated by passing the target filament instance to the running fiber’s resume_next().

Running fiber A has an associated filament instance filamentA, whose fiber_context filament::f_ is empty – because fiber A is running.
Desiring to switch to suspended fiber B (with associated filament filamentB), running fiber A calls filamentA.resume_next(filamentB).

resume_next() calls filamentB.f_.resume_with(<lambda>). This empties filamentB.f_—because fiber B is now running.

The lambda binds &filamentA as this. Running on fiber B, it receives a fiber_context instance representing the newly-suspended fiber A as its parameter f. It moves that fiber_context instance to filamentA.f_.

The lambda then returns a default-constructed (therefore empty) fiber_context instance. That empty instance is returned by the previously-suspended resume_with() call in filamentB.resume_next()—which is fine because resume_next() drops it on the floor anyway.

Thus, the running fiber’s associated filament::f_ is always empty, whereas the filament associated with each suspended fiber is continually updated with the fiber_context instance representing that fiber.

It is not necessary to know the overall control flow. It is sufficient to pass a reference/pointer of the caller (fiber that gets suspended) to the resumed fiber that move-assigns the synthesized fiber_context to caller (updating the instance).

### inject function into suspended fiber

Sometimes it is useful to inject a new function (for instance, to throw an exception or assign the synthesized fiber to the caller as described in returning synthesized fiber_context instance from resume()) into a suspended fiber. For this purpose resume_with() may be called, passing the function fn() to execute.

```c++
fiber_context f([[]](fiber_context&& caller){
  // ...
  std::move(caller).resume();
  // ...
});

fiber_context fn(fiber_context&&);

f = std::move(f).resume();
// ...
std::move(f).resume_with(fn);
```

The resume_with() call at line 11 injects function fn() into fiber f as if the resume() call at line 3 had directly called fn().

Like an entry-function passed to fiber_context, fn() must accept std::fiber_context&& and return fiber_context. The fiber_context instance returned by fn() will, in turn, be returned to f's lambda by the resume() at line 3.

In the example below, suppose that code running on the program’s main fiber calls resume() (line 12), thereby entering the first lambda. This is the point at which m is synthesized and passed into the lambda at line 2.

Suppose further that after doing some work (line 4), the lambda calls m.resume(), thereby switching back to the main fiber. The lambda remains suspended in the call to m.resume() at line 5.

At line 18 the main fiber calls f.resume_with() where the passed lambda accepts fiber_context &. That new lambda is called on the fiber of the suspended lambda. It is as if the m.resume() call at line 8 directly called the second lambda.

The function passed to resume_with() has almost the same range of possibilities as any function called on the fiber represented by f. Its special invocation matters when control leaves it in either of two ways:

1. If it throws an exception, that exception unwinds all previous stack entries in that fiber (such as the first lambda's) as well, back to a matching catch clause.†

2. If the function returns, the returned fiber_context instance is returned by the suspended resume() or resume_with() call.

* Boost.Fiber uses this pattern for resuming user-land threads.

†As stated in exceptions, if there is no matching catch clause in that fiber, std::terminate() is called.
int data = 0;
fiber_context f([&data](fiber_context&& m){
    std::cout << "f1: entered first time: " << data << std::endl;
    data+=1;
    m=std::move(m).resume();
    std::cout << "f1: entered second time: " << data << std::endl;
    data+=1;
    m=std::move(m).resume();
    std::cout << "f1: entered third time: " << data << std::endl;
    return std::move(m);
});
f=std::move(f).resume();
std::cout << "f1: returned first time: " << data << std::endl;
data+=1;
f=std::move(f).resume();
std::cout << "f1: returned second time: " << data << std::endl;
data+=1;
f=std::move(f).resume_with([&data](fiber_context&& m){
    std::cout << "f2: entered: " << data << std::endl;
    data=-1;
    return std::move(m);
});
std::cout << "f1: returned third time" << std::endl;
}

output:
f1: entered first time: 0
f1: returned first time: 1
f1: entered second time: 2
f1: returned second time: 3
f2: entered: 4
f1: entered third time: -1
f1: returned third time

The f.resume_with(<lambda>) call at line 18 passes control to the second lambda on the fiber of the first lambda. As usual, resume_with() synthesizes a fiber_context instance representing the calling fiber, passed into the lambda as m. This particular lambda returns m unchanged at line 21; thus that m instance is returned by the resume() call at line 8. Finally, the first lambda returns at line 10 the m variable updated at line 8, switching back to the main fiber.

One case worth pointing out is when you call resume_with() on a fiber_context that has not yet been resumed for the first time:

fiber_context topfunc(fiber_context&& prev);
fiber_context injected(fiber_context&& prev);

fiber_context f(topfunc);
// topfunc() has not yet been entered
std::move(f).resume_with(injected);

In this situation, injected() is called with a fiber_context instance representing the caller of resume_with(). When injected() eventually returns that (or some other) fiber_context instance, the returned fiber_context instance is passed into topfunc() as its prev parameter.

Member function resume_with() allows you to inject a function into a suspended fiber.

passing data between fibers

Data can be transferred between two fibers via global pointer, a calling wrapper (like std::bind) or lambda capture.

int i=1;
std::fiber_context lambda{[&i](fiber_context&& caller){
std::cout << "inside lambda,i==" << i << std::endl;
i+=1;
caller=std::move(caller).resume();
return std::move(caller);
};
lambda=std::move(lambda).resume();
std::cout << "i==" << i << std::endl;
lambda=std::move(lambda).resume();

output:
inside lambda,i==1
i==2

The `resume()` call at line 8 enters the lambda and passes 1 into the new fiber. The value is incremented by one, as shown at line 4. The expression `caller.resume()` at line 5 resumes the original context (represented within the lambda by `caller`).

The call to `lambda.resume()` at line 10 resumes the lambda, returning from the `caller.resume()` call at line 5. The `fiber_context` instance `caller` emptied by the `resume()` call at line 5 is replaced with the new instance returned by that same `resume()` call.

Finally the lambda returns (the updated) `caller` at line 6, terminating its context.

Since the updated `caller` represents the fiber suspended by the call at line 10, control returns to `main()`.

However, since fiber `lambda` has now terminated, the updated `lambda` is empty. Its `operator bool()` returns `false`.

Using lambda capture is the preferred way to transfer data between two fibers; global pointers or a calling wrapper (such as `std::bind`) are alternatives.

**termination**

Every `fiber_context` you launch must terminate gracefully by returning from its entry-function.

When an explicitly-launched fiber’s entry-function returns a non-empty `fiber_context` instance, the running fiber is terminated. Control switches to the fiber indicated by the returned `fiber_context` instance. The entry-function may return (switch to) any reachable non-empty `fiber_context` instance – it need not be the instance originally passed in, or an instance returned from the `resume()` family of methods.

Calling `resume()` means: “Please switch to the indicated fiber; I am suspending; please resume me later.”

Returning a particular `fiber_context` means: “Please switch to the indicated fiber; and by the way, I am done.”

Cancellation of another fiber is not explicitly supported by `fiber_context`. If it is important for consuming code to communicate to a suspended fiber the desire that it should terminate, lambda binding may be used to pass some relevant object, e.g. a `stop_token`.

It is up to the code running on the fiber in question to observe and respond to any such termination request. The fiber must be resumed after the request before it could possibly observe the change. Even then, the entry-function might not immediately return.

One tactic would be to request termination, then loop over `resume()` or `resume_with()` calls until the returned `fiber_context` is `empty()`. However, that information is ambiguous.

Suppose we have a `fiber_context` instance `f1` representing suspended fiber F, with an application-specific termination request mechanism. The running fiber M requests F to terminate, then calls `f1.resume()`, which in due course returns another `fiber_context` instance `f2`.

`f2` has various possible values.

- `f2` might be empty. This might mean that fiber F did in fact terminate.
- Alternatively, it might mean that fiber F, instead of terminating, resumed fiber G, which terminated by resuming fiber M.
- Or fiber F might have terminated by resuming fiber G, which might have terminated by resuming fiber M.
• In other words, if $f_2$ is empty, fiber M cannot know the present state of fiber F.

• $f_2$ might not be empty. That might mean that fiber F did not terminate before resuming fiber M. $f_2$ would represent fiber F.

• Or it might mean that fiber F terminated by resuming fiber G, which might have resumed fiber M. $f_2$ would represent fiber G.

• Or it might mean that fiber F, instead of terminating, resumed fiber G, which resumed fiber M. $f_2$ would (again) represent fiber G.

• In other words, if $f_2$ is not empty, fiber M cannot know the present state of fiber F.

The autocancel class introduced in Appendix A: support code for examples illustrates a possible cancellation implementation, subject to the limitations described above.

exceptions

If an uncaught exception escapes from a fiber’s entry-function, std::terminate is called.

fiber_context as building block for higher-level frameworks

A low-level API enables a rich set of higher-level frameworks that provide specific syntaxes/semantics suitable for specific domains. As an example, the following frameworks are based on the low-level fiber switching API of Boost.Context (which implements the API proposed here).

Boost.Coroutine2 implements asymmetric coroutines coroutine<>::push_type and coroutine<>::pull_type, providing a unidirectional transfer of data. These stackful coroutines are only used in pairs. When coroutine<>::push_type is explicitly instantiated, coroutine<>::pull_type is synthesized and passed as parameter into the coroutine function. In the example below, coroutine<>::push_type (variable writer) provides the resume operation, while coroutine<>::pull_type (variable in) represents the suspend operation. Inside the lambda, in.get() pulls strings provided by coroutine<>::push_type’s output iterator support.

```cpp
struct FinalEOL { ~FinalEOL(){ std::cout << std::endl; } }
std::vector<std::string> words{ "peas", "porridge", "hot", "peas", "porridge", "cold", "peas", "porridge", "in", "the", "pot", "nine", "days", "old"};
int num=5, width=15;
boost::coroutines2::coroutine<std::string>::push_type writer{
    [&, std::vector<std::string> words](boost::coroutines2::coroutine<std::string>::pull_type& in){
        FinalEOL eol;
        for (;;) {
            for (int i=0; i<num; ++i){
                if (!in) {
                    return;
                }
                std::cout << std::setw(width) << in.get();
                in();
            }
            std::cout << std::endl;
        }
        std::copy(std::begin(words), std::end(words), std::begin(writer));
    };
}
```

Synca (by Grigory Demchenko) is a small, efficient library to perform asynchronous operations using source code that resembles synchronous operations. The main features are a GO-like syntax, support for transferring execution context explicitly between different thread pools or schedulers (portals/teleports) and asynchronous network support.
int fibo(int v){
    if (v<2) return v;
    int v1,v2;
    Waiter()
        .go({v,&v1}{ v1=fibo(v-1); })
        .go({v,&v2}{ v2=fibo(v-2); })
        .wait();
    return v1+v2;
}

The code itself looks like synchronous invocations while internally it uses asynchronous scheduling.

**Boost.Fiber** implements user-land threads and combines fibers with schedulers (the scheduler algorithm is a customization point). The API is modelled after the `std::thread` API and contains objects such as future, mutex, condition_variable...

boost::fibers::unbuffered_channel<unsigned int> chan;
boost::fibers::fiber f1{[&chan]{
    chan.push(1);
    chan.push(1);
    chan.push(2);
    chan.push(3);
    chan.push(5);
    chan.push(8);
    chan.push(12);
    chan.close();
}};
boost::fibers::fiber f2{[&chan]{
    for (unsigned int value: chan) {
        std::cout << value << " ";
    }
    std::cout << std::endl;
}};
f1.join();
f2.join();

**Facebook’s folly::fibers** is an asynchronous C++ framework using user-land threads for parallelism. In contrast to Boost.Fiber, folly::fibers exposes the scheduler and permits integration with various event dispatching libraries.

folly::EventBase ev_base;
auto fiber_manager=folly::fibers::getFiberManager(ev_base);
folly::fibers::Baton baton;
fiber_manager.addTask{[&]{
    std::cout << "task 1: start" << std::endl;
    baton.wait();
    std::cout << "task 1: after baton.wait()" << std::endl;
}};
fiber_manager.addTask{[&]{
    std::cout << "task 2: start" << std::endl;
    baton.post();
    std::cout << "task 2: after baton.post()" << std::endl;
}};
ev_base.loop();

**folly::fibers** is used in many critical applications at Facebook for instance in mcrouter and some other Facebook services/libraries like ServiceRouter (routing framework for Thrift), Node API (graph ORM API for graph databases) ...
Bloomberg’s *quantum* is a full-featured and powerful C++ framework that allows users to dispatch units of work (a.k.a. tasks) as coroutines and execute them concurrently using the 'reactor' pattern. Its main features are support for streaming futures which allows faster processing of large data sets, task prioritization, fast pre-allocated memory pools and parallel `forEach` and `mapReduce` functions.

```cpp
// Define a coroutine
int getDummyValue(Bloomberg::quantum::CoroContext<int>::Ptr ctx) {
    int value;
    ...  // do some work
    ctx->yield(); // be nice and let other coroutines run (optional cooperation)
    ...  // do more work and calculate 'value'
    return ctx->set(value);
}

// Create a dispatcher
Bloomberg::quantum::Dispatcher dispatcher;
// Dispatch a work item to do some work and return a value
int result = dispatcher.post(getDummyValue)->get();
```

*quantum* is used in large projects at Bloomberg.

**Habanero Extreme Scale Software Research Project** provides a task-based parallel programming model via its *HClib.* The runtime provides work-stealing, async-finish, parallel-for and future-promise parallel programming patterns. The library is not an exascale programming system itself, but it manages intra-node resources and schedules components within an exascale programming system.

Intel’s *TBB* internally uses fibers for long running jobs as reported by Intel.

As shown in this section a low-level API can act as building block for a rich set of high-level frameworks designed for specific application domains that require different aspects of design, semantics and syntax.

**interaction with STL algorithms**

In the following example STL algorithm `std::generate` and fiber `g` generate a sequence of Fibonacci numbers and store them into `std::vector v`.

```cpp
int a;
autocancel consumer, generator;
generator = autocancel([&a,&consumer,&generator](std::fiber_context&& m){
    a=0;
    int b=1;
    while (! generator.stop_requested()){  
        generator.resume(consumer);
        int next=a+b;
        a=b;
        b=next;
    }
    return std::move(m);
});
consumer = autocancel([&a,&consumer,&generator](std::fiber_context&& m){
    std::vector<int> v(10);
    std::generate(v.begin(), v.end(), [&a,&consumer,&generator](){ mutable {
        consumer.resume(generator);
        return a;
    }});
    std::cout << "v: ";
```

*async-finish is a variant of the fork-join model. While a task might fork a group of child tasks, the child tasks might fork even more tasks. All tasks can potentially run in parallel with each other. The model allows a parent task to selectively join a subset of child tasks."

†because of the requirement to support a broad range of architectures `swapcontext()` was used
for (auto i: v) {
    std::cout << i << " ";
}
std::cout << "$\n$;
return std::move(m);
}
consumer.resume();

output: v: 0 1 1 2 3 5 8 13 21 34

(See Appendix A: support code for examples for the definition of autocancel.)

possible implementation strategies

This proposal does NOT seek to standardize any particular implementation or impose any specific calling convention!

Modern micro-processors are register machines; the content of processor registers represents the execution context of the program at a given point in time.

Operating systems maintain for each process all relevant data (execution context, other hardware registers etc.) in the process table. The operating system’s CPU scheduler periodically suspends and resumes processes in order to share CPU time between multiple processes. When a process is suspended, its execution context (processor registers, instruction pointer, stack pointer, ...) is stored in the associated process table entry. On resumption, the CPU scheduler loads the execution context into the CPU and the process continues execution.

The CPU scheduler does a full context switch. Besides preserving the execution context (complete CPU state), the cache must be invalidated and the memory map modified.

A kernel-level context switch is several orders of magnitude slower than a context switch at user-level.

hypothetical fiber preserving complete CPU state This strategy tries to preserve the complete CPU state, e.g. all CPU registers. This requires that the implementation identifies the concrete micro-processor type and supported processor features. For instance the x86-architecture has several flavours of extensions such as MMX, SSE1-4, AVX1-2, AVX-512.

Depending on the detected processor features, implementations of certain functionality must be switched on or off. The CPU scheduler in the operating system uses such information for context switching between processes.

A fiber implementation using this strategy requires such a detection mechanism too (equivalent to swapper/system_32() in the Linux kernel).

Aside from the complexity of such detection mechanisms, preserving the complete CPU state for each fiber switch is expensive.

A context switch facility that preserves the complete CPU state like an operating system is possible but impractical for user-land.

fiber switch using the calling convention For fiber_context, not all registers need be preserved because the context switch is effected by a visible function call. It need not be completely transparent like an operating-system context switch; it only needs to be as transparent as a call to any other function. The calling convention – the part of the ABI that specifies how a function’s arguments and return values are passed – determines which subset of micro-processor registers must be preserved by the called subroutine.

The calling convention of SYSV ABI for x86_64 architecture determines that general purpose registers R12, R13, R14, R15, RBX and RBP must be preserved by the sub-routine - the first arguments are passed to functions via RDI, RSI, RDX, RCX, R8 and R9 and return values are stored in RAX, RDX.
So on that platform, the `resume()` implementation preserves the general purpose registers (R12-R15, RBX and RBP) specified by the calling convention. In addition, the stack pointer and instruction pointer are preserved and exchanged too – thus, from the point of view of calling code, `resume()` behaves like an ordinary function call.

In other words, `resume()` acts on the level of a simple function invocation – with the same performance characteristics (in terms of CPU cycles).

This technique is used in `Boost.Context` which acts as building block for (e.g.) `folly::fibers` and `quantum`; see section `fiber_context` as building block for higher-level frameworks.

**in-place substitution at compile time** During code generation, a compiler-based implementation could inject the assembler code responsible for the fiber switch directly into each function that calls `resume()`. That would save an extra indirection (JMP + PUSH/MOV of certain registers used to invoke `resume()`).

**CPU state on the stack** Because each fiber must preserve CPU registers at suspension and load those registers at resumption, some storage is required.

Instead of allocating extra memory for each fiber, an implementation can use the stack by simply advancing the stack pointer at suspension and pushing the CPU registers (CPU state) onto the stack owned by the suspending fiber. When the fiber is resumed, the values are popped from the stack and loaded into the appropriate registers.

This strategy works because only a running fiber creates new stack frames (moving the stack pointer). While a fiber is suspended, it is safe to keep the CPU state on its stack.

Using the stack as storage for the CPU state has the additional advantage that `fiber_context` need not itself contain the stored CPU state: it need only contain a pointer to the stack location. Section synthesizing the suspended fiber describes how global variables are avoided by synthesizing a `fiber_context` from the active fiber (execution context) and passing this synthesized `fiber_context` (representing the now-suspended fiber) into the resumed fiber. Using the stack as storage makes this mechanism very easy to implement. Inside `resume()` the code pushes the relevant CPU registers onto the stack, and from the resulting stack address creates a new `fiber_context`. This instance is then passed (or returned) into the resumed fiber (see synthesizing the suspended fiber).

**Using the active fiber’s stack as storage for the CPU state is efficient because no additional allocations or deallocations are required.**

**std::uncaught_exceptions() and std::current_exception()** Both `std::uncaught_exceptions()` and `std::current_exception()` should report exceptions solely on the current thread of execution. Reporting exceptions on any other thread of execution would make them unreliable in practice.

A straightforward implementation could make `resume()` and `resume_with()` save and restore the data underlying `std::uncaught_exceptions()` and `std::current_exception()` as part of saving and restoring the rest of the fiber state. Since `std::uncaught_exceptions()` and `std::current_exception()` data is necessarily thread-local, the likely cost would be a TLS access on every `resume()` or `resume_with()` call.

Alternatively, `fiber_context`’s constructor could update an internal associative container whose key is the high end of the new fiber stack. `std::uncaught_exceptions()` and `std::current_exception()` could call `upper_bound()`, passing the current stack pointer, to discover which stack is current. This would shift the cost from every context switch to `std::uncaught_exceptions()` and `std::current_exception()` calls.

In 2023, though, the zeitgeist appears to be that `std::uncaught_exceptions()` and `std::current_exception()` are not sufficiently important to warrant altering exception-handling implementations to fix their behavior. This is why, in the Wording section, it is implementation-defined whether `std::uncaught_exceptions()` and `std::current_exception()` are specific to the current thread of execution or the current `std::thread`.

---

*The implementation of `Boost.Context` utilizes this technique.*
fiber switch on architectures with register window

The implementation of fiber switch is possible – many libc implementations still provide the ucontext-API (swapcontext() and related functions)∗ for architectures using a register window (such as SPARC). The implementation of swapcontext() could be used as blueprint for a fiber implementation.

how fast is a fiber switch

A fiber switch takes 11 CPU cycles on a x86_64-Linux system† using an implementation based on the strategy described in fiber switch using the calling convention (implemented in Boost.Context,‡ branch fiber).

interaction with accelerators

For many core devices several programming models, such as OpenACC, CUDA, OpenCL etc., have been developed targeting host-directed execution using an attached or integrated accelerator. The CPU executes the main program while controlling the activity of the accelerator. Accelerator devices typically provide capabilities for efficient vector processing‡. Usually the host-directed execution uses computation offloading that permits executing computationally intensive work on a separate device (accelerator).§

For instance CUDA devices use a command buffer to establish communication between host and device. The host puts commands (op-codes) into the command buffer and the device processes them asynchronously.§

It is obvious that a fiber switch does not interact with host-directed device-offloading. A fiber switch works like a function call (see fiber switch using the calling convention).

multi-threading environment

Any thread in a program may be shared by multiple fibers.

A newly-instantiated fiber is not yet associated with any thread. However, once a fiber has been resumed the first time by some thread, it must thereafter be resumed only by that same thread.

There could potentially be Undefined Behaviour if:

- a function running on a fiber references thread_local variables
- the compiler/runtime implementation caches a pointer to thread_local storage in that function’s stack frame
- that fiber is suspended, and
- the suspended fiber is resumed on a different thread.

The cached TLS pointer is now pointing to storage belonging to some other thread. If the original thread terminates before the new thread, the cached TLS pointer is now dangling.

For this reason, it is forbidden to resume a fiber on any thread other than the one on which it was first resumed.

acknowledgments

The authors would like to thank Andrii Grynenko, Detlef Vollmann, Geoffrey Romer, Grigory Demchenko, Lee Howes, Daisy Sophia Hollman, Eric Fiselier, Yedidya Feldblum, Kirk Shoop and Lewis Baker.

---

∗ ucontext was removed from POSIX standard by POSIX.1-2008
† Intel XEON E5 2620v4 2.2GHz
‡ warp on CUDA devices, wavefront on AMD GPUs, 512-bit SIMD on Intel Xeon Phi
Wording

This wording is relative to N4958.

Append to 14.4.8 \[except.handle\] as indicated:

It is implementation-defined whether the currently handled exception designates the exception with the most recently activated handler that is still active within:

- the current thread of execution, or
- the current \texttt{std::thread}.

Append to 14.6.3.1 \[except.uncaught\] as indicated:

It is implementation-defined whether \texttt{uncaught_exceptions()} returns the number of uncaught exceptions in:

- the current thread of execution, or
- the current \texttt{std::thread}.

Insert new final subclause in clause 33 \[thread\] as indicated:

### 33.12 Cooperative Threads \[fibercontext\]

#### 33.12.1 General \[fibercontext.general\]

33.12.1.1 A fiber is a thread of execution (\[intro.multithread.general\]) with weakly parallel forward progress guarantees (\[intro.progress\] paragraph 11).

33.12.1.2 The term \textit{cooperative} means that typically multiple fibers share an underlying execution agent, for example a \texttt{std::thread} (\[thread.thread.class\]). On the underlying execution agent, only one fiber is running at any given time. Sharing that agent is explicit rather than pre-emptive. The running fiber \textit{suspends} (or \textit{yields}) to another fiber. This action \textit{launches} a new fiber, or \textit{resumes} a previously-suspended fiber.

33.12.1.3 Suspending the running fiber in order to resume (or launch) another is called \textit{context switching}. This is an explicit variant of blocking with forward progress guarantee delegation (\[intro.progress\] paragraph 14).

33.12.1.4 Launching a fiber logically creates a new function call stack, which remains associated with that fiber throughout its lifetime. Calling functions on a particular fiber, and returning from them, is independent of function calls and returns on any other fiber.

33.12.1.5 Context switching can be effected by designating some other fiber’s stack as current, in a manner appropriate to the existing implementation of function call stacks.

#### 33.12.2 Empty vs. Non-Empty \[fibercontext.empty\]

33.12.2.1 A \texttt{fiber_context} instance is either \textit{empty} or \textit{non-empty}. A default-constructed \texttt{fiber_context} is empty. A moved-from \texttt{fiber_context} is empty. A \texttt{fiber_context} representing a suspended fiber is non-empty.

33.12.2.2 When the running fiber returns a \texttt{fiber_context} from its entry-function, thus resuming the designated fiber, the synthesized \texttt{fiber_context} passed into the resumed fiber is empty.
33.12.3 Explicit Fiber vs. Implicit Fiber

33.12.3.1 The default thread on which the program runs main() has an initial default fiber whose stack is the stack on which main() is entered. [Note: Thus, when main() instantiates a new fiber_context, it becomes the second fiber in the program. — end note] Similarly, every explicitly-launched std::thread ([thread.thread.class]) or std::jthread ([thread.jthread.class]) has an initial default fiber whose stack is the stack on which the function passed to std::thread or std::jthread’s constructor is entered.

33.12.3.2 explicit fiber or explicitly-launched fiber designates a fiber instantiated by user code; conversely, implicit fiber designates the default fiber on any thread.

33.12.3.3 An explicit fiber is instantiated by passing an entry-function to fiber_context’s constructor. This function is not entered until the first call to one of the fiber_context::resume() family of methods.

- An implicit fiber’s owning thread is the thread of which that fiber is the default fiber.
- An explicit fiber initially has no owning thread.
- When some thread first calls resume() or resume_with() on an explicit fiber, that thread becomes the fiber’s owning thread.

33.12.3.4 When a fiber is first entered, a synthesized non-empty fiber_context instance representing the newly-suspended previous fiber is passed as a parameter to its entry-function. Once entered, a fiber may suspend by calling one of the resume() family of methods on any available non-empty fiber_context instance. When the suspended fiber is resumed, that method returns a synthesized fiber_context instance representing the newly-suspended previous fiber.

33.12.3.5 The synthesized fiber_context instance received in either of those ways might represent either an explicit fiber or an implicit fiber.

33.12.3.6 An explicit fiber terminates by returning from its entry-function. If the entry-function returns a non-empty fiber_context instance, the fiber represented by that fiber_context instance is resumed.

33.12.3.7 If the fiber’s entry-function exits via an exception, std::terminate is called.

33.12.4 Header <fiber_context> synopsis

namespace std {

// [fibercontext], class fiber_context
class fiber_context;

}

33.12.5 Class fiber_context

namespace std {

class fiber_context {
public:
// [fibercontext.cons], constructors, move and assignment
fiber_context() noexcept;

template<class F>
explicit fiber_context(F&& entry);

template<class F, class D>
fiber_context(F&& entry, span<byte> stack, D&& deleter);

}
fiber_context();

fiber_context(fiber_context&& other) noexcept;
fiber_context& operator=(fiber_context&& other) noexcept;
fiber_context(const fiber_context& other) noexcept = delete;
fiber_context& operator=(const fiber_context& other) noexcept = delete;

// [fibercontext.mem], members
fiber_context resume() &&;
template<class Fn>
fiber_context resume_with(Fn&& fn) &&;

bool can_resume() noexcept;

explicit operator bool() const noexcept;

bool empty() const noexcept;

void swap(fiber_context& other) noexcept;

// [fibercontext.special], specialized algorithms
friend void swap(fiber_context& lhs, fiber_context& rhs) noexcept;

private:
    void* state;  // exposition only
};

} // namespace std

33.12.5.1 Constructors, move and assignment

fiber_context() noexcept ;

33.12.5.2 Effects: instantiates an empty fiber_context.

33.12.5.3 Postconditions:empty() is true.

template<class F> explicit fiber_context(F&& entry) ;

33.12.5.4 Constraints: remove_cvref_t<F> is not the same type as fiber_context.

33.12.5.5 Mandates:
— is_copy_constructible_v<F> is true.
— is_invocable_r_v<fiber_context, decay_t<F>, fiber_context&&> is true.

33.12.5.6 Preconditions:F meets the Cpp17CopyConstructible requirements.

33.12.5.7 Let entry_copy be state associated with the new fiber that is not a data member of fiber_context.
[Note: entry_copy could be stored at the base of the new fiber’s stack. —end note]
33.12.5.8  **Effects:**
- Initializes `entry_copy` with `decay-copy` (std::forward<F>(entry)).
- Initializes `state` to represent a fiber suspended before entry to `entry_copy`. [*Note: entry_copy is entered only when resume() or resume_with() is called. — end note*]
- The new fiber’s function call stack and any other necessary resources are created.

33.12.5.9  **Postconditions:** `empty()` is `false`.

33.12.5.10  **Throws:**
- `bad_alloc` if unable to acquire a new function call stack.
- `system_error` if unable to start the new fiber.
- Any exception thrown by `entry`’s copy constructor.

33.12.5.11  **Error conditions:** `resource_unavailable_try_again` – the system lacked the necessary resources to create another fiber.

```cpp
template<class F, class D> fiber_context(F&& entry, span<byte> stack, D&& deleter) ;
```

33.12.5.12  **Mandates:**
- `is_copy_constructible_v<F> is true.`
- `is_copy_constructible_v<D> is true.`
- `is_invocable_r_v<fiber_context, decay_t<F>, fiber_context&&> is true.`
- `is_invocable_v<decay_t<D>, span<byte>> is true.`

33.12.5.13  **Preconditions:**
- `F` meets the `Cpp17CopyConstructible` requirements.
- `D` meets the `Cpp17CopyConstructible` requirements.

33.12.5.14  **Let `entry_copy`, `stack_copy` and `deleter_copy` be state associated with the new fiber that are not data members of `fiber_context`. [*Note: entry_copy, stack_copy and deleter_copy could be stored at the base of the new fiber’s stack. — end note*]}

33.12.5.15  **Effects:**
- Initializes `entry_copy` with `decay-copy` (std::forward<F>(entry)).
- Initializes `stack_copy` with `decay-copy` (std::forward<F>(stack)).
- Initializes `deleter_copy` with `decay-copy` (std::forward<F>(deleter)).
- Initializes `state` to represent a fiber suspended before entry to `entry_copy`. [*Note: entry_copy is entered only when resume() or resume_with() is called. — end note*]
- The contiguous block of uninitialized memory referenced by `stack` is prepared as the new fiber’s function call stack. [*Note: It is the caller’s responsibility to provide a span of sufficient size for the most deeply nested function calls that will be performed by the new fiber. It may be advantageous to obtain from the operating system a memory block with a read-only guard page to trap stack overflow. — end note*] Any other necessary resources are created.

33.12.5.16  **Postconditions:** `empty()` is `false.`
Throws:

- invalid_argument if `stack` fails to meet implementation-defined alignment requirements.
- length_error if `stack` is less than the implementation-defined minimum length.
- system_error if unable to start the new fiber.
- Any exception thrown by `entry`'s copy constructor.
- Any exception thrown by `deleter`'s copy constructor.

Error conditions: `resource_unavailable_try_again` – the system lacked the necessary resources to create another fiber.

Remarks: If at any time during the life of the newly created fiber the function call stack depth exceeds the size of `stack`, the behaviour is undefined.

```cpp
fiber_context(fiber_context&& other) noexcept;
```

Effects: Move constructs `this`'s `state` from `other`'s `state`.

Postconditions: `other`'s `empty()` is true.

```cpp
~fiber_context();
```

Effects: If `empty()` is false, `terminate` is invoked ([except.terminate]).

[Note: If a `fiber_context` instance to be destroyed is not yet empty, an application must convey to the suspended fiber the need to terminate voluntarily. — end note]

```cpp
fiber_context& operator=(fiber_context&& other) noexcept;
```

Effects:

- If `empty()` is false, `terminate` is invoked ([except.terminate]).
- Equivalent to: `this`'s `state = move(other`'s `state`).

Returns: `*this`

Postconditions: `other`'s `empty()` is true

Members

```cpp
template<class Fn> fiber_context resume_with(Fn&& fn) & &;
```

Mandates: `is_invocable_r_v<fiber_context, decay_t<Fn>, fiber_context&&>`

Preconditions: `can_resume()` is true
33.12.5.29 Let target be the fiber represented by state.

33.12.5.30 Let caller be a synthesized fiber_context instance representing the suspended calling fiber.

33.12.5.31 Effects:
 — Suspends the calling fiber.
 — Resets state so that empty() is true.
 — Resumes target.
 — Evaluates `invoke(std::forward<Fn>(fn), std::move(caller))` on fiber target. Let returned be the fiber_context instance returned by fn. [Note: returned may or may not be the same as caller. — end note] [Note: returned may be empty. — end note]
 — If target has not previously been entered, passes returned to its entry-function. Let continuation be the result of executing `invoke_r<fiber_context>(entry_copy, std::move(returned))`. On return:
   • restores the execution context of continuation
   • destroys entry_copy
   • if target has an associated stack_copy and deleter_copy:
     – executes `invoke(deleter_copy, stack_copy)`
     – destroys deleter_copy
   • otherwise reclaims the implementation-provided stack
   • resumes continuation as if by `continuation.resume()`.
 — Otherwise, target previously suspended itself by calling one of `resume()` or `resume_with()`. Returns returned from that resume function.

33.12.5.32 Remarks: A newly constructed but not yet resumed fiber may be resumed by any thread.

33.12.5.33 Returns:
 — If the previous fiber resumed this one by returning a fiber_context, an empty fiber_context.
 — If the previous fiber resumed this one by passing some fn to `resume_with()`, the fiber_context returned by that fn.

33.12.5.34 Throws:
 — Nothing before suspending the calling fiber and ensuring empty() is true.
 — On being resumed:
   • If the previous fiber resumed this one by returning a fiber_context:
     – Any exception thrown as a result of destroying the previous fiber’s associated entry_copy.
     – Any exception thrown by the previous fiber’s associated deleter_copy.
     – Any exception thrown as a result of destroying the previous fiber’s associated deleter_copy.
   • If the previous fiber resumed this one by calling `resume_with()`:
     – Any exception thrown by the fn passed by the previous fiber to `resume_with()`.
Postconditions: empty() is true.

[Note: The returned fiber_context indicates via empty() whether the previous active fiber has terminated (returned from entry-function). —end note]

[Note: resume() or resume_with() empties the instance on which it is called. In order to express the state change explicitly, these methods are rvalue-reference qualified. For this reason, no fiber_context instance ever represents the currently-running fiber. —end note]

```cpp
fiber_context resume() && ;
```

Effects: Equivalent to:

```cpp
return resume_with([](fiber_context&& caller){ return std::move(caller); });
```

```cpp
bool canResume() noexcept ;
```

Returns:

- `false` if empty() is `true`
- `true` if the fiber represented by `this` has no owning thread
- `true` if the calling thread is the owning thread of the fiber represented by `this`
- `false` otherwise.

[Note: When can_resume() is `true`, the fiber_context instance may be resumed by resume() or resume_with(). —end note]

[Editorial note: can_resume() is intentionally not marked const. —end editorial note]

```cpp
bool empty() const noexcept ;
```

Returns: `false` if state represents a fiber of execution, `true` otherwise.

[Note: Regardless of the number of fiber_context declarations, exactly one fiber_context instance represents each suspended fiber. —end note]

```cpp
explicit operator bool() const noexcept ;
```

Effects: Equivalent to: (! empty()).

```cpp
void swap(fiber_context& other) noexcept ;
```

Effects: Equivalent to: swap(this->state, other.state).

Specialized algorithms

```cpp
friend void swap(fiber_context& lhs, fiber_context& rhs) noexcept ;
```

33.12.5.42  Effects: Equivalent to: `lhs.swap(rhs)`.

**Feature-test Macro**

Add a new feature-test macro to 17.3.2 `[version.syn]`:

```c
#define __cpp_lib_fiber_context 2023XXL // also in <fiber_context>
```
**Appendix A: support code for examples**

Destroying a non-empty `fiber_context` instance invokes Undefined Behaviour (see termination). To simplify code examples in this paper, we introduce an `autocancel` wrapper class that launches a fiber and tracks the sequence of `fiber_context` instances representing that fiber. When an `autocancel` instance is destroyed, it sets a stop flag and loops until the fiber voluntarily terminates.

```cpp
// notify_done is an RAII class that binds a bool& reference and, when
// destroyed, sets the referenced bool true.
class notify_done
{
public:
    notify_done(bool& done):
        done_(done)
    {
        done_ = false;
    }

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

    ~notify_done()
    {
        done_ = true;
    }

private:
    bool& done_;  
};

// autocancel is a wrapper class that launches a fiber and, when destroyed,
// implicitly requests stop on that fiber. It uses the tactic seen in the
// example 'filament' class to continually update the fiber_context
// representing the fiber of interest. (See "returning synthesized
// std::fiber_context instance from resume()")
class autocancel
private:
    std::fiber_context f_;  
    bool stop_flag_{false};
    bool done_(false);
public:
    autocancel() = default;
    template <typename Fn>
    autocancel(Fn&& entry_function)
    {
        f_ = std::fiber_context{[this, entry=std::forward<Fn>(entry_function)]
            (std::fiber_context&& prev)
        {
            notify_done term(done_);
            return entry(std::move(prev));
        }};
    }

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

    ~autocancel()
    {
        stop_flag_ = true;
    }
```
while (f_ && ! done_) {
    resume(*this);
}  
assert(done_);

bool stop_requested() const noexcept {
    return stop_flag_;  
}

// for initial entry from a plain fiber rather than an autocancel instance
std::fiber_context resume() {
    return std::move(f_).resume();
}

void resume( autocancel& ac) {
    std::move(ac.f_).resume_with(
        [this](std::fiber_context&& f)->std::fiber_context {
            f_ = std::move(f);
            return {};
        });
    
};
References

[1] SYS V AMD64 unwinding
[2] x64 Windows unwinding
[3] ARM64 Windows unwinding
[8] N3985: A proposal to add coroutines to the C++ standard library
[10] P0099R0: A low-level API for stackful context switching
[13] P0660R10: Stop Tokens and a Joining Thread
[15] P0772R1: Execution Agent Local Storage
[16] P0876R0: fibers without scheduler
[17] P0876R2: fibers without scheduler
[18] P0876R3: fibers without scheduler
[19] P0876R5: fibers without scheduler
[20] P0876R6: fibers without scheduler
[21] P0876R7: fibers without scheduler
[22] P0876R8: fibers without scheduler
[23] P0876R9: fibers without scheduler
[24] P0876R10: fibers without scheduler
[25] P0876R11: fibers without scheduler
[26] P0876R12: fibers without scheduler
[27] P0876R13: fibers without scheduler
[28] P1677R2: Cancellation is serendipitous-success
[29] P1820R0: Recommendations for a compromise on handling errors and cancellations in executors
[30] P2175R0: Composable cancellation for sender-based async operations
[31] C++ Core Guidelines
[32] System V Application Binary Interface AMD64 Architecture Processor Supplement
[33] Library Boost.Context
[34] Library Boost.Coroutine2
[35] Library Boost.Fiber
[36] Facebook’s mcrouter
[37] Facebook’s Thrift
[38] Facebook’s folly::fibers
[39] Bloomberg’s quantum
[40] Habanero Extreme Scale Software Research Project
[41] Habanero HClib
[42] Library Synca
[43] Intels’s TBB