N4411 | Task Block (formerly Task Region) R4

Pablo Halpern Arch Robison {pablo.g.halpern, arch.robison}@intel.com
Hong Hong Artur Laksberg Gor Nishanov Herb Sutter
{honghong, arturl, gorn, hsutter}@microsoft.com

2015-04-10

1 Abstract

This paper introduces C++ a library function template `define_task_block` and a library class `task_block` with member functions `run` and `wait` that together enable developers to write expressive and portable fork-join parallel code.

2 Document Status and History

The proposals in this document are targeted at a future Parallelism TS or, potentially, a separate TS.

The predecessor to this document, N4088, was approved by the Parallelism and Concurrency study group (SG1) at the June 2014 meeting in Rapperswil. This revision contains changes resulting from LEWG review at the November 2014 meeting in Urbana-Champaign.

The changes from N4088 are described below.

2.1 New guarantee for outermost task block

An *outermost task block* is one that is created (by a call to `define_task_block`) when there is no previously-active task block, i.e., at the outermost level of parallel execution. On return from the outermost task block, the caller's thread is restored as if the call had been to `define_task_block_restore_thread`. This new guarantee was added to help ease the introduction of parallel libraries into serial code by guaranteeing that, if called from serial code, a library function will always return on the same thread even if the library function uses parallelism internally.

2.2 Added *DECAY_COPY* for `run`

We borrowed some of the language used to describe `async` for the invocation of the asynchronous function in `task_block::run`. In particular, it is necessary to copy the invocable passed as a function argument to `run` so that the continuation can modify or destroy the argument without invalidating the asynchronously-called invocable.

2.3 Naming changes

A number of identifiers have changed names in this proposal for the following reasons:

- What was previously called a "task region" is now called a "task block". The term "task region" is used in OpenMP to mean something that is just close enough in meaning to cause confusion and just different enough that it cannot be considered the same thing. The C Parallel Language Extensions study group (CPLEX) also adopted the term "task block"
- LEWG observed that `task_region` is a noun and that it is better for function names to be verbs or verb phrases. Thus the verb "define" was prefixed to the names of the function templates.
- LEWG also observed that there was nothing final about `task_region_final`. Since the feature that distinguished `task_region_final` from `task_region` was that the former is guaranteed to return on the same C++ thread, the "final" suffix was changed to "restore_thread".
- Finally, since the "define" prefix was added to the function templates, there was no longer a need for a suffix to distinguish the *class* representing a task region from the *function* that defines it. Thus, the "handle" suffix was dropped.

Putting all of these changes together, the mapping from old names to new names is summarized in the following table:

Old (N4088)	New (N4411)
`task_region`	`define_task_block`
`task_region_final`	`define_task_block_restore_thread`
`task_region_handle`	`task_block`

2.4 Editorial changes

A number of non-technical changes were made to improve readability and prepare the document for review by a wider audience:

- The issues section has been replaced by a Design Decisions and Alternatives section, now that all of the issues have been resolved.
- This change history has been condensed.

2.5 Summary of previous changes

N4088 2014-06-21	Improved wording for asynchronous execution and thread switching.
N3991 2014-05-23	Added `task_region_handle` (now `task_block`) as an explicit means of communicating between the definition of the task block and the `run` function. Changed from terminally strict to fully strict semantics.
N3832 2014-01-17	Original version.

3 Motivation and Related Proposals

The Draft Parallelism TS, N4312, augments the STL algorithms with parallel execution policies. Programmers use these as a basis to write additional high-level algorithms that can be implemented in terms of the provided parallel algorithms. However, the scope of N4312 does not include lower-level mechanisms to express arbitrary fork-join parallelism.

Over the last several years, Microsoft and Intel have collaborated to produce a set of common libraries known as the Parallel Patterns Library (PPL) by Microsoft and the Threading Building Blocks (TBB) by Intel. The two libraries have been a part of the commercial products shipped by Microsoft and Intel. Additionally, the paper is informed by Intel's experience with Cilk Plus, an extension to C++ included in the Intel C++ compiler in the Intel Composer XE product and also in gcc 4.9.

The `define_task_block`, `task_block::run` and `task_block::wait` functions proposed in this document are based on the `task_group` concept that is a part of the common subset of the PPL and the TBB libraries. A previous proposal, N3711, was presented to the Committee at the Chicago meeting in 2013. N3711 closely follows the design of the PPL/TBB with slight modifications to improve exception safety.

This proposal adopts a simpler syntax than N3711 – one that is influenced by language-based concepts such as `spawn` and `sync` from Cilk and `async` and `finish` from X10. It improves on N3711 in the following ways:

- The exception handling model is simplified and more consistent with normal C++ exceptions.
- Most violations of strict fork-join parallelism can be enforced at compile time (with compiler recognition of the constructs, in some cases).
- The syntax allows scheduling approaches other than child stealing.

We aim to converge with the language-based proposal for low-level parallelism described in N3409 and related documents.

4 Overview

Consider an example of a parallel traversal of a tree, where a user-provided function `compute` is applied to each node of the tree, returning the sum of the results:

```
template<typename Func>
int traverse(node *n, Func&& compute)
{
    int left = 0, right = 0;

    define_task_block([&](task_block& tb) {
        if (n->left)
            tb.run([&] { left = traverse(n->left, compute); });
        if (n->right)
            tb.run([&] { right = traverse(n->right, compute); });
    });

    return compute(n) + left + right;
}
```

The example above demonstrates the use of two of the functions proposed in this paper, `define_task_block` and `task_block::run`.

The `define_task_block` function delineates a region in the program code potentially containing invocations of tasks spawned by the `run` member function of the `task_block` class.

The `run` function spawns a *task*, a unit of work that is allowed to execute in parallel with respect to the caller. Any parallel tasks spawned by `run` within the `define_task_block` are joined back to a single thread of execution on return from `define_task_block`.

`run` takes a user-provided function object `f` and starts it asynchronously – i.e. it may return before the execution of `f` completes. The implementation's scheduler may choose to run `f` immediately or delay running `f` until compute resources become available.

A `task_block` can be constructed only by `define_task_block` because it has no public constructors. Thus, `run` can be invoked (directly or indirectly) only from a user-provided function passed to `define_task_block`:

```
void g();

void f(task_block& tb)
{
    tb.run(g);    // OK, invoked from within define_task_block in h
}

void h()
{
    define_task_block(f);
}

int main()
{
    task_block tb; // Error: no public constructor
    tb.run(g);     // No way to call run outside of a define_task_block
    return 0;
}
```

5 Task Parallelism Model

5.1 Strict fork-join task parallelism

The model of parallelism supported by the constructs in this paper is called *strict fork-join task parallelism*, which has decades of research behind it and is the form of structured parallelism supported by all of the prominent parallel languages, including Cilk, X10, Habanero, and OpenMP. These languages can be subdivided into two groups: those with *fully-strict* semantics and those with *terminally-strict* semantics.

In both the fully-strict and terminally-strict models, there is a notion of a *task block* that "owns" all of the tasks spawned within it. A *child* task spawned within a task block is automatically joined when the task block ends (i.e., the program waits for it to finish before continuing).

In the fully-strict model (Cilk and Cilk Plus), a task cannot complete until it has joined with all of its immediate child tasks. This form of fork-join parallelism provides strong guarantees and make a program easy to reason about. The terminally-strict model (X10, Habanero, and OpenMP), relaxes this rule and allows a child task to join with an ancestor rather than with its immediate parent. See Guo2009 for a description of terminally strict computations.

We have elected to specify fully strict semantics because it is easier to implement efficiently and has a longer track record. We did retain a feature of terminally-strict languages like X10: the option for a child task to escape from a (synchronously-called) function. That is, a called function can spawn child tasks and return without joining with those tasks. This (useful) feature is not directly related to parallel strictness because a called function does not create a new task, but it does make the program less structured in that a function may return to its caller before it has completely finished (i.e., while sub-tasks are still running). The dangers

of this relaxation of structured function-call semantics are mitigated by the `task_block`, which, when passed from caller to callee, gives both the programmer and the compiler enough information to recognize the call as special.

This design choice means is that a `task_block` can be passed to a *synchronous* function call (or captured by synchronously-called lambda function), but not to an *asynchronous* function call. Thus the following code would have undefined behavior. This violation and most such innocent violations can be diagnosed by a savvy compiler:

```
define_task_block([&](auto& tb) {
    tb.run([&]{ g(tb); }); // Error, tb captured by asynchronous lambda
    ...
});
```

It is important to note that strict semantics (whether fully strict or terminally strict) is not a given in parallelism proposals. Coming from a background in concurrency, many people look to unstructured constructs such as `std::async`, citing their flexibility vs. the comparative rigidity of strict parallelism. While these constructs have their place, the research has shown that fine-grain, large-scale parallelism benefits from a highly-structured approach. Just as a compiler can implement much more efficient memory allocation for (highly structured) local variables than for (unstructured) heap-allocated variables, so, too, can a compiler and scheduler take advantage of the structure of strict fork-join parallelism to implement efficient queuing and scheduling of parallel tasks. The algorithms described in N4312 neither require nor benefit from unstructured parallelism.

5.2 Non-mandatory parallelism

Whereas concurrency constructs such as threads, producer-consumer queues, and the like are primarily about program *structure*, parallelism constructs of the kind presented in this paper are primarily about maximum exploitation of available hardware resources to achieve *performance*. Critical to this distinction is that, while separate threads are independently expected to make forward progress, parallel tasks are not.

The most common scheduling technique for fork-join parallelism is called work-stealing. In a work-stealing scheduler, each hardware resource (usually a CPU core) maintains a queue (which may or may not be FIFO) of tasks that are ready to run. If a CPU's queue becomes empty, it "steals" a task from the queue of some other CPU. In this way, the CPUs stay busy and process their work as quickly as possible. Conversely, if all of the CPUs are busy working on their own tasks, then those tasks will be executed serially until the queues are empty. In fact, if the operating system allocates only one CPU to a process, then the entire parallel computation would be completed on a single core. This automatic load balancing allows a program to scale efficiently from one core to many cores without recompilation.

The constructs in this paper allow a programmer to indicate tasks that are *permitted* to run in parallel, but does not *mandate* that they actually *run* concurrently. An important consequence of this approach, known as "serial semantics," is that a task in the queue will not make any forward progress until another task (on the same or different core) completes, whether or not there is a dependency relationship between them. Thus, using concurrency constructs such as producer-consumer queues between parallel tasks (including between parent and child tasks or between sibling tasks) is a sure way to achieve deadlock. If the program is not valid as a serial program, then it is not valid as a parallel program, either.

6 Interface

The proposed interface is as follows. With the exception of `define_task_block_restore_thread`, the implementation of each of the functions defined herein is permitted to return on a different thread than that from which it was invoked. See Thread switching in the design section for an explanation of when this matters and how surprises can be mitigated.

6.1 Header <experimental/task_block> synopsis

```
namespace std {
  namespace experimental {
  namespace parallel {
  inline namespace v2 {

    class task_canceled_exception;

    class task_block;

    template<typename F>
      void define_task_block(F&& f);
    template<typename F>
      void define_task_block_restore_thread(F&& f);
  }
  }
  }

}
```

6.2 Class task_canceled_exception

```
class task_canceled_exception : public exception {
public:
    task_canceled_exception() noexcept;
    task_canceled_exception(const task_canceled_exception&) noexcept;
    task_canceled_exception& operator=(const task_canceled_exception&) noexcept;
    virtual const char* what() const noexcept;
};
```

The class task_canceled_exception defines the type of objects thrown by task_block::run or task_block::wait if they detect that an exception is pending within the current parallel block. See Exception Handling, below.

6.3 Class task_block

```
class task_block {
private:
    // Private members and friends (for exposition only)
    template<typename F>
      friend void define_task_block(F&& f);
    template<typename F>
      friend void define_task_block_restore_thread(F&& f);

    task_block(_unspecified_);
    ~task_block();

public:
    task_block(const task_block&) = delete;
    task_block& operator=(const task_block&) = delete;
    task_block* operator&() const = delete;
```

```
    template<typename F>
      void run(F&& f);

    void wait();
};
```

The class `task_block` defines an interface for forking and joining parallel tasks. The `define_task_block` and `define_task_block_restore_thread` function templates create an object of type `task_block` and pass a reference to that object to a user-provided callable object.

An object of class `task_block` cannot be constructed, destroyed, copied, or moved except by the implementation of the task block library. Taking the address of a `task_block` object via `operator&` is ill formed. Obtaining its address by any other means (including `addressof`) results in a pointer with unspecified value; dereferencing such a pointer results in undefined behavior.

A `task_block` is *active* if it was created by the nearest enclosing task block, where "task block" refers to an invocation of `define_task_block` or `define_task_block_restore_thread` and "nearest enclosing" means the most recent invocation that has not yet completed. Code designated for execution in another thread by means other than the facilities in this section (e.g., using `thread` or `async`) are not enclosed in the task block and a `task_block` passed to (or captured by) such code is not active within that code. Performing any operation on a `task_block` that is not active results in undefined behavior.

On entry to the invocable argument to `task_block::run`, no `task_block` is active, including the `task_block` on which `run` was called. (The invocable object should not, therefore, capture a `task_block` from the surrounding block.) [*Example:*

```
define_task_block([&](auto& tb) {
    tb.run([&]{
        tb.run([] { f(); });        // Error: tb is not active within run
        define_task_block([&](auto& tb) { // Nested task block
            tb.run(f);              // OK: inner tb is active
            ...
        });
    });
    ...
});
```

– *end example*] [*Note:* implementations are encouraged to diagnose the above error at translation time – *end note*]

6.3.1 `task_block` member function template `run`

```
template<typename F>
  void run(F&& f);
```

Requires: `F` shall be `MoveConstructible`. *INVOKE(DECAY_COPY*`(std::forward<F>(f)`*))*, shall a valid expression.

Precondition: `this` shall be the active `task_block`.

Effects: Let *fcopy* = *DECAY_COPY*(`std::forward<F>(f)`). Computes *fcopy* synchronously within the current thread, then calls *INVOKE(fcopy)*. The call to *INVOKE* is permitted to run on an unspecified thread in an unordered fashion relative to the sequence of operations following the call to `run(f)` (the *continuation*), or indeterminately sequenced within the same thread as the continuation. The call to `run` synchronizes

with the call to *INVOKE*. The completion of *INVOKE* synchronizes with the next invocation of `wait` on the same `task_block` or completion of the nearest enclosing task block (i.e., the `define_task_block` or `define_task_block_restore_thread` that created this `task_block`).

Throws: `task_canceled_exception`, as described in Exception Handling.

Postconditions: A call to `run` may return on a different thread than that on which it was called. [*Note*: The call to `run` is sequenced before the continuation as if `run` returned on the same thread. – *end note*]

Remarks: The invocation of the user-supplied callable object `f` may be immediate or may be delayed until compute resources are available. `run` might or might not return before invocation of `f` completes.

6.3.2 `task_block` member function `wait`

```
void wait();
```

Precondition: `this` shall be the active `task_block`.

Effects: Blocks until the tasks spawned using this `task_block` have finished.

Throws: `task_canceled_exception`, as described in Exception Handling.

Postcondition: All tasks spawned by the nearest enclosing task block have finished. A call to `wait` may return on a different thread than that on which it was called. [*Note*: The call to `wait` is sequenced before subsequent operations as if `wait` returns on the same thread. – *end note*]

[*Example:*

```
define_task_block([&](auto& tb) {
    tb.run([&]{ process(a, w, x); }); // Process a[w] through a[x]
    if (y < x) tb.wait();             // Wait if overlap between [w,x) and [y,z)
    process(a, y, z);                 // Process a[y] through a[z]
});
```

– *end example*]

6.4 Function templates `define_task_block` and `define_task_block_restore_thread`

```
template<typename F>
  void define_task_block(F&& f);
template<typename F>
  void define_task_block_restore_thread(F&& f);
```

Requires: `F` shall be `MoveConstructible`. Given an lvalue `tb` of type `task_block`, the expression, `(void) f(tb)`, shall be well-formed.

Effects: Constructs a `task_block`, `tb`, and invokes the expression `f(tb)` on the user-provided object, `f`.

Throws: `exception_list`, as specified in Exception Handling.

Postcondition: All tasks spawned from `f` have finished execution. A call to `define_task_block` may return on a different thread than that on which it was called unless there are no task blocks active (see Class `task_block`) on entry to `define_task_block`, in which case the call returns on the same thread as that on which it was called. A call to `define_task_block_restore_thread` always returns on the same thread as that on which it was called. (See Thread switching in the Issues section.) [*Note*: The call to `define_task_block` is sequenced before subsequent operations as if `define_task_block` returns on the same thread. – *end note*]

Notes: It is expected (but not mandated) that `f` will (directly or indirectly) call `tb.run(_callable_object_)`.

7 Exception Handling

Every task block has an associated exception list. When the task block starts, its associated exception list is empty.

When an exception is thrown from the user-provided callable object passed to `define_task_block` or `define_task_block_restore_thread`, it is added to the exception list for that task block. Similarly, when an exception is thrown from the user-provided function object passed into `task_block::run`, the exception object is added to the exception list associated with the nearest enclosing task block. In both cases, an implementation may discard any pending tasks that have not yet been invoked. Tasks that are already in progress are not interrupted except at a call to `task_block::run` or `task_block::wait`, as described below.

If the implementation is able to detect that an exception has been thrown by another task within the same nearest enclosing task block, then `task_block::run` or `task_block::wait` may throw `task_canceled_exception`; these instances of `task_canceled_exception` are not added to the exception list of the corresponding task block.

When a task block finishes with a non-empty exception list, the exceptions are aggregated into an `exception_list` object (defined below), which is then thrown from the task block.

The order of the exceptions in the `exception_list` object is unspecified.

The `exception_list` class is described in N4312 and is defined as follows:

```
class exception_list : public exception
{
  public:
    typedef _unspecified_ iterator;

    size_t size() const noexcept;
    const_iterator begin() const noexcept;
    const_iterator end() const noexcept;

    const char* what() const noexcept override;
};
```

8 Scheduling Strategies

A possible implementation of the `task_block::run` is to spawn individual tasks and immediately return to the caller. These *child* tasks are then executed (or *stolen*) by a scheduler using a different (native) thread, based on the availability of hardware resources and other factors. The original *parent* thread may participate in the execution of the tasks when it reaches the join point (i.e. at the end of the execution of the callable object passed to the `define_task_block` or `define_task_block_restore_thread`). This approach to scheduling is known as *child stealing*.

Other approaches to scheduling exist. In the approach pioneered by Cilk, the parent thread immediately executes the spawned task at the spawn point. The execution of the rest of the function – i.e., the *continuation* – is stolen by the scheduler if there are hardware resources available. Otherwise, the parent thread returns from the spawned task and continues as if it had been a normal function call instead of a spawn. This approach to scheduling is known as *continuation stealing* (or *parent stealing*).

Both approaches have advantages and disadvantages. It has been shown that the continuation stealing approach provides better asymptotic space guarantees and prevents threads from stalling at a join point. Child stealing is generally easier to implement without compiler involvement. N3872 provides a worthwhile primer that addresses the differences between, and respective benefits of, these scheduling approaches.

It is the intent of this proposal to enable either scheduling approach and, in general, to be as open as possible to additional scheduling approaches.

9 Design Decisions and Alternatives

The constructs proposed in this paper have a strong theoretical foundation from previous work on language-based parallelism such as Cilk, X10, and Habanero. However, there are some practical issues that arise from trying to harmonize these constructs with the existing C++ threading model. For example, the ability in X10 and Habanero to return from a function without joining with sub-tasks is more difficult to achieve in C++ because the strict scoping of C++ function variables is less forgiving than the garbage collected variables in the other two languages.

This section describes a couple of important issues that we considered along with some discussion of why we chose the specific resolutions that we did.

9.1 Thread switching

9.1.1 Description of the issues

One of the properties of continuation stealing and greedy scheduling is that a `define_task_block`, `task_block::run` or `task_block::wait` call might return on a different thread than that from which it was invoked, assuming scheduler threads are mapped 1:1 to standard threads. This phenomenon, which is new to C++, can be surprising to programmers and break programs that rely on the OS thread remaining the same throughout the serial portions of the function (for example, in programs accessing GUI objects, mutexes, thread-local storage and thread ID).

There are a number of possible approaches to mitigate the problems caused by thread switching. In considering mitigation proposals, it was important to avoid overly-constraining future implementations in order to support today's limited view of threads. For example, this proposal does not require that parallelism be implemented using OS threads at all – it could be implemented using specialized hardware such as GPUs or using other facilities such as light-weight execution agents.

Additionally, the solution to problems caused by thread switching may be different for mutexes than for thread-local storage (TLS) and thread ID. For example, a prototype mutex exists that works well with thread switching and has nice theoretical properties that allow it to be used for both parallelism and for traditional concurrency. The desired behavior for thread-local storage varies depending on its intended use, even in the absence of thread switching. For example, the handle to a GUI object might need to be shared among all of the tasks executing on behalf of a single original thread. Conversely, thread-local caches should not be shared between concurrently-executing tasks. With or without thread switching, we will certainly need new TLS-like facilities.

9.1.2 Design in this paper

In this proposal, the `define_task_block_restore_thread` function template provides a minimal but powerful approach for addressing thread switching. Using this facility, a user can be sure that both thread-local variables and mutexes are in a consistent state before and after the execution of a parallel computation. This feature is expected to solve most of the issues that a user may run into in well-structured parallel code. Because `define_task_block_restore_thread` requires stalling at a join point, it can potentially reduce parallel speed-up. For this reason, our advice to users would be to use `define_task_block` except in circumstances where thread identity is important. In implementations that do not support greedy scheduling, the behavior of `define_task_block_restore_thread` would probably be identical to that of `define_task_block`.

The "outermost" call to `define_task_block` always returns on the thread from which it was called. This special-case simplifies the job of adding parallelism to serial programs. A task block can be added to a function within a program that was, until that point, entirely serial, without breaking the potential assumptions of the caller.

9.1.3 Alternatives considered

We also discussed the possibility of language or library constructs to mark a function as potentially returning on a different thread than that on which it was called. A straw-man proposal involved a `thread_switching` keyword that would be applied as a suffix in the declarator for such functions:

```
void f() thread_switching;
```

If function decorated with the `thread_switching` modifier were called from a function that did not have the modifier, the compiler would inject code at the call site that would, on return from the decorated function, stall the caller until the original thread became available to resume execution:

```
void f() thread_switching;

int main() {
    auto thread_id_begin = std::this_thread::get_id();
    f();
    auto thread_id_end = std::this_thread::get_id();
    assert(thread_id_end == thread_id_begin);
    return 0;
}
```

Alternatively, calling a decorated function from an undecorated function could simply be ill formed, requiring the programmer to call `define_task_block_restore_thread` explicitly to avoid an error:

```
void f() thread_switching;

void g() {
    f();                // ill-formed
}

void h() {
    define_task_block_restore_thread([]{
        f();            // OK
    });
}
```

Other approaches were considered, including making a theoretical distinction between a "thread" as defined in C++11 and a "worker" as the agent that executes tasks. Making this distinction would solve certain problems with parallelism and thread identity, including issues of object and thread lifetimes that the `thread_switching` keyword does not address. As we refine our notion of "execution agent", it may become attractive to adopt this terminology.

9.2 Returning with unjoined children

9.2.1 Description of the issue

The *escaping asynchronous children* feature of the constructs proposed in this paper allow a function to return to the caller while some of its child tasks are still running. As in the case of thread switching, this behavior can be surprising to programmers and break programs that rely on functions finishing their work before they return. Although unstructured concurrency constructs such fire-and-forget threads already violate these assumptions, we are attempting, in this paper, to define much more structured constructs that operate at a finer granularity of work. Programmers writing *structured* parallel code need to be put on notice when a function invoked in their program might spawn parallel tasks and return without joining them first. The compiler may need to generate heap-allocated stack frames for such functions and the optimizer might be impaired in doing its job if it needs to defensively assume that any function might return with child tasks still running.

9.2.2 Design in this paper

This proposal, unlike a previous revision, requires that a `task_block` be available in order to spawn a child task. It can be argued that the presence of a `task_block&` argument to a function is sufficient notice for both the compiler and the programmer to recognize that the called function might spawn children and return with them still running. Indeed, there seems to be little reason to pass a `task_block` to a called function except to allow exactly this usage.

9.2.3 Alternatives considered

We previously considered adding an `unjoined_children` decoration, similar to the `thread_switching` keyword described above. This decoration would be automatically inherited by lambda functions, so that common cases would not require the use of this keyword. This idea is explored in more detail in the original Task Region paper N3832. With the advent of `task_block`, it seems that this approach is unnecessary and is not discussed further in this revision.

9.3 Violating structured parallelism

9.3.1 Description of the issue

The `task_block` class was introduced (or re-introduced, as it was present in an early draft for task regions) in order to avoid "out of band" communication between `define_task_block` and `run`, which a number of committee members found to be troubling, especially during the experimentation phase when multiple incompatible implementations might exist and silently collide with one another. The addition of `task_block` solves a number of other problems, including that of returning with unjoined children (above). However, `task_block` also exposes a name that can be abused to violate structured parallelism. For example:

```
define_task_block([&](auto& tb1) {
    tb1.run(f);
    define_task_block([&](auto& tb2) {
        tb2.run(g);
        tb1.run(h);  // Using tb1 violates strict fork-join rules
        tb2.run(j);
    }
    k();
}
```

Simply allowing such code makes all parallel programs harder to reason about, both for tools such as race detectors and for human programmers. Additionally, support FOR scheduling children other than from the inner-most task block might require more expensive data structures in the scheduler and/or more expensive synchronization than the strict constructs. This proposal would make such usage undefined behavior, but it is unfortunate that we cannot make it ill-formed because this is a library-only interface. Nevertheless, we believe that most, if not all, such abuses can be caught by an implementation that integrates the parallelism library with the compiler.

Another way in which `task_block` can be misused is by passing one to an asynchronous call. An example of such misuse appears in the formal wording for `task_block`, above. Again, such abuses are generally detectable by a sufficiently sophisticated compiler, but it is unfortunate that we cannot declare such misuse "ill formed."

9.3.2 Design in this paper

It is our opinion that actual errors caused by programmers violating structured parallelism will be rare, especially when compared to the much larger set of hazards that are possible within a parallel or concurrent program. Moreover, the risk can be mitigating by following simple coding rules, such as always giving your `task_block` the same name, thus preventing two `task_block`s from being in scope at the same time. Thus, our approach to this potential problem can be summarized as "live with it."

9.3.3 Alternatives considered

The original version of this proposal did not have a `task_block` object whose that could be abused to violate structured parallelism. Instead, the runtime library would be require to track the dynamic nesting of `defined_task_block` calls to deduce the correct task block for any call to `run` or `wait`. The benefits of the `task_block` parameter, however, were considered to outweigh the small risk of structured parallelism violations caused by its introduction.

10 References

TBB Threading Building Blocks (TBB)

PPL Parallel Patterns Library (PPL)

Cilk The Cilk Project

Cilk Plus cilkplus.org home page

X10 X10 Home Page

Habanero Habanero Extreme Scale Software Research Project

OpenMP openmp.org home page

Guo2009 *Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism*, Yi Guo et. al., Rice University 2009

N4312 *Draft Technical Specification for C++ Extensions for Parallelism*, J. Hoberock (editor), 2014-11-21

N4088 *Task Region R3*, P. Halpern, A. Robison, H. Hong; A. Laksberg, G. Nishanov, H. Sutter, 2014-06-21

N3991 *Task Region R2*, P. Halpern, A. Robison, H. Hong; A. Laksberg, G. Nishanov, H. Sutter, 2014-05-23

N3832 *Task Region*, P. Halpern, A. Robison, H. Hong; A. Laksberg, G. Nishanov, H. Sutter, 2014-01-17

N3711 *Task Groups As a Lower Level C++ Library Solution To Fork-Join Parallelism*, A. Laksberg, H. Sutter, 2013-08-15

N3409 *Strict Fork-Join Parallelism*, Pablo Halpern, 2012-09-24

N3872 *A Primer on Scheduling Fork-Join Parallelism with Work Stealing*, Arch Robison, 2014-01