Document Number: N3557
Date: 2013-03-18

Pablo Halpern, Intel Corp.
pablo.g.halpern@intel.com

Considering a Fork-Join Parallelism Library

Abstract

There is general consensus in the Concurrency and Parallelism study group (SG1) that strict fork-join parallelism would be a desirable feature to add to C++. They asked me to research whether it is possible to create a pure library interface for strict fork-join parallelism that achieves the same benefits as the well-established keyword-based language interface pioneered by the Cilk project and proposed for standardization in N3409. The technical and aesthetic advantages offered by the language approach include simple syntax, appropriate lifetimes for arguments in asynchronous function calls, correct overload resolution for asynchronous function calls, clear and enforceable strictness, and correct exception handling. This paper describes the challenges of creating a comparable library interface and explores the possibility of making small, general-purpose, language changes to enable a library solution to overcome those challenges. Ultimately, however, the library interface shows significant weaknesses when integrating with core features such as object lifetimes and exception scope. The library interface in particular is susceptible to misuses which may introduce subtle problems into programs that would be hard for many programmers to diagnose. Since our goal is making parallel programming accessible to the widest possible range of programmers I question whether a library approach could ever achieve this goal.

Contents

1 Introduction

Although there was strong interest in adding strict fork-join parallelism to C++ during the October 2012 C++ Standards meeting in Portland, some members of the Concurrency and Parallelism study group (SG1) were less than enthusiastic about the language-based approach advocated in N3409. Those members reasoned that a library-based approach would be superior because:

- They oppose language changes that serve only a single purpose.

- Library changes are easier to move through the standardization process than are core language changes.
- They feel that library features are easier than core language features to deprecate if we later decide that our approach was flawed.
- Library solutions are easier for vendors implement and therefore faster to get to market.

SG1 assigned me the task of imagining a library interface that would give most of the benefits of the language interface. In places where the library interface would fall short, I was asked to consider whether small language changes could correct the problem, if such language changes were generally useful and not focused on parallelism.

It should be our goal that any fork-join parallelism feature added to C++ be accessible to the widest possible range of programmers, regardless of whether the feature is in language or library form (or a combination of both). It is important that parallelism not become an “advanced-user-only” feature that scares programmers away with arcane idioms and/or traps for the unwary.

I will say at the onset that, in many respects, I was given an impossible task. Implicit in the assignment was the understanding that if a good library interface could not be constructed then the language interface would garner greater support. No library proposal for fork-join parallelism has been proposed that is equal in quality to the language proposal, and I cannot prove that none is possible (that is, I cannot prove the negative). Nevertheless, I was up for doing the thought experiment. I have no fundamental philosophical objection to using a library interface, provided it does not significantly complicate the task of parallel programming compared to the Cilk-like interface proposed in Portland. However, it is important to consider whether the library interface preserves all the important properties of the language-based interface, in particular properties valuable to non-expert programmers.

2 Summary of N3409 Proposed Features

The following example is slightly modified from N3409. It shows a parallel tree walk in which a computation `f()` is performed on the value of each node in a binary tree, yielding an integer metric. The results of the computation are summed over the entire tree:

```
int tree_walk(node *n)
{
    int a = 0, b = 0, c = 0;
    cilk_block {
        if (n->left)
            a = cilk_spawn tree_walk(n->left);
        if (n->right)
```

```
            b = cilk_spawn tree_walk(n->right);
        c = f(n->value);
    }
    return a + b + c;
}
```
In the example, the presence of `cilk_spawn` indicates to the compiler that execution can proceed asynchronously to the next statement, without waiting for the recursive `tree_walk` calls to complete. A `cilk_spawn` defines a *task* – a piece of work that is permitted (but not required) to execute asynchronously with respect to the caller and with respect to other spawned tasks. The "strict" part of the model means that execution of the function does not proceed beyond the end of the `cilk_block` until all `cilk_spawn` expressions within the block complete. A function body is implicitly a `cilk_block`. (The example in the original proposal used `cilk_sync` to wait on asynchronous tasks instead of a lexically-scoped `cilk_block`. The addition of a `cilk_block` construct was suggested by the members of the committee. Keyword names are placeholders; Standard keywords (or attributes) can be determined at a later time.)

If the `cilk_spawn` and `cilk_block` keywords are removed from the example above, the result is a valid program called the "serialization". The serialization of a program is always a valid interpretation of the parallel program and is equivalent to running the parallel program with only one CPU core (or more precisely, one *worker*).

3 Advantages of Strict Fork-Join Parallelism

The strict fork-join parallelism shown above has some important advantages over less-structured approaches. The following is a condensation of the description of these advantages in N3409. (Note that these advantages do not require a language-based feature. Any implementation of the strict fork-join parallelism model, whether via a language or library feature, will exhibit these benefits.)

1. **Serial semantics**: A serial execution is always a legal interpretation of the parallel program. Testing parallel correctness can be separated from testing serial correctness.
2. **Composable performance**: Parallel computations can be nested arbitrarily without resource oversubscription. This is important for modularity.
3. **Parallelism is encapsulated:** A caller does not need to know whether a function uses parallelism internally. Asynchronous tasks do not accidentally "leak" from functions, causing data races and other problems.

4. **Local variables obey normal C++ rules:** Variables declared in a task block can be passed by reference to asynchronous children and will remain alive until the children return.
5. **Mathematical rigor enables powerful analysis tools:** The mathematical qualities of strict fork-join parallelism allow analysis tools to work within reasonable memory bounds. Local parallelism results in localized analysis.

4 Imagining a Simple Library Interface for Fork-Join Parallelism

It is possible to create a library interface for fork-join parallelism that, used correctly, provides all of the advantages listed in the previous section. We are interested in whether or not such a library interface can provide the same ease of use as the language-based interface. Ease of use needs to include both simple syntax and resistance to user error.

To explore the library possibility, I chose a syntax inspired by TBB's and PPL's `task_group` constructs as well as C++11's `std::async()`. I imagined the following class definition as a starting point:

```
namespace std {

  class task_group {
   public:
    task_group();
    task_group(const task_group&) = delete;
    task_group(task_group&&) = delete;

    ~task_group(); // automatically calls sync()

    template <class F, class... Args>
      void spawn(F f, Args&&...args);

    void sync();   // Waits for spawned tasks to complete
  };
} // namespace std
```

The key members of `task_group` are as follows:

`spawn(f, args...)`	Runs `f(args...)` asynchronously. This function may return before `f()` returns.
`sync()`	Waits for all spawned calls to complete.
`~task_group()`	The destructor calls `sync()`. Thus, no asynchronous functions can escape the scope of the `task_group`.

The copy and move constructors are deleted so that a program cannot violate strictness by returning a `task_group` up the call stack (but see below for other strictness issues with this interface).

Using this interface, the tree-walk subroutine is straightforward, and the syntax is relatively clean:

```
int tree_walk(node *n)
{
    int a = 0, b = 0, c = 0;
    {
        std::task_group tg;
        if (n->left)
            tg.spawn([&]{ a = tree_walk(n->left); });
        if (n->right)
            tg.spawn([&]{ b = tree_walk(n->right); });
        c = f(n->value);
    }
    return a + b + c;
}
```

The `task_group` interface was chosen over the `parallel_invoke` interface (see N3429) because it supports a run-time determination of the number of asynchronous calls. In the tree-walk case, zero, one, or two asynchronous recursive calls to `tree_walk` may be invoked in each recursion, depending on the state of `n` on entry to the function. To get the same effect from `parallel_invoke` would require the use of a painful continuation-passing pattern.

5 Library Flaws Needing Solutions

Although the simple tree-walk example makes a library interface look attractive, problems emerge when considering less trivial situations. In the list below, I describe situations that the language interface handles well but where the library solution falls short. In a subsequent section, I'll explore some potential remedies to the library shortcomings that involve small (and sometimes not so small) enhancements to the core language – ideally enhancements that benefit not only parallelism, but other aspects of C++ programming.

As will be demonstrated in the following, the most important areas where the language proposal is superior to the `task_group` construct are:

- Enforced strictness
- Exception handling
- Simple and transparent syntax in more complex situations such as complex parameter expression and return values.

To fully appreciate the challenges of trying to create a library interface that approaches the quality of the language interface in the areas above, some lower-impact issues must also be examined. For example, the return value of an asynchronous function call presents a special challenge for a library interface, one that directly affects the simplicity of the syntax. Therefore, in order to give a complete picture, I will present also present the following additional areas where the language proposal outperforms `task_group`:

- Parameter passing
- Delayed return values
- Overload resolution and template instantiation

These second-tier issues inform our understanding the top-tier issues of strictness, exception handling, and syntax, and will therefore be presented first.

5.1 Parameter Passing

Consider the following code fragment call using the `cilk_spawn` feature described in N3409:

```
class Xyz { ... };
Xyz h(int);

// argument v by reference, x by const reference, z by value
void f(std::vector<int>& v, const Xyz& x, int z);

vector<int> v(...);

...
void calc() {
    int j = q(), k = 2 * j;
    cilk_spawn f(v, h(j), k);   // asynchronous call
    ++j;
    k += 2;
    ...
}
```

In the asynchronous call to `f()`, the `v` argument is passed by reference and modified within `f()`.It is common to modify an array or vector in parallel, avoiding races among parallel operations by operating on disjoint subsets of elements. The call to `h(j)` is evaluated before the “detach point” of the asynchronous call and results in the construction of a temporary object that is passed by `const` reference to `f()`. The destructor for this temporary is deferred until `f()` completes, even though execution of `calc()` continues asynchronously. The `k` argument is, of course, passed by value. This carefully-considered parameter-passing protocol closely resembles that of a serial call while ensuring that any temporaries that are referenced by the asynchronous function remain live through the execution of that function. In fact, the protocol can be derived directly from a basic principle of `cilk_spawn`: **Any operations in the spawning expression that are not dependent on the function-call completion execute serially in the caller (parent) and all other operations execute in the detached context (child)**. In practice, we have found this principle and the rules that derive from it to be intuitive and to protect against subtle bugs.

Let's try to get the same thing accomplished using the `task_group` facility I put forth in Section 4. The simplest syntax for calling `f()` is to embed the call in a lambda:

```
void calc() {
    std::task_group tg;
    int j = q(), k = 2 * j;
    tg.spawn([&]{ f(v, h(j), k); });   // asynchronous call
    ++j;
    k += 2;
    …
}
```

Unfortunately, the code above has a race condition because `h(j)` and `++j` execute in parallel and copying `k` races with `k += 2`. Staying with the lambda syntax, there are several ways to avoid the race, depending on which form the programmer finds least confusing:

```
tg.spawn([&,=j,=k]{ f(v, h(j), k); });   // capture j & k by value
tg.spawn([&](int j, int k){ f(v, h(j), k); }, j, k);
int jtmp = j, ktmp = k;   // explicit copies: won't work in a loop
tg.spawn([&]{ f(v, h(jtmp), ktmp); }, j);   // use explicit copies
```

All of these workarounds have in common that they demand extra work from the programmer to make up for the fact that compiler is unable to do the analysis for them. The lambda by itself does not do the right kind of analysis and results in an impoverished interface. If we avoid the lambda syntax, we can rely a bit more on the spawn construct itself to do the correct bookkeeping. The `spawn()` member of `task_group`, like `std::async` takes a variadic list of arguments that it saves and passes to the functor call. Passing the arguments to `f()` through `task_group::spawn()`, we get:

```
tg.spawn(f, v, h(j), k);
```

Unfortunately, `task_group::spawn` does not have enough information to know that `v` should be captured by reference and `k` should be captured by value. `std::async` "solves" this problem by always capturing by value and forwarding as rvalue (except that arrays and functions decay to pointers). If we adopt the same approach for `task_group::spawn`, then the above call fails to compile because the first argument to `f()` cannot be bound to an rvalue. The fix for this problem is to use a `reference_wrapper to` pass `v` explicitly by reference:

```
tg.spawn(f, std::ref(v), h(j), k);
```

What we have should now should work correctly, but we had to abandon the lambda syntax in favor of something that no longer looks like a call to `f()` – the transformation to the parallel call from its serialization (see Section 2) and vice-versa cannot be considered trivial.

5.2 Delayed Return Values

When a function is called asynchronously, its return value is not available until the function returns. If we modify the example from the previous section such that `f()` returns a value of type `Abc`, the language-based proposal would capture the return value of `f()` using a simple syntax:

```
Abc r = cilk_spawn f(v, h(j), k);
```

Although the value of `r` is undefined until a later `cilk_sync` or until the end of the `cilk_block`, the strict scoping rules mean that these two events will be in the same block – probably within the same screenful of code. Unlike futures, the initialization of `r` will not be seen for the first time in some far-distant part of the code.

The simplest way to return a value with `task_group::spawn` would be using a lambda:

```
Abc r;                                  // Hope Abc is DefaultConstructible…
tg.spawn([&]{ r = f(v, h(j), k); });    //… and MoveAssignable!
```

This approach, as we know, runs afoul of all of the issues described for lambdas in section 5.1. An alternative is to create a variant of `task_group::spawn` that takes the return value by reference:

```
Abc r;                            // Hope Abc is DefaultConstructible…
tg.spawn_r(r, f, v, h(j), k);     //… and MoveAssignable!
```

As the comments indicate, this approach limits us to return values that are `DefaultConstructible` and `MoveAssignable`, but is otherwise workable for those who don't mind the syntax. One way to avoid the `DefaultConstructible` and `MoveAssignable` requirements is to use `optional<Abc>` as proposed in N3406:

```
std::optional<Abc> r;             // r starts out disengaged (i.e. nullopt)
tg.spawn_r(r, f, v, h(j), k);     // initializes r's contents by move-construction
```

There is a pitfall to the above use of `optional`, however: Just as in the previous example, the value of `r` is not usable until after `tg.sync()` is called, but `optional` has a test for emptiness. The trap is that users may be tempted to use `optional` as if it were a `future` and test it for "ready" status, even though `optional`'s semantics do not make any ordering guarantees:

```
std::optional<Abc> r;
tg.spawn_r(r, f, v, h(j), k);
…
if (r) // BAD IDEA: Test if r is ready.
    foo(*r);
```

A final option is to return some kind of future. The existing `std::future` is fairly heavy-weight, owing to the need for dynamically-allocated memory and synchronization with the corresponding promise. Thus, we might conceive of a light-weight `task_group::future` to serve as the return value of `task_group::spawn`:

```
{
    task_group tg;
    task_group::future<Abc> rf = tg.spawn(f, v, h(j), k);
    …
}
Abc r = rf.get();
```

This approach has a number of disadvantages:

- The future outlives its task group. This is not a technical problem, but I consider it an aesthetic one.
- The call to `get()` must be explicit.
- The call to `get()` can be confused with `std::future::get`, which blocks if the future is not ready. To allow for the widest possible range of efficient implementations, we do not want to require blocking semantics for `task_group::future`, which should be assumed to be ready when the `task_group` has synchronized, and not before.

But the worst problem with the `task_group::future` idea is that nobody has shown that it can be done efficiently. There was a group at the first SG1 meeting that tried to design something like it, and did not succeed. To be fair, the constraints they were trying to meet may have been different, so an effort connected to `task_group` might succeed where the other one did not.

5.3 Overload Resolution and Template Instantiation

Returning to the example from section 5.1, let's change the declaration of function `f` to be a template:

```
template <class T>
  void f(std::vector<int>& v, const T& x, int z);
```

The `cilk_spawn` statement works without change:

```
cilk_spawn f(v, h(j), k);   // asynchronous call
```

However, the corresponding `task_group::spawn` statement fails to compile:

```
tg.spawn(f, std::ref(v), h(j), k);   // error: f is a template
```

Similarly, if `f` is not a template but is overloaded, the above call will fail to compile because overload resolution has not yet occurred, and `f` is considered ambiguous. The obvious workaround for this problem is to use a lambda, accepting the need to work around all of the parameter-passing problems described in section 5.1:

```
tg.spawn([&,=j,=k]{ f(v, h(j), k); });
```

Unfortunately, this is the kind of hard-to-explain *dark corner* that could cause parallelism to become an advanced-user-only feature.

5.4 Enforced Strictness

Not all parallelism must be strict, but strict fork-join parallelism is an important subclass of parallelism, just as block-scoped variables are an important subclass of memory allocation. By deleting the copy and move constructors of my theoretical `task_group` class, I prevent the task group from escaping outside of the block in which it was declared. Unfortunately, this is not enough to prevent passing the block down to a called function. Even worse, a program could allocate a `task_group` on the heap.

The most insidious violation of strictness is when a `task_group` is captured by reference in a lambda expression. The following innocent-looking code violates strictness requirements:

```
task_group tg;
std::parallel_for(0, N, [&](int i){
    tg.spawn(f, i);
    g(i);   // Run g(i) in parallel with f(i)
});
```

The lambda expression within the `parallel_for` is a separate function call and should not use the `task_group` that was captured from the caller's scope. The result is that the `parallel_for` may return with children still running.

Of course, we can tell programmers "don't do that," but chances are the programmer wasn't doing it on purpose. Tools, also, would need to assume that `task_group` is used idiomatically, rather than being able to take advantage of inherent strictness guarantees. Without experience, it is not clear if that would be enough. Even `goto` can be used in a well-structured way, but that didn't prevent it from being considered harmful.

5.5 Exception Handling

Consider the following code using the language-based proposal:

```
try {
    cilk_spawn f();

    try {
        cilk_spawn g();
        h();
    }
    catch (...) {
        // Catch Block 2
        …
    }
}
catch (...) {
    // Catch Block 1
    …
}
```

If `f()` throws an exception, it is caught in Catch Block 1. If `g()` or `h()` throw an exception, it is caught in Catch Block 2. If more than one of these functions throws an exception, the one that is caught is the same as the one that would have been caught in the corresponding serial program. In other words, this behavior mimics the behavior of the same program with the `cilk_spawn` keywords removed (the serialization of the program). There is a difference from the serial behavior, of course, in that an exception thrown from `g()`, for example, does not prevent `h()` from running. Furthermore, since `g()` and `h()` can run in parallel, they may both throw, but only the exception from `g()` is propagated to the catch block. The latter situation can be addressed using a library interface to recover a list of lost exceptions.

This exception-handling behavior is possible because every `try` block is implicitly a `cilk_block` and thus can re-throw an exception at the implicit join point that occurs at the end of every `cilk_block`.

Trying to do something similar with `task_group` could lead programmers into a trap:

```
try { // Try Block 1
    task_group tg;
    tg.spawn(f);

    try { // Try Block 2
        tg.spawn(g);
        h();
    } // End of Try Block 2
    catch (...) {
        // Catch Block 2
        …
    }
} // End of Try Block 1
catch (...) {
    // Catch Block 1
    …
}
```

The first problem we encounter is that there is no place to re-throw an exception that is thrown by `f()` or `g()`. Logically, we would re-throw the exception at the join point. Up until now, `task_group` was specified with an implicit join point in the destructor, that is, when `tg` goes out of scope. Unfortunately, throwing an exception from within a destructor is considered a *Bad Thing*™. To avoid that possibility, we must add explicit calls to `task_group::sync()`:

```
        tg.sync();
    } // end of Try Block 1
    catch (...)
```

In other words, in order to avoid having destructors that throw, we need to give up automatic joining via RAII.

Now we have to contend with some additional problems. If `g()` throws an exception, it will be re-thrown at the `sync()` call shown above. This means that `g`'s exception will be caught in Catch Block 1, which is likely to be disturbing to the programmer who started with working serial code and added the parallelism. This problem stems from the lack of linguistic support for strictness. To get strictness in the presence of exceptions, each `try` block must define a region from which child tasks cannot escape. We can accomplish this manually with `task_group` by defining a nested `task_group`:

```
    try { // Try Block 1
        task_group tg1;
        tg1.spawn(f);

        try { // Try Block 2
            task_group tg2;
            tg2.spawn(g);
            h();
            tg2.sync();
        } // End of Try Block 2
        catch (...) {
            // Catch Block 2
            …
        }
        tg1.sync();
    } // End of Try Block 1
    catch (...) {
        // Catch Block 1
        …
    }
```

But our troubles are not over. If `h()` throws, then `tg2`'s destructor will be invoked before the call to `tg2.sync()`. If the destructor were to call `sync()` and an exception were propagated from `g()`, then the destructor would need to discard the exception (or else terminate). Notice that the library would need to discard the exception that would have been caught in the serialization of the program, such that the *wrong exception* is caught, according to the serialization. In summary, it is very difficult to make the exception behavior of a library interface consistent with the behavior of serial programs.

5.6 Simple and Transparent Syntax

A syntax that is writable, readable, and has a straightforward meaning is critical for making parallelism accessible to the widest-possible range of programmers. The examples in the rest of this section show that what looks like a straightforward library interface can hide a number of traps that can result in hard-to-read code with unexpected semantics.

In addition to the immediate aesthetic advantages, `cilk_spawn` has the following beneficial qualities:

5.6.1 Serialization

It is no accident that the following two lines look nearly identical:

```
Abc r = cilk_spawn f(v, h(j), k);   // asynchronous call
Abr r =            f(v, h(j), k);   // synchronous call
```

These two statements have the same meaning except that the first statement *allows* parallel execution and the second does not. In fact, the Intel® Cilk™ Plus language specification uses the second construct (called the "serialization") to describe the semantics of the parallel construct. It is easy to see what the program would do if run with only one worker (CPU core). The benefits of being able to reason about a program serially before trying to understand its parallel behavior should not be understated.

It is possible to define the serialization of a `task_group::spawn` , but it is not nearly as obvious to the reader. A `task_group::spawn` of `f()` simply does not look like a call to `f()`. Even if the lambda syntax is used (with all of the perils that involves), the lambda syntax adds significant clutter.

5.6.2 Ease of Unparallelizing Code

Parallelizing code efficiently is an iterative process. Whether parallelizing a serial program or writing a parallel program from scratch, using whatever tools you have at your disposal, you choose the sites in the program that are the best candidates for parallelization. Some of these sites may yield disappointing results. Lock contention, false sharing, insufficient parallelism, or irresolvable race conditions may force you to change parallel code to serial code. You might, in fact simply be experimenting to see which performs better, the parallel or the serial version of the code. If the code is parallelized using the `cilk_sync` keyword, switching back and forth between serial and parallel code involves simply adding or removing the keyword itself.

In contrast, the task_group library interface (as well as every other library interface I've seen), requires a near-complete rewrite of a serial function call to make it asynchronous. Thus:

```
Abr r = f(v, h(j), k);    // synchronous call
```

Becomes

```
Abc r;
tg.spawn_r(r, f, v, h(j), k);    // asynchronous call
```

or

```
Abc r;
tg.spawn([&,=j,=k]{ r = f(v, h(j), k); });    // asynchronous call
```

Thus, the programmer must commit significant time to parallelizing a call, only to discover that he/she needs to undo it later.

6 Language Solutions for Library Shortcomings

As I described in the introduction, part of my charter in writing this paper is to imagine language changes that would improve the library interface but which would not be specific to parallelism. The ideas below are necessarily incomplete, as it is not clear that there would be interest in any specific ones.

6.1 Better Parameter Passing

The need to use std::ref when passing arguments by reference can be eliminated if we adopted the signature metafunction proposed in N3466, *More Perfect Forwarding*. This metafunction would give the spawn function enough information to choose pass-by-reference or pass-by-value. I do support this proposal, but it does not eliminate the extra move constructor call in pass-by-value arguments, which can be expensive if the type does not have a constant-time move constructor, nor does it make invoking lambdas asynchronously any safer.

6.2 Simpler Return-value Handling

It is perhaps possible to make a language extension that would allow task_group::spawn to obtain a reference to its return value, and delay construction of the return value just as cilk_spawn does. I imagine syntax something like this:

```
template <class F, class... Args>
typename result_of<F>::type
spawn(typename result_of<F>::type return r, F f, Args&&... args);
```

The implementation of spawn would, at some point, initialize r:

```
r.return(f(std::forward<Args>(args)...);
```

Of course, this is very inventive, and we would need to figure out what the meaning would be if `result_of<F>::type` is a reference type or `void`, but it is a general-purpose feature (in the sense that it is not specific to parallelism), as befits my charter.

6.3 Better Overload Resolution and Template Instantiation

Although I do not have a specific proposal, I believe that a linguistic solution to the problem described in section 5.3 would be beneficial for a larger class of problem than parallelism. One possible direction would be a construct similar to the `signature` metafunction proposed in N3466, but with syntax that would allow it to be used in the argument list of a function template to select a specific overload or template specialization from the possible candidates.

6.4 Constructs to Enforce Strictness

To solve this problem it would be necessary to declare a class such that instances are prevented from being passed by reference to another function, captured by reference in a lambda, or used within a nested `try` block. Yet, to be useful, the semantics of such a construct would need to be defined very carefully, so that, for example, member functions could still be called. Though not specific to parallelism, I do not know of another use case for such a language feature.

6.5 Manipulation of Exceptions

Some of the difficulties that the `task_group` idea has with exceptions are related to the strictness problem, particularly the use of a `task_group` declared outside of a `try` block being used within the `try_block`. As previously stated, any solution to that part of the problem would be of dubious general value.

With respect to the difficulties involved with throwing an exception from a destructor, it would perhaps be helpful if there were library functions that provided more information about the current exception state and ability to manipulate it by, for example, replacing the current exception by a different one, or chaining them together. Again, I don't have a specific proposal in mind.

To get the clean syntax originally envisioned for `task_group`, where explicit calls to `sync()` were not necessary, there may be no choice but to throw from the `task_group` destructor, using library mechanisms to avoid throwing when another exception is in flight.

6.6 User-defined Control Constructs for Improved Syntax

The language changes described in sections 6.1, 6.2, and 6.3 would allow some improvements to the library syntax, but the ideal way to get the desired syntax would be to add language features for user-defined control constructs. For example, if one could define a function-modifier template that could be instantiated with an unevaluated expression and which could deduce from the expression the identity of the function or functor being invoked, the types of arguments being passed, and the address of the return value for the invocation. Then it would be possible to make `task_group::spawn` look very similar to `cilk_spawn`. Such a change would be very ambitious, but would also add a new and exciting dimension of extensibility to C++.

7 C Compatibility

WG14 (the C standards committee) has taken up parallelism and appears moving forward on a syntax resembling that described in N3409. Since C lacks both templates and lambdas, it is almost certain that the C feature will be language-based, not library-based.

Although C compatibility is not a primary consideration with regards to adding new features to C++, when a similar feature is being considered by both WG21 and WG14, there has historically been an effort to make them as similar as possible. For example, the atomics and `alignas` proposals were harmonized between the two committees.

Harmonizing the two languages will make it more likely that they will share a common runtime library and that C code that calls C++ code will work correctly, and vice-versa. If the C standard adds a syntax similar to `cilk_spawn`, then it is likely that certain vendors will make that feature available in C++, just as some vendors have made `restrict` and variable-length arrays available in C++. One danger is that users will become dependent on these extensions and there will be pressure to adopt the C language approach in a future standard *in addition* to whatever C++ library has been adopted in the intervening years. Meanwhile, we will have lost an opportunity to influence the C language fork-join design so that it is, for example, usable in the presence of destructors, exceptions, and function objects.

8 Conclusion

My goal in proposing the fork-join language constructs in N3409 was to add parallelism to C++ in such a way that it would be accessible to a wide range of programmers. Those constructs were designed to have a simple syntax and nearly intuitive semantics so as to make the task of parallelizing code as easy as possible. They are also fully implemented in the Intel® and GNU compilers and an effort is well under way to implement them in Clang.

A simple library interface that works similar to the `cilk_spawn` construct in N3409 seems possible, at first. A parallel region is delimited by constructing a `task_group` object, a member function of `task_group` is used to invoke functions asynchronously (fork), and the destructor of `task_group` ensures that all asynchronous calls complete (join) before control can leave the parallel block. The interface appears simple and the use of RAII seems to ensure strictness.

Unfortunately, we must give up the RAII mechanism to avoid the possibility of an exception being thrown from the `task_group` destructor. Strictness is further compromised by the fact that the `task_group` is visible to called functions, especially lambda captures. Even with the `signature` trait proposed in N3466, issues remain with passing argument, returning values, and resolving overloads and function template calls.

These issues with the library approach can be described as ***the final gap*** between a good idea and a standardizable feature. As existing libraries like TBB and PPL have shown, a large percentage of real parallelization jobs do not run into these issues. A lot of code, for example, simply pretends that exceptions don't exist and ignores the issues involved with handling exceptions. In the standard, however, we cannot ignore the remaining percentage. We avoid making code ill-formed or undefined if a reasonable person would expect it to work (remember the rush to add exception guarantees to the STL in 1998?). When we distort interfaces or semantics in order to handle important, albeit rare, cases, we create *dark corners* that make a feature hard to use, and cause people to avoid it for all but the most advanced uses.

Engineers trying to write parallel software should be able to focus their attention on distributing work efficiently in parallel and avoiding races, not on circumventing dark corners of the language or library.

As we discover issues, both major and minor, that complicate a library interface, we should be asking ourselves whether a library approach should even be considered for fork-join parallelism. Few would argue that local variables, branching constructs, and exceptions should be rendered purely with library interfaces. These features are language primitives because they involve fundamental language properties such as object lifetime, control flow, and scope. Fork-join parallelism can be looked at the same way – it interacts with the same primitive concepts and has semantics that extend beyond specific call sites. Although there is no single criterion that distinguishes a potential language feature from a potential library feature, the breadth of the issues described in this paper should make one consider whether a fork-join library could ever be truly integrated into C++.

Experts in parallel programming know that it's *hard*, especially in the absence of a regular and composable parallel language. Cilk has made significant gains in making parallelism accessible by providing a truly simple syntax, enforceable strictness, and encapsulation of non-determinacy. It integrates fully with the existing language. Powerful tools exist for measuring parallelism and deterministically detecting races within a Cilk Plus program. The core syntax and concepts have been implemented for over 15 years. No existing or proposed parallelism library can claim all of that.

As stated in the introduction, this paper has not, and cannot, prove that a library-based parallelism proposal as good as `cilk_spawn` is impossible. With some cleverness, some of the obstacles can be overcome and with 2-3 of the language proposals described in Section 6, perhaps it is possible to come close enough. Yet, except for `parallel_invoke`, no general fork-join library facility has been proposed and, with the exception of `std::signature`, none of the small-to-medium language proposals in Section 6 have been specified or formally proposed. The features in N2409, by way of contrast, are fully specified and implemented, and could quickly be rendered as formal wording. Holding out for a better library interface is a highly speculative activity, one that could leave us with no parallelism solution in the standard at all.

One final note: I do not pretend to be neutral on this issue. Although I remain open to a truly effective library solution, parallel programming is hard and I do not want to compromise the proven benefits of Cilk in the service of an ostensible principle that library solutions are preferable to language solutions.

9 References

MIT *The Cilk Project Home Page*

cilkplus.org *The Intel® Cilk™ Plus home page*

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

N3466 *More Perfect Forwarding*, Mike Spertus, 2012-11-03 (See also revised paper N3579)

N3579 *A type trait for signatures*, Mike Spertus, 2013-03-15 (A revision of N3466)

N3429 *A C++ Library Solution To Parallelism*, A. Laksberg, H. Sutter, A. Robison, S. Mithani, 2012-09-21

N3406 *A proposal to add a utility class to represent optional objects (Revision 2)*, F. Cacciola and A. Krzemieński, 2012-09-20

Cilk Plus GCC *How to Download, Build, and Install Cilkplus GCC on Linux*

Cilk Plus/LLVM *Github page for Cilk Plus in Clang/LLVM*