Doc No: N3409=12-0099
Date: 2012-09-24
Author: Pablo Halpern
Intel, Corp.

pablo.g.halpern@intel.com

Strict Fork-Join Parallelism

Contents

1 Vision 2
2 Background and Motivation 2
3 A Quick Introduction to the Proposed New Features 2
4 Fork-join Execution Model 3
4.1 What is Fork-join Parallelism? 3
4.2 What is Strict Fork-Join Parallelism? 4
4.3 Local Variable Access and the Cactus Stack 5
4.4 Why do We Want Strictness? 6
5 Language vs. Library? 7
5.1 The Difference Between Language and Library Features 7
5.2 A Memory-Allocation Analogy 7
6 Current Threading Features in the Standard 8
7 Proposal 10
7.1 Syntax Summary 10
7.2 Semantics 11
8 Known Issues 14
8.1 Implementation Background 14
8.2 Thread Identity in Non-stalling Implementations 15
9 Related Proposals 16
10 Future Directions 16
11 Implementation experience 17
12 References 18
13 Acknowledgements 18

1 Vision

Parallel C++ programming should be a seamless extension of serial C++ programming.

2 Background and Motivation

At the C++ Standards meeting in February, 2012 in Kona, I presented N3361, *C++ Language Constructs for Parallel Programming*, which argued that C++ needs parallel programming constructs. To summarize briefly, the presentation described the growth of multicore (task parallel) and vector (data parallel) hardware and the need to support programming this new hardware cleanly, portably, and efficiently in C++. The Evolution Working Group (EWG) in Kona agreed that parallelism is an important thing to support and created a study group to research it further. The study group met in Bellevue, OR in May, 2012. There appeared to be enthusiasm for targeting some level of parallelism support for the next standard (also known as *C++1y*, tentatively targeted for 2017).

In this paper, I will propose new language constructs to address the central aspects of task parallelism. This paper addresses the language vs. library decision and why the proposal herein uses language extensions rather than relying on a library interface. Although this paper offers many specifics describing the semantics of the proposed constructs, it does not yet attempt to present formal wording for WP changes. Those details will be forthcoming if and when the committee agrees with the direction of this proposal. The keywords used in the syntax examples are the keywords supported by the Intel® Cilk™ Plus compiler and are intended as a straw-man proposal; actual keyword names, attributes, and/or operators can be determined later, when discussion has progressed to the point that a bicycle-shed discussion is in order.

This paper does not directly address constructs for SIMD (vector) loops and parallel loops which are proposed separately in N3418. In addition, thread-safe containers, which are very useful in parallel programming, are being proposed in N3425. Future proposals may include library components, including additional task-parallel constructs.

3 A Quick Introduction to the Proposed New Features

The following example 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;
    if (n->left)
        a = cilk_spawn tree_walk(n->left);
    if (n->right)
        b = cilk_spawn tree_walk(n->right);
    int c = f(n->value);
    cilk_sync;
    return a + b + c;
}
```

This example uses the Intel® Cilk™ Plus keywords, `cilk_spawn` and `cilk_sync`, because they have the same semantics as are being proposed in this paper. Discussion on the names of keywords (or attributes) for the C++ standard is left to a future time.

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.

When the results of the spawned functions are needed, we issue a `cilk_sync` to indicate that the next statement cannot be executed until all `cilk_spawn` expressions within this function complete. In the absence of an explicit `cilk_sync`, one is inserted automatically at the end of the function.

Assuming that the `tree_walk` function is deterministic (i.e., no determinacy races are introduced by `f()`), the meaning of the program would be unchanged if we remove the Cilk keywords. Such a program with the Cilk keywords removed is called the *serialization* of the parallel program. The ability of a programmer to easily grasp the serialization of a program is one of the core strengths of this approach. The importance of this benefit should not be underestimated – our parallelization strategy should not change C++ so that it no longer looks like C++.

I will return to the details of syntax and semantics of `cilk_spawn` and `cilk_sync` later in the paper.

4 Fork-join Execution Model

4.1 What is Fork-join Parallelism?

The term *fork-join parallelism* refers to a method of specifying parallel execution of a program whereby the program flow diverges (forks) into two or more flows that can be executed concurrently and which come back together (join) into a single flow when all of the parallel work is complete. I'll use the term *strand* to describe a serially-executed sequence of instructions that does not contain a fork point or join point. At a fork point, one strand (the

initial strand) ends and two strands (the new strands) begin. The initial strand runs in series with each of the new strands but the new strands may (but are not required to, a fact essential for a scalable implementation) run in parallel with each other. At a join point, one or more strands (the initial strands) end and one strand (the new strand) begins. The initial strands may run in parallel with one another but each of the initial strands runs in series with the new strand.

The strands in an execution of a program form a directed acyclic graph (DAG) in which fork points and join points comprise the vertices and the strands comprise the directed edges, with time defining the direction of each edge.[1] Figure 1 illustrates such a DAG:

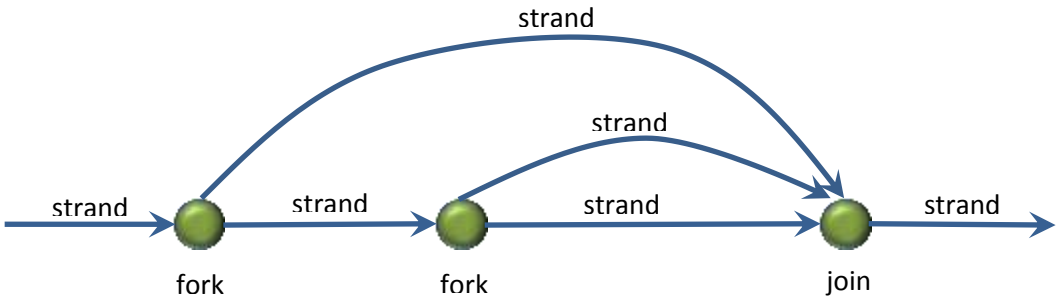

Figure 1 – A fork-join DAG

4.2 What is Strict Fork-Join Parallelism?

The property of *strictness* that is of interest for this paper is that each function has exactly one incoming strand and one outgoing strand because asynchronous function calls created by `cilk_spawn` operations, like other function calls, strictly nest within each other. Figure 2 shows each `cilk_spawn` creating an asynchronous function call nested within the calling function (also called the *parent function*).

[1] In an alternative DAG representation, sometimes seen in the literature, the strands comprise the vertices and the dependencies between the strands comprise the edges.

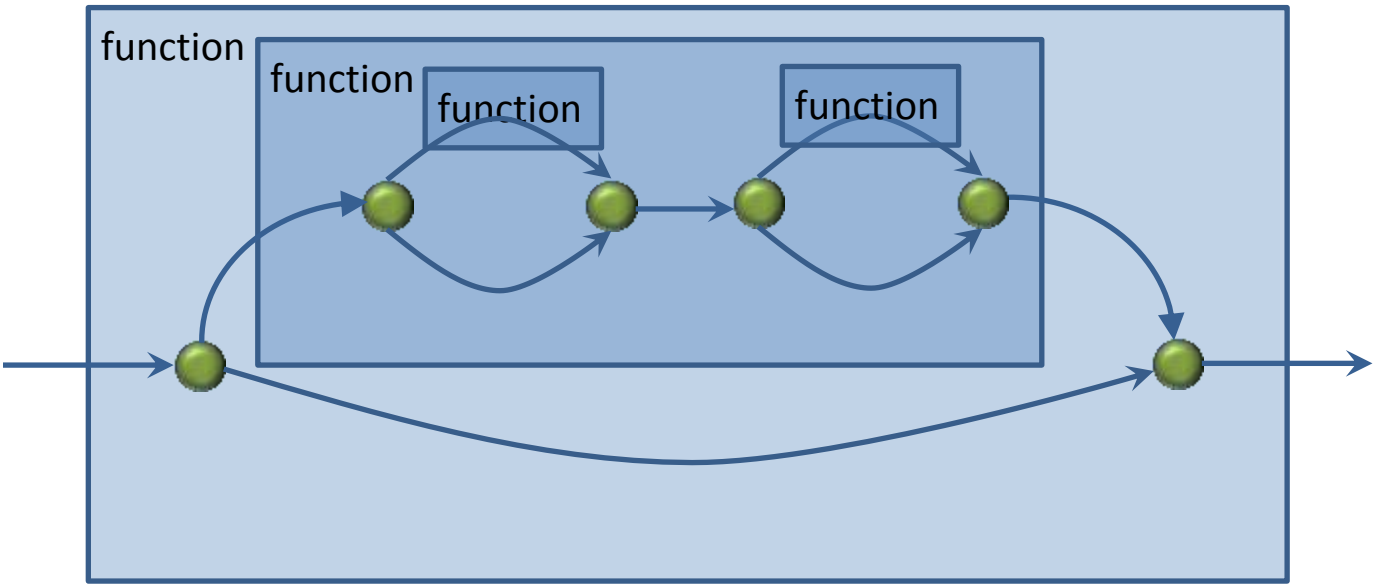

Figure 2 – Strict function-call nesting

A strict fork-join execution model has the following properties:

- A task can fork off one or more child tasks, each of which may execute in parallel with each other and with the parent task.
- A task can join (wait for) its children to complete. A task cannot wait for another task that is not its child. (The latter property is called *fully strict* in the Cilk literature.)
- A task cannot complete until all of its children complete.

The execution DAG is always a series-parallel DAG.

4.3 Local Variable Access and the Cactus Stack

The strict fork-join constructs proposed in this paper maintain the flavor of C++ execution by making access to local variables in parallel code as similar as possible to the serial case:

- Function-scope variables within the parent function can be accessed (by pointer or reference) from within the child function just as if it were called serially, since the parent is guaranteed to have a valid stack frame if the child is active.
- If a parent function spawns more than one child, these *sibling* function calls each have their own separate stack frames but share the rest of the stack with each other. Again, this mimics the behavior of a serial program.

The technology that makes this possible is called a *cactus stack*. When a child function "looks up" towards `main()`, the stack looks like a normal, linear stack, i.e., the child's local variables, it's parent's local variables, and its parent's parent's local variables are all live. Conversely, "looking down" from `main()` towards the currently-executing functions, one would see the stack diverging into different branches (like a tree or a cactus), with a leaf branch for each

function executing in parallel. Because of the tree structure of a cactus stack, it cannot occupy contiguous memory like a linear stack does (see implementation experience, below).

4.4 Why do We Want Strictness?

In general, parallel programming is hard. The job becomes easier when we avail ourselves of constructs that facilitate reasoning about the program. The benefits of strictness can be compared to that of using local variables and argument passing instead of global variables: decisions can be localized and individual parts of a program can be reasoned about without needing to understand the entire program at once. Thus, strict fork-join parallelism should be a critical tool in our parallelism strategy.

Note that **I am not proposing that strict fork-join parallelism be the only form of parallelism in C++**. Other forms of parallelism can and should be available for those situations where strict fork-join parallelism is not sufficient, just as global variables and heap-allocated objects exist for those situations where local variables are not sufficient. Both strict and less strict constructs are needed.

However, the use of strict fork-join parallelism yields the following benefits over less-structured approaches to parallelism:

- **Serial semantics**: Every program can be executed serially. A serial execution is always a legal interpretation of the parallel program. Execution on one processor exactly matches the (serial) execution of the serialization.
- **Composable**: Parallel computations can be nested arbitrarily without taxing system resources.
- **Modular**: Parallelism is encapsulated within functions so a caller does not need to know whether a function executes in parallel. Asynchronous tasks do not accidentally "leak" from functions, causing data races and other problems. Note that task leaks are usually worse than memory leaks because even a single rogue task can crash your program.
- **Faithful extension of serial C++:** This approach requires minimum reordering of control flow. The cactus stack eliminates the need create "packaged tasks".
- **Well-behaved exceptions:** An exception thrown from an asynchronous task will be caught at in the same place as it would be in the serialization of the program.
- **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.

5 Language vs. Library?

When the idea of adding parallelism to C++ was introduced, one of the first questions to be raised was whether it should be specified as a core language extension or purely as a new library. I assert that the correct answer is to do some of both. While pure library constructs for parallelism do exist (TBB is a good example), the most essential constructs should be built into the language. Conversely, other more flexible and powerful constructs can and should be implemented in a library. This proposal addresses only language support for strict fork-join parallelism. Other forms of parallelism, including pipelines, graphs, and coordination constructs can be productively specified as libraries features beyond the scope of this proposal.

5.1 The Difference Between Language and Library Features

Maximally useful fork-join parallelism constructs can take advantage of the compiler's understanding of types, variables, blocks, function boundaries, variable scope, control flow, etc.. The C++ language gives library authors the tools necessary to express user-defined abstract data types and user-defined functions, but (unfortunately) not user-defined control constructs. Libraries need objects to communicate context between parts of a control construct (e.g., between a fork and its corresponding join). Although lambdas help, they don't handle everything, and they can make code hard to read. For example: a serial for loop could be implemented as a library function, but communicating the loop control variable to the lambda that implements the loop body requires an explicit, non-obvious syntax. It is intriguing to consider language extensions that would allow for control-construct libraries, but that is not the task at hand[2].

Finally, the most efficient implementations of strict fork-join parallelism will need to take advantage of compiler support. It is possible to describe the feature using library syntax but implement it in the compiler, but then we would end up with the worst of both worlds: the implementation headaches of a language feature and the clumsy syntax of library-implemented control constructs.

5.2 A Memory-Allocation Analogy

In C++, we benefit from having core language constructs for defining automatic (local) variables with limited scope and separate library facilities for allocating longer-lived objects on the heap. As with automatic vs. heap objects, we would also benefit from having language-

[2] It is not reasonable to propose and ratify new language features that permit specification of user-defined control constructs *and* a parallelism library that depends on those new features, all within the C++1y time frame. The EWG has already agreed on the importance of adding parallelism but has not considered any proposals for adding features for user-defined control constructs.

based mechanism for highly-structured parallelism, and separate library-based facilities with less-structured semantics.

In the case of automatic object creation and destruction, the allocation of space, call to the constructor, call to the destructor, and deallocation of space is completely described by language constructs and is handled by the compiler. Conversely, library facilities `new`, `delete`, `malloc`, and `free` allow the programmer to escape the "strict" automatic allocation and deallocation semantics of the language and invoke the constituent parts under program control. As with many other library facilities, one or more objects are used to pass context between the allocation and deallocation functions; in this case the pointer to the allocated object provides the context. The heap management functions are powerful, but require more care by the programmer for correct use (to avoid memory leaks or double-deallocation). Thus, programmers try to use local variables when possible, and use heap allocation only when local variables will not suffice.

It would not make sense, in a language like C++, to support only heap-based objects, as a program in such a language would likely be efficient due to the heap-allocation overhead and suffer from memory leaks due to the difficulties in reasoning about object lifetimes. Nor have I have ever heard anyone suggest that automatic variables should be handled using a library rather than a language feature, although such a library implementation *is* possible.[3]

Strict fork-join parallelism, as proposed here, can take advantage of the execution context for simplicity and semantic rigor. Less-strict types of parallelism would require the use of "handle" objects (e.g., futures or task groups) for communicating between different parts of the program and would not need as much access to the implicit execution context. It would be a mistake to take a one-size-fits-all approach to task parallelism; both strict language constructs and less-structured library features are needed. Nevertheless, some library features could be built on top of language features and some of the implementation (e.g., a work-stealing scheduler) could be shared between language and library features.

6 Current Threading Features in the Standard

The current standard provides the building blocks for multithreaded programming: threads, mutexes, futures, and `async`. These features can be used for concurrent execution of multiple tasks in order to make use of multicore hardware. Unfortunately, these features alone do not address the challenges of programming for multicore hardware for several reasons:

[3] As a thought experiment, I designed such a library interface. See Library Implementation of Automatic Variables.

- Thread creation and destruction is a relatively heavy-weight operation. The guarantees that the standard makes with respect to the lifetime of thread-local variables imposes some cost even on code that does not rely on those guarantees.
- Threads and mutexes are not composable: If a multithreaded application calls a multithreaded library, the result could be an explosion of threads (easily the square of the number of cores in the system). Since all threads are expected to make forward progress, this oversubscription can result in lost performance due to context switches, and in the worst case, exhaustion of system resources and a crash. Likewise, if a function exits while holding a mutex, then the caller must be aware of that side-effect, breaking modularity.
- The `async` and `future` features come closest to providing support for parallelism. Unfortunately, they also make guarantees that limit the efficiency of potential implementations. In addition, extensive use of futures has been shown to cause unbounded memory use in large-scale parallel programs. Finally, C++11 futures are syntactically clumsy in situations where conditional execution is involved; the simple tree walk example above becomes very messy when expressed with `async` and `future`, though some of these failings can probably be addressed.

The core problem with the existing facilities is that they are specified in terms of *threads* and are thus *processor centric* instead of *compute centric*; a programmer must indicate which operations are to occur on which thread instead of describing the computation as a whole. The interaction between threads (e.g., through synchronization constructs) is inherently non-deterministic. Reasoning about nondeterministic constructs is much more difficult than reasoning about deterministic constructs (how do you establish confidence in a program that executes differently every time it is run?). Furthermore, thread-based constructs are too coarse-grained for effective parallelization of many workloads; they do not work well for many small, irregular tasks and they do not scale well when a program written for one system is run on a system with many more cores.

The constructs proposed in this paper *encapsulate* nondeterminacy and hide thread interactions within the scheduler. In order to obtain scalability, the user should be able to express a large amount of parallelism however the runtime system needs the flexibility to use only the fraction of this available parallelism that matches system resources. For this reason, we must *not guarantee* that tasks actually run in parallel. When parallel execution is not mandatory and serialization has the same behavior as parallel execution, the system can automatically scale up or down the amount of actual parallelism used to match system resources. This scalable quality also supports composability, since parallel libraries can call other parallel libraries without fear of exhausting system resources.

7 Proposal

What follows is a fairly detailed specification of the syntax and semantics of a fork-join parallelism feature for C++. Exact wording changes to the working paper are not being presented here, pending an expression of strong interest from the EWG as well as the needed bicycle-shed discussions on syntax.

7.1 Syntax Summary

Note: the `cilk_spawn`, `cilk_sync`, and `cilk_for` keywords below are for exposition only. The actual syntax will be determined by a later bicycle-shed discussion. The keywords could be replaced by attributes, operators, different keywords, or some combination of those.

spawning-expression:

`cilk_spawn` *function-or-functor* `(` *expression-list*$_{opt}$ `)`

sync-statement:

`cilk_sync ;`

parallel-loop:

`cilk_for (` *init-clause*$_{opt}$ `;` *condition-expr* `;` *increment-expr* `)` *statement*

The expression following the `cilk_spawn` keyword may be a normal function call, a member-function call, or the function-call (parentheses) operator of a function object (functor) or lambda expression. The function or functor following the `cilk_spawn` is called the *spawned function* and the function that contains the `cilk_spawn` expression is called the *spawning function*. Overloaded operators other than the parentheses operator may be spawned by using the function-call notation (e.g. `cilk_spawn operator+(arg1, arg2)`). A spawning expression shall appear only within an expression statement or within the initializer clause in a simple declaration. There shall be no more than one `cilk_spawn` within a full expression.

Note: The current implementations of `cilk_spawn` in the Intel and gcc compilers limit `cilk_spawn` to the following contexts:

- as the entire body of an expression statement OR
- as the entire right-hand size of an assignment expression that is the entire body of an expression statement OR
- as the entire *initializer-clause* in a simple declaration.

We know of no technical reason why the above restrictions need to exist, but there may be aesthetic reasons to maintain them.

The details of the parallel loop construct, which is built upon the basic functionality provided by the spawn and sync constructs using a scalable divide-and-conquer approach, are described in Robert Geva's paper, N3418. It is introduced here in order to make certain definitions simpler (see Task Block, below).

7.2 Semantics

7.2.1 Serialization Rule

The behavior of a deterministic parallel program is defined in terms of its *serialization*, which is the same program but with `cilk_spawn` and `cilk_sync` removed and `cilk_for` replaced by `for`.

The strands in an execution of a parallel program are ordered according to the order of execution of the equivalent code in the program's serialization. Given two strands, the *earlier* strand is defined as the strand that would execute first in the serial execution of the same program with the same inputs, even though the two strands are unordered in the actual parallel execution. Similarly, the terms *earlier, earliest*, *later*, and *latest* are used to designate strands according to their serial ordering. The terms *left*, *leftmost*, *right*, and *rightmost* are equivalent to *earlier*, *earliest*, *later*, and *latest*, respectively.

7.2.2 Task Blocks

A *task block* is a region of the program subject to special rules. Task blocks may be nested. The body of a nested task block is not part of the outer task block. Task blocks never partially overlap. The following blocks are task blocks:

- the body of a function
- the body of a `cilk_for` loop
- a `try` block

A task block does not complete until all of its children have completed. (See description of `cilk_sync`, below, for details of task block exit). A `cilk_sync` within a nested task block will synchronize with `cilk_spawn` statements only within that task block, and not with `cilk_spawn` statements in the surrounding task block. An attempt to enter or exit a task block via a `goto` statement is ill-formed. An attempt to exit the task block that comprises the body of a `cilk_for` via `break` or `return` is ill-formed.

7.2.3 `cilk_spawn`

A `cilk_spawn` expression invokes a function and suggests to the implementation that execution *may* proceed asynchronously while the spawned function executes. The call to the spawned function is called the *spawn point* and is the point at which a control flow fork is considered to have taken place. Any operations within the spawning expression that are not required by the C++ standard to be sequenced after the spawn point shall be executed before the spawn point. Specifically, the arguments to the spawned function are evaluated before the spawn point.

The strand that begins at the statement immediately following the spawning statement (in execution order) is called the *continuation* of the spawn. The scheduler may execute the child

and the continuation in parallel. A consequence of parallel execution is that the program may exhibit undefined behavior (as a result of data races) not present in the serialization.

The sequence of operations within the spawning statement that are sequenced after the spawn point comprise the *child* of the spawn. Specifically, the destructor for any temporary variables that comprise arguments to the spawned function are invoked in the child. This last point is critical, as it keeps temporaries around long enough to be passed by `const` reference without special handling by the user. The same semantic is not obtainable using a library syntax.

7.2.4 `cilk_sync`

A `cilk_sync` statement indicates that all children of the current task block must finish executing before execution may continue within the task block. The new strand coming out of the `cilk_sync` is not running in parallel with any child strands, but may still be running in parallel with parent and sibling strands. If a task block has no children at the time of a `cilk_sync`, then the `cilk_sync` has no observable effect. The compiler may elide a `cilk_sync` if it can statically determine that the `cilk_sync` will have no observable effect.

On exit from a task block, including abnormal exit due to an exception, destructors for automatic objects with scope ending at the end of the task block are invoked as usual, but the block does not complete until all of its children are complete. In effect, there is a `cilk_sync` as part of the tear down of the task block frame.

A consequence of the semantics described above is that the return value of a function may be initialized and the destructors of block-scoped variables may be invoked while children of the task block are still running.

Some people have argued that it would be better if there were an automatic `cilk_sync` before destructors were invoked (and possibly before the return value is initialized). Some advantages and disadvantage of the invoking destructors before waiting for child tasks (as described above and implemented in the Intel and gcc compilers) are:

- **Advantage:** If a child and parent try to acquire the same lock using (separate) `lock_guard`s, then a `cilk_sync` occurring while the parent is holding the lock can result in deadlock because the child would never succeed at acquiring the lock and the parent will never reach the point (after the `cilk_sync)` where it would release the lock. Invoking the `lock_guard` destructor without waiting for the child prevents such a deadlock.
- **Disadvantage:** Our definition of strict fork-join parallelism with a cactus stack ensures that function-scoped variables in the parent are alive within the child *except* that the current semantics allows those variables' destructors to be invoked while the child is still running (possibly causing a data race). Adding an implicit `cilk_sync` before the

destructors would ensure that the child completes before those variables go out of scope, strengthening the model of strict function nesting.

- **Disadvantage:** It is hard to explain why it is sometimes necessary to put a `cilk_sync` at the end of a function even though most of the time it has no observable effect. We would prefer to avoid this sort of difficult-to-explain subtlety.

I welcome reasonable debate about adding an implicit `cilk_sync` before destructors are invoked would be welcome, but the outcome will have only a modest effect on the useability of the constructs proposed here. Work-arounds exist to compensate for the disadvantages of either outcome: A user concerned about destructors racing with children can add an explicit `cilk_sync` at the end of the function; conversely, if an implicit `cilk_sync` were added before the destructors, a user could force destructors to run sooner by creating a nested block containing the variables at issue (as is already a common idiom with `lock_guard` variables anyway).

`7.2.5` Exceptions

There is an implicit `cilk_sync` before a `throw`, after the exception object is constructed.

If an exception escapes a spawned function, it is nondeterministic whether the continuation of the caller runs. If it *does* run, then it is not aborted; it runs until the `cilk_sync`. Exceptions do not introduce asynchronous termination into the language.

The `catch` clause of a `try` block does not execute until all children of the `try` block have completed.

If multiple spawned functions throw exceptions within a single `try` block, the earliest one according to the serialization is propagated. If a spawned child function and the parent continuation both throw exceptions, the child's exception is propagated. Discarded exceptions are destroyed.

Exception handling can introduce nondeterminacy into otherwise deterministic parallel programs. The rules described here allow the exception handling in parallel code to most closely resemble the serialization, at the cost of potentially discarding some exceptions. It would be possible, using exception pointers, to keep all of the exceptions somehow, but it is probably not worth it. We have not seen a use case where the use of exceptions was so intricate as to warrant adding such additional levels of complexity.

8 Known Issues

8.1 Implementation Background

8.1.1 Work Stealing

All implementations of the constructs described in this paper, and all expected implementations, use a work-stealing scheduler. A work-stealing scheduler uses a pool of *worker* threads, typically one per available CPU core. Each idle worker polls the other workers, typically at random, to see if there is any asynchronous work to be done. If so, it "steals" the work from the other worker and begins executing it. Among other advantages, this strategy means that the cost of load balancing is borne by workers that would otherwise be idle.

It is important to note that a worker may or may not be implemented using the same underlying OS mechanism as `std::thread`. We should avoid conflating `std::thread` with workers, as tying them too closely will limit implementation options, especially on novel hardware, and could cause confusion (both for the standards committee and for the end user) that could lead to bad design decisions.

8.1.2 Child Stealing vs. Parent Stealing Implementations

On execution of a `cilk_spawn`, the program can either queue the child function and begin executing the continuation, or it can queue the continuation and begin executing the child function. The first approach is called *child stealing* because an idle worker (the *thief)* will pull the child off the queue and execute it. The second approach is called *parent stealing* because the thief will pull the parent continuation off the queue and execute it. Parent stealing has proven bounds on stack usage whereas child stealing has no such bounds and implementations sometimes limit parallelism when stack space becomes tight. Parent stealing is the only approach that has well-defined serial semantics (i.e., single-worker execution matches the serialization). All modern implementations of languages derived from Cilk use parent stealing. TBB and PPL use child stealing.

8.1.3 Stalling vs. Non-stalling implementations

Another variation among implementations is whether a worker stalls at a sync point. There are two choices: A stalling implementation is one where a specific worker waits until all children are complete, then continues after the sync point. If the computation is unbalanced, then parallelism is lost while the worker waits. A non-stalling implementation does not designate a specific worker to continue after the sync. Whichever worker reaches the sync point last continues after the sync. The remaining workers are free to steal other work. Although most parent-stealing implementations are non-stalling and most child-stealing implementations are stalling, the two factors can be separated.

8.2 Thread Identity in Non-stalling Implementations

One potential problem with non-stalling work-stealing implementations is that the worker that continues after a `cilk_sync` is non-deterministic. This means that a function can return on a different (OS) thread than it was called on. A caller that is not expecting this change of thread could make incorrect assumptions about thread-local storage (TLS) and thread IDs.

Another problem with functions returning on a different thread is that the current definition of mutexes is bound to threads rather than to tasks. As a result, a mutex acquired before a `cilk_spawn` may not be able to be released after the `cilk_sync` if the thread has changed in-between.

8.2.1 Mutexes

Mutexes exist to handle interactions between threads and are a form of nondeterministic programming. Parallelism, by contrast, is generally about splitting up work such that mutual exclusion is unnecessary. When mutexes *are* needed in parallel code, their use needs to be disciplined so that the code does not become serialized. This usually means acquiring mutexes over very short regions of code with no parallel control. Thus, concerns about the behavior of mutexes across `cilk_spawn` and `cilk_sync` invocations should be considered a minor issue when measured against the essential qualities of performance, composability, and serial semantics.

8.2.2 TLS

Thread local storage is basically another form of global storage. Its use should be limited to carefully-considered situations where the nondeterminism introduced by their use can be hidden or managed. Reaching for TLS in a parallel-programming environment is nearly useless, since, regardless of the work-stealing implementation, one can rarely depend on TLS being stable across a parallel-control construct (e.g., a `cilk_spawn` or `cilk_sync`). About the only reliable use of TLS in a parallel context is for per-thread caches and memory pools, where correctness does not depend on knowing what thread is doing the work. Thus, we should minimize the impact of TLS on our decision-making when designing a parallel programming system for C++, especially if making TLS behave "well" would compromise parallel performance. The sooner we wean users off of TLS, the better. C++ will not become the parallel programming language of choice if we limit the ability of implementations to use the most efficient schedulers. For more information you can view my presentation for the parallelism study group, May 7-9, 2012 in Bellevue, WA. Hyperobjects, especially holders (see Future Directions, below) can be used to encapsulate TLS in a composable way.

9 Related Proposals

N3418: Vector and Parallel loops. This paper contains a more detailed description of the `cilk_for` loop as well as additional constructs to allow a program to take full advantage of the vector units within a single CPU core.

N3425: Thread-safe Containers. This document describes containers that can be used effectively in parallel and multithreaded programs.

10 Future Directions

If the EWG is interested in moving forward with the ideas in this proposal, then some future proposals will build on the concepts:

- **Hyperobjects**: These are special variables implemented using a class library that present a different "view" to each strand executing in parallel. Using views, parallel tasks can update the same hyperobject without causing a data race and without acquiring locks. A type of hyperobject called a *holder* can be used as a composable replacement for TLS. The most common type of hyperobject is a *reducer*, which automatically combines the different views using an associative (but not necessarily commutative) operation to produce a deterministic result. For example, the following serial code performs a post-order walk over a binary tree and builds a list of all nodes for which `condition(value)` returns true:

```
std::list<Node*> matches;

void walk_tree(Node* n)
{
    if (n->left) walk_tree(n->left);
    if (n->right) walk_tree(n->right);
    if (condition(n->value))
        matches.push_back(n);
}

int main()
{
    extern Node* root;
    tree_walk(root);
}
```

  The following parallel code generates the identical list (in the same order!) with minimal restructuring:

```
std::list<Node*> matches;
reducer<list_append<Node*> > matches_r;

void walk_tree(Node* n)
{
    if (n->left) cilk_spawn walk_tree(n->left);
    if (n->right) cilk_spawn walk_tree(n->right);
    if (condition(n->value))
        matches_r->push_back(n);
}

int main()
{
    extern Node* root;
    matches_r.move_in(matches);
    tree_walk(root);
    matches_r.move_out(matches);
}
```

Hyperobjects have been implemented in the Intel® Cilk™ Plus open-source runtime library. Their performance is comparable to PPL's `combinable` facility, but with the additional benefit of permitting reductions on operations that are not commutative.

- **Library API for controlling the scheduler**: Users will want the ability to get and set the number of workers in the scheduler's thread pool, as well as get the ID of the current worker.
- **Pedigrees**: Although the scheduler introduces non-determinacy into a parallel program, every part of that execution can be given a deterministic identification string. The Intel compiler and runtime library maintain this string, which we call the *pedigree* and use it for such purposes and deterministic parallel random number generation and deterministic replay of parallel programs.
- **Parallel algorithms**: The constructs proposed in this paper can be used for parallel sort, parallel find, and other parallel algorithms.
- **Other parallelism constructs:** In addition to language constructs for strict fork-join parallelism, there is plenty of room for library facilities to provide less structured parallelism such as task groups, pipelines, graphs, and distributed multiprocessing. Some of these libraries could be built on top of the constructs described in this paper.

11 Implementation experience

- Cilk, a version of C with `spawn` and `sync` constructs essentially the same as this proposal, was implemented more than 15 years ago. It has been the subject of active evolution and has benefitted from extensive user experience.
- Implemented at MIT, Cilk Arts, Intel, and in gcc.
- There have been at least 3 different implementations of a cactus stack:

    - The Cilk 5 implementation from MIT and Cilk++, formerly from Cilk Arts, allocates all frames on the heap.
    - The Intel compiler runtime library and the open-source runtime library used in gcc use a collection of contiguous stacks. This implementation is slightly less memory efficient, but preserves the existing C++ calling convention on each OS.
    - Cilk-M, implemented at MIT, uses worker-local memory-mapping to make the cactus stack look like a linear stack to each worker.
- Implementations of this proposal have low spawn overhead (5-10 times the cost of a simple function call). Changing the calling conventions could lower the cost even further; one possibility would be to use a private calling convention within a compilation unit.
- Applications with sufficient available parallelism scale linearly unless they saturate memory bandwidth.
- Applications cope gracefully with increased system loads, unlike most Open MP programs, which are design for dedicated systems with no other loads.

12 References

N3361 C++ Language Constructs for Parallel Programming

N3418 Vector and Parallel loops

N3425 Thread-safe Containers

TLS and Parallelism

Intel® Cilk™ Plus Specification

Edward Lee, *The Problem with Threads*

Bocchino et. al., *Parallel Programming Must Be Deterministic by Default*

Mohr et. al., *Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs*

Library Implementation of Automatic Variables

13 Acknowledgements

Thank you to everybody who helped with the formidable editing task: Charles Leiserson, Angelina Lee, Michael McCool, Jim Sukha, Arch Robison, and Robert Geva.