Task Region R2 | N3991

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

2014-05-23

1 Abstract

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

This proposal is a revision of N3832.

2 Changes from N3832

The main change from the previous revision of this paper is the addition of the `task_region_handle` class as an explicit means of communicating from a `task_region` to the spawn point of a child task. A change in this vein was requested at the February 2014 Issaquah meeting as a way to avoid "magic" or "out-of-band communication." It was also suggested that requiring an explicit handshake between these parts of a task region would make it easier for compilers and linkers to avoid combining disparate implementations of the library proposed herein.

An additional change is that the type of parallelism described in N3832 was "terminally strict" whereas the type of parallelism described in this paper is "fully strict" (see Strict Fork-join Parallelism, below). The change was made because terminal strictness was deemed to present implementation difficulties for little gain in expressiveness.

These changes created an issue not present in the original paper: potential violations of fully-strict fork-join parallelism. A few sentences in the formal wording and a new issue in the issues section was added to the paper in response.

Finally, `task_cancelled_exception` was renamed to `task_canceled_exception` (one el instead of two).

3 Motivation and Related Proposals

The working draft for the Parallelism TS N3960 augments the STL algorithms with the inclusion of 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 N3960 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 `task_region`, `run` and the `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 assistance, 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;

    task_region([&](task_region_handle& tr) {
        if (n->left)
            tr.run([&] { left = traverse(n->left, compute); });
        if (n->right)
            tr.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, `task_region` and `task_region_handle::run`.

The `task_region` function delineates a region in a program code potentially containing invocations of tasks spawned by the `run` member function of the `task_region_handle` 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 `task_region` are joined back to a single thread of execution at the end of the `task_region`.

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

```
void g();

void f(task_region_handle& tr)
{
    tr.run(g);     // OK, invoked from within task_region in h
}

void h()
{
    task_region(f);
}

int main()
{
    task_region_handle tr; // Error: no public constructor
    tr.run(g);              // No way to call run outside of a task_region
    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 region* that "owns" all of the tasks spawned within it. A *child* task spawned within a task region is automatically joined when the task region 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_region_handle`, 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_region_handle` 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:

```
task_region([&](auto& tr) {
```

```
    tr.run([&]{ g(tr); }); // Error, tr 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 N3960 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 `task_region_final`, the implementation of each of the functions defined herein is permitted to return on a different native thread than that from which it was invoked. See Thread Switching in the issues section for an explanation of when this matters and how surprises can be mitigated.

6.1   Header `<experimental/task_region>` synopsis

```
namespace std {
  namespace experimental {
  namespace parallel {

    class task_canceled_exception;
```

```
    class task_region_handle;

    template<typename F>
      void task_region(F&& f);
    template<typename F>
      void task_region_final(F&& f);
  }
  }

  experimental::parallel::task_region_handle*
    addressof(experimental::parallel::task_region_handle&) = delete;

}
```

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_region_handle::run` or `task_region_handle::wait` if they detect that an exception is pending within the current parallel region. See Exception Handling, below.

6.3 Class `task_region_handle`

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

    task_region_handle(_unspecified_);
    ~task_region_handle();

public:
    task_region_handle(const task_region_handle&) = delete;
    task_region_handle& operator=(const task_region_handle&) = delete;
    task_region_handle* operator&() const = delete;

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

    void wait();
};
```

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

An object of class `task_region_handle` cannot be constructed, destroyed, copied, or moved except by the implementation of the task region library. Taking the address of a `task_region_handle` object via `operator&` or `addressof` is ill formed. The result of obtaining its address by any other means is unspecified.

A `task_region_handle` is *active* if it was created by the nearest enclosing task region, where "task region" refers to an invocation of `task_region` or `task_region_final` 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 region and a `task_region_handle` passed to (or captured by) such code is not active within that code. Performing any operation on a `task_region_handle` that is not active results in undefined behavior.

The `task_region_handle` that is active before a specific call to the `run` member function is not active within the asynchronous function that invoked `run`. (The invoked function should not, therefore, capture the `task_region_handle` from the surrounding block.) [*Example:*

```
task_region([&](auto& tr) {
    tr.run([&]{
        tr.run([] { f(); });        // Error: tr is not active
        task_region([&](auto& tr) { // Nested task region
            tr.run(f);              // OK: inner tr is active
            ...
        });
    });
    ...
});
```

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

6.3.1 `task_region_handle` member function template `run`

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

Requires: F shall be `MoveConstructible`. The expression, `(void) f()`, shall be well-formed.

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

Effects: Causes the expression `f()` to be invoked asynchronously at an unspecified time prior to completion of the next invocation of `wait` or completion of the nearest enclosing task region (i.e., the `task_region` or `task_region_final` that created this `task_region_handle`).

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

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_region_handle` member function `wait`

```
void wait();
```

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

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

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

Postcondition: All tasks spawned by the nearest enclosing task region have finished.

[*Example:*

```
task_region([&](auto& tr) {
    tr.run([&]{ process(a, w, x); }); // Process a[w] through a[x]
    if (y < x) tr.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 `task_region` and `task_region_final`

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

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

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

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

Postcondition: All tasks spawned from `f` have finished execution. `task_region_final` always returns on the same native thread as that on which it was invoked. (See Thread Switching in the Issues section.)

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

7   Exception Handling

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

When an exception is thrown from the user-provided callable object passed to `task_region` or `task_region_final`, it is added to the exception list for that task region. Similarly, when an exception is thrown from the user-provided function object passed into `task_region_handle::run`, the exception object is added to the exception list associated with the nearest enclosing task region. 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_region_handle::run` or `task_region_handle::wait`, as described above.

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

When a task region 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 region.

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

The `exception_list` class is described in N3960 and is defined as follows:

```
class exception_list : public exception
{
public:
    typedef exception_ptr                                    value_type;
    typedef const value_type&                                reference;
    typedef const value_type&                                const_reference;
    typedef _implementation-defined_                         const_iterator;
    typedef const_iterator                                   iterator;
    typedef typename iterator_traits<const_iterator>::difference_type
                                                             difference_type;
    typedef size_t                                           size_type;

    size_t size() const noexcept;
    const_iterator begin() const noexcept;
    const_iterator end() const noexcept;
private:
    // ...
};
```

8   Scheduling Strategies

A possible implementation of the `task_region_handle::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 `task_region` or `task_region_final`). 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   Issues

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 along with some discussion of how they might be addressed.

9.1 Thread Switching

One of the properties of continuation stealing and greedy scheduling is that a `task_region` or `run` or `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 is 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.

In this proposal, the `task_region_final` 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 `task_region_final` requires stalling at a join point, it can potentially reduce parallel speed-up. For this reason, our advice to users would be to use `task_region` except in circumstances where thread identity is important. In implementations that do not support greedy scheduling, the behavior of `task_region_final` would probably be identical to `task_region`.

We have 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 involves 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 `task_region_final` explicitly to avoid an error:

```
void f() thread_switching;

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

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

Other approaches are also being 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.

Final resolution of issues related to thread-switching may need to wait until a thorough discussion of "execution agents."

9.2   Returning with Unjoined Children

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 stack-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.

This proposal, unlike the previous revision, requires that a `task_region_handle` be available in order to spawn a child task. It can be argued that the presence of a `task_region_handle&` 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_region_handle` to a called function except to allow exactly this usage.

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_region_handle`, it seems that this approach is unnecessary and is not discussed further in this revision.

9.3 Violating Structured Parallelism

The `task_region_handle` 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 `task_region` 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_region_handle` solves a number of other problems, including that of returning with unjoined children (above). However, `task_region_handle` also exposes a name that can be abused to violate structured parallelism. For example:

```
task_region([&](auto& tr1) {
    tr1.run(f);
    task_region([&](auto& tr2) {
        tr2.run(g);
        tr1.run(h);  // Using tr1 violates strict fork-join rules
        tr2.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, in order to support scheduling children other than from the inner-most task region 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 a an implementation that integrates the parallelism library with the compiler.

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

10 Moving Forward with Unresolved Issues

The authors continue to research a number of alternatives for addressing the issues described above, both with and without language support, and we seek committee guidance in the form of feedback on the ideas we've presented as well as additional ideas. We believe, however, that working through these issues will have only a modest impact on the library interfaces described in this paper and that these interfaces should therefore be used as the basis for adding strict fork-join library constructs, either to the parallelism TS or to a new TS.

11 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

N3960 *Working Draft, Technical Specification for C++ Extensions for Parallelism*, J. Hoberock (editor), 2014-02-28

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