Task Region | N3832

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

2014-1-12

1 Abstract

This paper introduces C++ library functions `task_region`, `task_run` and `task_wait` that enable developers to write expressive and portable parallel code.

This proposal subsumes and improves the previous proposal, N3711.

2 Motivation and Related Proposals

The Parallel STL proposal N3724 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 N3724 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.

The `task_region`, `task_run` and the `task_wait` methods 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. The paper also enables approaches to fork-join parallelism that are not limited to library solutions, by proposing a small addition to the C++ language.

The 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 the exception safety.

This proposal adopts a simpler syntax that eschews a named object in favor of the two static functions. It improves N3711 in the following ways:

- The exception handling model is simplified. Strict fork-join parallelism is
- now enforced at compile time. The scheduling approach is no longer limited to
- child stealing.

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

3 Introduction

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

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

    task_region([&] {
        if (n->left)
            task_run([&] { left = traverse(n->left, f); });
        if (n->right)
            task_run([&] { right = traverse(n->right, f); });
    });

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

The example above demonstrates the use of the two proposed functions.

The `task_region` function delineates a region in a program code potentially containing invocations of tasks spawned by the `task_run` function.

The `task_run` function spawns a *task*, a unit of work that is allowed to execute in parallel with respect to the caller.

4 Interface

The proposed interface is as follows.

4.1 task_region

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

Effects: Applies the user-provided function object.

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

Postcondition: All tasks spawned from `f` have finished execution.

Notes: It is expected (but not mandated) that the user-provided function object can (directly or indirectly) call `task_run`. The function is declared with the *thread-switching-specification*, as described in Scheduling Strategies.

4.2 task_run

```
template<typename F>
void task_run(F && f) placeholder noexcept;
```

Requires: Invocation of the function must be inside the `task_region` context.

Effects: Starts the user-provided function, potentially on another thread.

Notes: The function may or not return before the user-provided function completes. The function is declared with the *thread-switching-specification*, as described in Scheduling Strategies.

The function takes a user-provided function object and starts it asynchronously – i.e. it may return before the execution of `f` completes.

The function `task_run` can only be invoked (directly or indirectly) from a user-provided function passed to `task_region`, otherwise the behavior is undefined:

```
void f() placeholder
{
    task_run(g);
}

void h() placeholder
{
    task_region(f);
}

int main()
{
    task_run(g);     // the behavior is undefined
    return 0;
}
```

It is allowed to invoke `task_run` directly or indirectly from a function object passed to `task_run`:

```
task_region([] {
    task_run([] {
        task_run([] {
            f();
        });
    });
});
```

The nested task started in such manner is guaranteed to finish with rest of the tasks started in the closest enclosing `task_group` [*Note:* This provision allows an implementation to join such tasks earlier, for example at the end of the enclosing `task_run`. – *end note.*]

4.3 task_wait

```
void task_wait() placeholder;
```

Effects: Blocks until the tasks spawned by the closest enclosing `task_region` have finished.

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

Postcondition: All tasks spawned by the closest enclosing `task_region` have finished.

Notes: `task_wait` has no effect when it is invoked outside of a `task_region`.

5 Exception Handling

Every `task_region` has an exception list associated with it. When the `task_region` starts, the exception list associated with it is empty.

When an exception is thrown during the execution of the user-provided function object passed to `task_region`, it is added to the exception list for that `task_region`.

Similarly, when an exception happens during the execution of the user-provided function object passed into `task_run`, the exception object is added to the exception list associated with the closest enclosing `task_region`.

When `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 was first introduced in N3724 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 size_t size_type;
    typedef vector<exception_ptr>::iterator iterator;
    typedef vector<exception_ptr>::const_iterator const_iterator;

    exception_list(vector<exception_ptr> exceptions);

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

6 Scheduling Strategies

A possible implementation of the `task_run` is to spawn individual tasks and immediately return to the caller. These *child* tasks are then executed (or *stolen*) by a scheduler based on the availability of hardware resources and other factors. The 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 function object passed to the `task_region`). This approach to scheduling is known as the *child stealing*.

Other approaches to scheduling exist. In the approach pioneered by Cilk, the parent thread proceeds executing the task at the spawn point. The execution of the rest of the function – i.e. the *continuation* – is left to a scheduler. This approach to scheduling is known as the *continuation stealing* (or *parent stealing*).

Both approaches have advantages and disadvantages. It has been shown that the continuation stealing approach provides lower asymptotic space guarantees in addition to other benefits.

It is the intent of the proposal to enable both scheduling approaches.

7 Thread Switching

One of the properties of the continuation stealing is that a part of the user function may execute on a thread different from the one that invoked the `task_run` or the `task_region` methods. This phenomenon is not allowed by C++ today. For this reason, this behavior can be surprising to programmers and break programs that rely on the thread remaining the same throughout the execution of the function (for example, in programs accessing GUI objects, mutexes, thread-local storage and thread id).

Additionally, programmers writing structured parallel code need to be put on notice when a function invoked in their program spawns parallel tasks and returns before joining them.

We propose a new keyword, the `placholder` to be applied on the declaration of functions that are allowed to return on a different thread and with unjoined parallel tasks.

A function declaration specifies whether it can return on another thread or with unjoined tasks by using a *thread-switching-specification* as a suffix of its declarator:

```
void f() placeholder;
```

7.1 Semantics - Alternative A

When a function declared with the *thread-switching-specification* is directly invoked in a function declared without the *thread-switching-specification*, the compiler injects a synchronization point that enables the caller to proceed on the original thread:

```
void f() placeholder;

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;
}
```

7.2 Semantics - Alternative B

A function declared with the *thread-switching-specification* can only be directly invoked in functions declared with the *thread-switching- specification*:

```
void f() placeholder;

void g() placeholder {
    f();                // OK
}

void h() {
    f();                // ill-formed
}
```

A lambda expression that directly calls a function with the *thread-switching-specification* is considered to inherit the *thread-switching-specification*:

```
void f() placeholder;
auto l1 = [] placeholder { f(); };  // OK
auto l2 = [] { f(); };              // OK
```

This makes it possible for a lambda expression passed into `task_region` to not have an explicit *thread-switching-specification* while still invoking a function with the *thread-switching-specification*:

```
void f() placeholder;
task_region([] {
    f();            // OK
});
```

The programmer is not required to decorate the entire call chain of the program starting from the entry point to use the parallelism constructs introduced in this proposal. We therefore introduce the function `task_region_final`, declared without the *thread-switching-specification*, which acts as `task_region` but blocks until the calling thread becomes available.

In implementations that do not support continuation stealing, the behavior of `task_region_final` is identical to `task_region`.

8 Open Issues

- Pick between Alternative A and B for Thread Switching semantics
- rename `placeholder`
- revisit the term *thread-switching-specification*
- The order of the exceptions in the `exception_list` object is unspecified. Is this fine?
- consider renaming task_region, task_run to parallel_region, parallel_spawn.