Document number: N3711
Date: 2013-8-15
Reply-to: Artur Laksberg <arturl@microsoft.com>,
Herb Sutter <hsutter@microsoft.com>

Task Groups As a Lower Level C++ Library Solution To Fork-Join Parallelism

---

Abstract

This paper introduces the concept of the *task_group*, a C++ library that enables developers to write expressive and portable parallel code.

Motivation and Related Proposals

The separate Parallel STL proposal [2] proposes to augment the STL algorithms with the inclusion of parallel execution policies. Programmers use these as a basis to write many additional high-level algorithms that can be implemented in terms of the provided parallel algorithms. However, the scope of [2] does not include lower-level mechanisms to express arbitrary fork-join parallelism.

A language extension proposal in [3] proposes lower-level parallelism language constructs by adding several new keywords such as *cilk_spawn* and *cilk_sync*. Several members of the Committee have expressed interest in seeing how the functionality of *cilk_spawn* and *cilk_sync* can be expressed in a library solution.

Commercial C++ libraries already offer similar functionality as a library-only solution. In particular, since 2009 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 and have been well received by customers.

The *task_group* concept proposed in this document is based on the common subset of the PPL and the TBB libraries, which also use *task_group* internally to implement many of their own parallel algorithms. This proposal complements the high-level Parallel STL algorithms proposal [2] by enabling arbitrary fork-join parallelism, including arbitrary additional higher-level parallelism algorithms, to be built in a natural and portable way.

Together with [2], we believe this offers a viable alternative to a language-based proposal for low-level fork-joined parallelism with competitive (or in some cases better) usability, generality, and performance.

Introduction

The interface of the *task_group* class is as follows:

```
class task_group
{
public:
    static const auto ignore_exceptions = implementation-defined;

    template<typename ExceptionHandler>
    task_group( ExceptionHandler&& handler );

    ~task_group() nothrow;

    task_group(const task_group&) = delete;
    task_group& operator=(const task_group&) = delete;

    template<typename Function, typename... Args>
    void run(Function&& func, Args&&...args);
```

```
    template<typename Function, typename... Args>
    void run_and_wait(Function&& func, Args&&...args);

    void wait() nothrow;
};
```

The constructor takes an *ExceptionHandler* parameter that can be either the *ignore_exceptions* value or a callback that accepts an *exception_list* (defined in [2]).

The destructor of the *task_group* calls *wait* joining all the unfinished tasks. If any tasks have thrown exceptions, the destructor invokes the exception handler passing the *exception_list* of all thrown exceptions, and catches and ignores any exceptions thrown from the handler. (See Exception Handling, below.)

The *run* method takes a callable object *func* with an optional list of arguments *args* and spawns a task that evaluates *func(args)*. The method throws an exception if there are insufficient resources to allocate the task. Otherwise it returns immediately, allowing the evaluation to be performed asynchronously on a different thread or later on the calling thread.

The *run_and_wait* method is conceptually equivalent to separate calls to *run* and *wait*, but can be more efficient, because it executes the functor on the calling thread (the process known as inlining, see below).

The *run_and_wait* is often used as the last task executed before final join on the *task_group*. For example, the *parallel_invoke* function can be implemented as follows[1]:

```
template <typename Func1, typename Func2>
void parallel_invoke(const Func1& f1, const Func2& f2)
{
    task_group tg(/* ... */);
    tg.run(f1);
    tg.run_and_wait(f2);
}
```

Here the *task_group* runs *f1* and *f2* potentially in parallel; exceptions from both *f1* or *f2* are handled by the *task_groups*' exception handler (omitted for brevity in this example).

The *wait* method blocks until all the tasks spawned by the *run* method have completed. A task is considered completed if it runs to completion normally or throws an exception.

Inlining

Inlining (also called "waiter-assist") is an optimization that allows *wait* (including when called by the *task_group* destructor) to execute one or more unfinished tasks on the calling thread (i.e. "inlined"). This capability is essential for divide-and-conquer scenarios that helps to prevent thread explosion.

[1] A more realistic implementation would, of course, use variadic templates

Exception Handling

There are two kinds of exceptions that must be considered: exceptions thrown by a task, and exceptions thrown by the calling thread containing a *task_group*. Consider the following code, which shows two versions of conceptually the same function, one fully sequential and one internally parallel. Assume that *g*, *h*, and *std::string* could all potentially throw.

```
void f_seq() {
    g();
    h();
    string s = "Hello"s + " world";
}

void f_par() {
    task_group tg(/*...*/);
    tg.run([]{ g(); });
    tg.run([]{ h(); });
    string s = "Hello"s + " world";
}
```

Here, *f_seq* could result in up to one exception being thrown if *g*, *h*, or a *string* operation throws. It will propagate the first and only exception encountered, if any.

Note that *f_par* could encounter two kinds of exceptions:

1. An exception from the parent context, thrown by *run* itself (if there are insufficient resources to allocate the task) or a *string* operation.
2. An exception from a task potentially on another thread, thrown from *g* or *h*.

If an exception is thrown from the parent context, the *task_group* object is destroyed as a result of stack unwinding. The destructor of the *task_group* object will wait for all the unfinished tasks to complete, invoke the *ExceptionHandler* if appropriate (see below), and propagate the exception.

If any of the tasks throws an exception, *~task_group* invokes the *ExceptionHandler* with the *exception_list* containing the *std::exception_ptr* objects that represent all the exceptions thrown by the tasks spawned by the *run* method. Individual exceptions can be obtained by iterating over the *exception_list* object:

```
task_group tg( [](const exception_list& ex) {
    for(auto e : ex) {
        try { std::rethrow_exception(e); }
        catch(const some_exception_type&) {
            ...
        }
    }
});
```

An exception thrown out of the exception handler is ignored.

Appendix A: Comparison with PPL/TBB

This proposal is informed by the experience of PPL and TBB (primarily the former) that have been part of the commercially shipped software since 2010.

The proposal differs from the PPL/TBB in the following ways:

1. The *task_group* proposed for standardization does not have the built-in support for cancellation. It has been our experience that cancellation has a non-trivial impact on performance and impedes usability. Many of the pitfalls and anti-patterns arising from cancellation are captured in [5].
   In Microsoft, we have seen better results with explicit cooperative cancellation model based on cancellation tokens.
2. In PPL and TBB, the *task_group* destructor terminates the program in the presence of unfinished tasks in the *task_group* during normal destruction, and not during stack unwinding. In the proposal, the destructor always implicitly joins.
3. In the current PPL/TBB implementation, when a task spawned by the *run* method throws an exception, that exception is associated with the *task_group* and is re-thrown by *wait*. If more than one task throws an exception, only one exception is preserved. The execution of other tasks is either cancelled or detached. This behavior is undesirable and is not proposed. The proposed semantics preserve all exceptions by aggregating them.
4. In PPL/TBB, *task_group::wait* returns a status, indicating whether the execution has completed successfully, was cancelled, or finished with an exception. Without cancellation, such status is no longer necessary, hence the method returns *void* in the proposal.
5. PPL and TBB define an additional type, *structured_task_group* that introduces some usage restrictions compared to the *task_group* in order to gain performance, as described in [6]. In this proposal, *task_group* and *structured_task_group* are coalesced into a single type with competitive performance.

Appendix B: Possible Extensions

make_future

As specified above, the *task_group::run* method returns *void*. It is sometimes useful to deal with the individual tasks spawned by the *task_group*.

In order to avoid any overhead on *task_group::run*, it is conceivable to add a new function tentatively called *make_future*.

Such function can also be written by the user as a global function:

```
template<typename Func>
auto make_future(task_group &tg, Func&& func) ->
std::future<decltype(func())>
{
    typedef decltype(func()) T;

    auto p = std::make_unique<std::promise<T>>();
    auto f = p->get_future();
    tg.run([func, p{ std::move(p) }] {
        p->set_value(func());
    });
    return f;
}
```

We leave it as an open question and seek feedback from the Committee as to whether such an API on the *task_group* is desirable.

Appendix C: Prototype Implementation

The proposal is closely based on PPL/TBB, modulo the differences described above in Appendix A. A prototype implementation that implements this proposal is available at http://aka.ms/Tiop66.

Achnowledgements

Authors thank Steve Gates (Microsoft), Niklas Gustafsson (Microsoft), Hong Hong (Microsoft) and Arch Robison (Intel) for providing valuable feedback.

References

[1] Pablo Halpern, "Considering a Fork-Join Parallelism Library", WG21 paper N3557:
http://isocpp.org/blog/2013/03/n3557-considering-a-fork-join-parallelism-library

[2] Jared Hoberock, et al, "A Parallel Algorithms Library", WG21 paper N3554:
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3554.pdf

[3] Pablo Halpern, "Strict Fork-Join Parallelism", WG21 paper N3409:
http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2012/n3409.pdf

[4] Intel ® C++ Compiler XE 13.1 User and Reference Guides, cilk_spawn:
http://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/GUID-6DFD6494-58B4-40ED-88E9-90FEAF5AF8F6.htm

[5] Messmer, B., Parallel Patterns Library, Asynchronous Agents Library, & Concurrency Runtime: Patterns and Practices, 2010.
http://www.microsoft.com/downloads/en/confirmation.aspx?displaylang=en&FamilyID=0e70b21e-3f10-4635-9af2-e2f7bddba4ae

[6] MSDN, structured_task_group specification:
http://msdn.microsoft.com/en-us/library/dd504799.aspx