Doc. No.: N4157
Date: 2014-10-02
Author: Arch D. Robison, Jared Hoberock, Artur Laksberg
Reply to: ⟨arch.robison@intel.com⟩

Relaxing Packaging Rules for Exceptions Thrown by Parallel Algorithms

Contents

1 Abstract

If an algorithm invocation throws only a single exception, then it should be allowed to propagate the singleton exception directly instead of returning it wrapped in an `exception_list`.

2 Introduction

Experience with implementing N4105 has revealed that the obvious way to implement some of the functionality does not work because of a requirement that exceptions be packaged in an `exception_list` (except for `bad_alloc`). This requirement turns out to overly burden implementers with little gain for users.

The first author discovered this issue in his own implementation, and then found out that the two other known (and independent) implementations of N4105 (by NVIDiA and Microsoft) do not implement the requirement being questioned. This paper reviews implications of the requirement, why there is little gain for users, and relaxing the rule.

To reduce distraction, this paper assumes suitable `using` declarations make explicit namespace qualifications unnecessary.

A motivating example is the following attempt to implement `none_of`:

```
template <class ExecutionPolicy, class InputIterator, class Predicate>
bool none_of(ExecutionPolicy&& exec,
             InputIterator first, InputIterator last, Predicate pred)
{
    return !any_of(exec, first,last, pred);
}
```

This implementation is concise, but violates the aforementioned requirement if the copy constructor for any of the arguments to `any_of` throw an exception.

3 Survey of Current Solutions

There are solutions for dealing with the problem. The objection is not that they are unworkable, but that they involve extra complexity with little gain for users. Furthermore, extending this exception handling convention to user code will burden them with the same awkwardness.

Implementers could avoid using `any_of` to implement `none_of`. For example, they are free to create a private `any_of_ref` that takes reference arguments and contains a `try` block that can catch and wrap exceptions. Then `any_of` and `none_of` can be implemented on top of `any_of_ref`.

But we're not done yet, because it would be nice to be able to implement `all_of` as:

```
template <class ExecutionPolicy, class InputIterator, class Predicate>
bool all_of(ExecutionPolicy&& exec,
            InputIterator first, InputIterator last, Predicate pred)
{
    return any_of_ref(first,last, not1<Predicate>(pred));
}
```

Alas this implementation does not conform to N4105 if the copy-constructor for `Predicate` throws an exception. A possible solution is to avoid using `not1`, and instead augment `any_of_ref` with a boolean template parameter that indicates whether it should negate the predicate.

Another solution on the part of implementers is to create a wrapper for capturing and packaging exceptions. For example, implementers could write something like:

```
template <class ExecutionPolicy, class InputIterator, class Predicate>
bool none_of(ExecutionPolicy&& exec,
```

```
            InputIterator first, InputIterator last, Predicate pred)
{
    try_invoke([&]{return !std::any_of(policy,first,last,pred);})
}
```

where `try_invoke` is something like:

```
template<typename F>
void try_invoke( F f ) {
    try {
        f();
    } catch(...) {
        ...wrap current exception in an exception_list...
    }
}
```

This solution meets the requirements of N4105. But suppose `f()` throws an `exception_list`? Should `try_invoke` wrap it in another `exception_list`, or not? Either choice is correct, but either seems like gratuitous additional wrapping or ad-hoc flattening.

The point is not that the problem is unsolvable, merely that it creates a composition problem that seems burdensome, and as the next section shows, gains no simplicity for users. Furthermore, this composition burden extends to users who want to write their own algorithms and follow N4105's rule for reporting exceptions.

4 Does the Current Rule Really Simplify Exception Handling?

Let's look back at the motivations for requiring that all exceptions be packaged in an `exception_list`:

- The ability to report multiple exceptions.
- The ability to easily capture and decode that report.

The second point is the point of debate. At first glance, requiring all exceptions be packaged in an `exception_list` would seem to simplify matters by providing a uniform form. However, the form is intrinsically recursive: some of the exceptions therein may themselves be `exception_list` wrappers around more exceptions. The only way to inspect all exceptions is a recursive walk. Here is an example of such a walk that looks for `range_error` exceptions:

```
1  void walk1(const exception_list& x) {
2      for (auto e: x)
3          try {
4              rethrow_exception(e);
5          } catch (const range_error& r) {
6              cout << "found a range error\n";
7          } catch (const exception_list& y) {
8              walk1 (y);
9          }
10 }
11
12 bool example1(Iter first, Iter last, bool(*p)(const Foo&)) {
13     try {
14         return none_of(par, first, last, p);
15     } catch (const exception_list& x) {
16         walk1(x);
17     }
18 }
```

Listing 1: Processing catch(exception_list)

Now suppose that a parallel algorithm were allowed to throw an unwrapped exception if only one exception ocurred. A recursive walk is still required, but is fundamentally no more complicated than Listing 1:

```
1  void walk2(const exception_ptr& e) {
2      try {
3          rethrow_exception(e);
4      } catch (const range_error& r) {
5          cout << "found a range error\n";
6      } catch (const exception_list& y) {
7          for (auto d: y)
8              walk2 (d);
9      }
10 }
11
12 bool example2(Iter first, Iter last, bool(*p)(const Foo&)) {
13     try {
14         return none_of(par, first, last, p);
15     } catch (...) {
16         walk2(current_exception());
17     }
18 }
```

Listing 2: Processing catch(...)

The difference is that it starts with an `exception_ptr` instead of an `exception_list`. Other than that, it's essentially the same logic rotated.

5 Proposal

Any future revision of N4105 should relax Section 3.1 paragraph 2 to permit an implementation to throw an exception that is not an `exception_list` if only one invocation of an element access function throws an exception. Here is proposed rewording:

> If the execution policy object is of type `sequential_execution_policy` or `parallel_execution_policy`, the execution of the algorithm terminates with an ~~`exception_list`~~ exception. The exception shall be an `exception_list` containing all ~~All~~ uncaught exceptions thrown during the invocations of element access functions, or optionally the uncaught exception if there was only one ~~shall be contained in the `exception_list`~~.
>
> [*Note:* For example, the number of invocations of the user-provided function object in `for_each` is unspecified. When `for_each` is executed sequentially, if an `exception_list` is thrown, it will contain a single exception object. ~~only one exception will be contained in the `exception_list` object.~~ *end note*]