Document number: P0797R0
Date: 20171016 (pre-Albuquerque)
Project: Programming Language C++, WG21, SG1,SG14, LEWG, LWG
Authors: Matti Rintala, Michael Wong, Carter Edwards, Patrice Roy, Gordon Brown, ...
Email: matti.rintala@tut.fi
Reply to: Matti Rintala

Exception Handling in Parallel STL Algorithms

The problem

In concurrent execution it is possible that several parallel executions throw exceptions asynchronously. If these exception end up in the same thread of execution, the situation is problematic, since C++ allows only one exception to propagate at any time.

C++17 added parallel versions of STL algorithms, which may perform the algorithm concurrently in several threads (and even vectorize the algorithm). Original sequential STL algorithms allow exceptions to be used to signal failure (disappointment) to complete the algorithm. These exceptions may be thrown from iterator operations, invoked operators (like assignment, comparison, etc.), or from functions provided by the programmer (predicates, etc.). In sequential STL throwing an exception is the only way to abandon the execution of the algorithm (in addition to successful completion).

This paper concentrates on handling multiple disappointments arising from parallel STL algorithms. In parallel algorithms all concurrently running executions have finished when the algorithm returns (either normally or by throwing an exception). Additionally, possible multiple disappointments during the algorithm execution arise from the same set of operations, possibly making it easier for the programmer to define how they should be handled. Multiple disappointment handling in other contexts (asynchronous execution, disappointments embedded in futures, etc.) may be more complex, and is beyond the scope of this paper (for a more general discussion on multiple exception handling, see [1], for example.)

The possibility for multiple exceptions has to be dealt with, since only one exception can propagate out of the parallel STL algorithm. The original Parallelism TS allowed for an exception_list of exception_ptr, effectively a vector of exception_ptrs. This was shown to be problematic in P0394 for both the consumer who would have trouble disambiguating useful information, and from the producer in implementing such a complex system. It was discovered that of all the parallel STL implementations, only Codeplay's SYCL had in fact implemented the original exception system. At the SG1 meeting in Oulu, the group decided that lacking a better replacement, it would be best to simply reduce it to terminate with no unwinding. A further amendment also binded the exception to the execution policy, instead of binding to the algorithm. This was deemed to enable future exception policy systems. Other methods were discussed, including having dual parallel algorithms (ones that throw exceptions, and a nothrow version) but that was deemed by many to be unacceptable. As a result, in current ratified C++17 any exception in parallel STL algorithms causes std::terminate() to be

called. This was regarded as a "safe strict choice" when there was no time to come up with a better solution.

Parallel STL algorithms do not guarantee how much parallelism they use, so it may be non-deterministic which container elements are accessed and how many times predicates etc. are been called. This applies both to successful execution and execution ending in disappointment(s). Since the number of concurrently arisen disappointments is unknown and depends also on how much parallelism the algorithm execution uses (possibly hundreds of executions in GPUs in future), memory management for unknown number of exception objects is at least somewhat problematic. Dynamic memory allocation during exception handling is risky, especially with so many possible exceptions objects.

The non-determinism in the amount of parallelism used raises additional problems. On one hand, it would be useful to stop executing the algorithm as early as possible to avoid wasting time and CPU resources. On the other hand, if some possible disappointments are an indication of a more severe problem than others, it would be useful select the one disappointment that best represents the situation. However, this requires executing the algorithm as far as possible, because otherwise all possible disappointments are not detected.

To further complicate things, continuing the algorithm after disappointments may be problematic, as these disappointments may have caused some invariants of the algorithm to no longer hold. For example, if a comparison of some elements in std::sort fails, continuing the algorithm (even to gather further disappointments) is problematic, since the correct ordering of the elements can no longer be determined.

Note: In this paper, the term "disappointment" is used to represent a situation where execution of a parallel STL algorithm is not successful. This may be caused by exceptions or some other methods for signalling unsuccessful execution of element access functions (proposed std::expected<>, etc.). In most places the paper concentrates on exceptions, since that is the only disappointment mechanism supported by STL algorithms at the moment. However, the section on Future directions discusses heterogeneous (GPU-based) parallelism, where exception handling may be difficult to implement.

The State of the Art

C++17 adds parallel versions of most STL algorithms, where the given execution policy determines how the algorithm is allowed to parallelize its operation. Currently any exception from a parallel STL algorithm causes std::terminate() to be called. The choice to call std::terminate() when encountering exceptions in a concurrent context is used elsewhere in C++ as well. If an exception attempts to escape an execution inside a std::thread, terminate() is called. And of course, in a sequential context terminate() is called if a destructor throws an exception during stack unwinding (causing multiple simultaneous exceptions). When concurrency is achieved through std::async(), resulting exceptions are embedded in the future returned from the async() call. This does not cause terminate() to be called in any situation, but exceptions may end up being ignored if the future is destroyed without its wait() having been called.

Having multiple exceptions in one place causes a need to somehow store them together for analysis (and possibly propagation inside a single exception). Current C++ provides no such mechanisms. std::nested_exception allows a single exception to be embedded inside another exception, but that is not suitable for multiple exceptions (and nested_exception only allows rethrowing the nested exception, not analyzing its type etc.).

In OpenMP with its fork-join architecture, the rule is that if an exception escapes a parallel region, the OpenMP system will terminate and forgo unwinding. Exceptions caught within the parallel region is ok, and can even be rethrown as long as they do not escape. Codeplay's SYCL implements the original exception system of the Parallelism TS (providing an iterable list of std::exception_ptr objects).

Proposed Solution and discussion

The proposed solution consists of a disappointment_buffer class, which can contain multiple exception objects (or other disappointment types such as error_code, expected, or outcome). The maximum number of disappointments inside a disappointment buffer can be unlimited (in which case the buffer allocates more memory on the fly as disappointments are added) or set to a fixed upper limit when the buffer object is created (in which case enough memory for the maximum number of disappointments is allocated during construction of the buffer)

We propose an API as an initial strawman to discuss this idea, with every intention of changing it according to the direction of the discussion.

```
template<typename disappointment_type>
class disappointment_buffer
{
 disappointment_buffer(size_t maximum_size, size_t
maximum_disappointment_size=sizeof(disappointment_type))

 // iterators, * provides reference to elems

 size_t size() const noexcept; // Current object count
 size_t missed() const noexcept; // Number of failed insertions
};
```

The disappointment buffer has a template parameter, which specifies the type of the disappointments stored in the buffer. If the template parameter is a pointer type, that type represents the base class of all exception objects that can be stored in the buffer. If the template parameter is not a pointer type, it represents some other disappointment type (like error code, etc.).

The exception base class pointer allows the programmer to specify the common base class of all buffered exceptions (std::exception, or something else). The base class parameter is used here, because C++17 does not provide necessary mechanisms to store arbitrary exception objects in the buffer's memory. There is std::exception_ptr, but that type currently provides no way to access the exception object itself, which would be necessary to analyse buffered exceptions. Additionally creation of C++17 std::exception_ptrs may require dynamic memory allocation, which can be problematic with a potentially large amount of exceptions. If necessary additions are made to std::exception_ptr support, that type would be appropriate to store and access exception objects in the buffer.

If more than the maximum number of disappointments are added to the buffer, only the first disappointments up to the maximum size fit in, the rest are discarded. However, the disappointment buffer keeps count of discarded disappointments. Discarding disappointments was chosen here, since the number of encountered disappointments is nondeterministic in parallel STL algorithms

anyway due to parallelism. Alternatively it would be possible to call std::terminate() if the buffer overflows.

The disappointment buffer object takes care of memory allocation of the disappointments it contains, and also controls their lifetime. Since the types and sizes of disappointments may differ, the maximum size of allowed disappointments is also given to the buffer during construction (a buffer with a fixed maximum number of disappointments allocates at least maximum_size * maximum_disappointment_size bytes during construction). If an exception object larger than the set maximum size, or an exception object not derived from the buffer's base exception type is added to the buffer, it is discarded just like if the buffer were already full.

In practise, it is quite common that derived exception classes in the same exception hierarchy do not add new data members or member functions (i.e., the derived classes are only used to distinguish the types of the disappointments), and their sizes are at least close to equal. Therefore allocating enough memory for objects with the maximum size does not probably waste much memory in many cases.

If the maximum size of the buffer is given as zero, then the buffer grows dynamically to contain as many disappointments as necessary. Alternatively there could be two buffer types, one with fixed amount of memory allocated during construction, and one which dynamically allocates more memory.

If exceptions are stored in the buffer (template parameter is a pointer), those exceptions are polymorphically copied to the buffer's memory. In current C++ this requires some implementation dependent compiler magic, but it would of course be more convenient if C++ allowed polymorphically copying thrown exception objects to a given memory location.

The disappointment buffer provides iterators with normal iterator operations to provide access to the contained disappointments.

In analysing encountered exceptions, there should be a way to check the types of the buffered exception objects. Currently the only way to provide access to arbitrary exception objects is to use std::exception_ptr. However, it's currently impossible to convert an exception_ptr to an actual regular pointer of any type (except by throwing the exception and catching it, which causes too much overhead), which is why the proposed solution uses regular pointers and needs an exception base class as a template parameter. There have been talks about a proposal which would add exception_ptr_cast or similar to attempt casting std::exception_ptr to actual exception pointer types.

The disappointment buffer is created by the programmer (who also control its lifetime). The disappointment buffer is used in conjunction with new execution policies which receive the disappointment buffer as a constructor parameter (alternatively the buffer could be attached to an execution context). When a parallel STL algorithm executed under that execution policy ends up with disappointments, those disappointments are inserted into the buffer. If the buffer becomes full, the rest of the disappointments are discarded (but their number is recorded in the buffer). If the buffer is configured to collection exceptions, algorithm execution is terminated and a "parallel STL exception" containing a reference to the buffer is thrown out of the algorithm. If the buffer is configured for other kinds of disappointments, the algorithm simply returns (with unspecified return value, if any).

In the proposed mechanism above, the type of the exception thrown out of the algorithm is always the same ("parallel STL exception"). In many cases it would be beneficial if the programmer could choose which exception to throw. For this, a disappointment reduction function [2] can also be passed to the execution policy as another constructor parameter. The reduction function analyzes the buffer and throws an appropriate exception based on the analysis (this is similar to how SYCL exception handlers work). That exception then propagates out of the algorithm. If the reduction function is given, when disappointments arise from the algorithm execution and they have been stored in the policy's buffer, the buffer is given to the reduction function as a parameter.

Code example of possible use:

```
class my_exception_base; // Base class for my exceptions
class my_largest_exception : public my_exception_base; // Exception class
with largest sizeof
using my_buffer = disappointment_buffer<my_exception_base*>;

my_buffer buffer(100, sizeof(my_largest_exception));

execution::par_exception my_policy1(buffer);
try
{
  sort(my_policy1, v.begin(), v.end());
  for_each(my_policy1, v.begin(), v.end(), [](auto e){ if (...) throw ...;
});
}
catch (parallel_exception const& e)
{
   // Use the buffer to analyze exceptions. Buffer accessible either
directly or through e.buffer()
}

void my_exception_reduction(my_buffer& buffer); // Analyse exceptions and
throw something
execution::par_exception my_policy2(buffer, my_exception_reduction);
try
{
  sort(my_policy2, v.begin(), v.end());
  for_each(my_policy2, v.begin(), v.end(), [](auto e){ if (...) throw ...;
});
}
catch (my_exception_base const& e)
{
  // Catch the selected/produced exception
}
```

Alternatives considered

Many sequential C++ algorithms (like find_if) only access some of the elements, so potential exceptions from later elements are not detected because those elements are never accessed. For parallel algorithms, this does not necessarily hold, because operations on elements are not performed in sequence. For simplicity, it is still best to abort the algorithm as early as possible after first encountered exception(s), after running currently active operations to completion (i.e. not start new executions), even if the algorithm could still be potentially run to completion if it was called in a sequential manner. For example, if the last element of a vector would throw, but the first element fulfills find_if's criteria, sequential find_if would complete without exceptions, but a parallel version might access the last element concurrently with the first one, triggering the exception. It would of

course be possible to make the parallel version to ignore exceptions that would have gone unnoticed in the sequential version.

The proposed solution stops executing the algorithm as early as possible when disappointments are discovered. The alternative would be to try to run the algorithm for as many elements as possible before terminating. This could find more severe or "important" disappointments, but on the other hand waste computational resources as the algorithm would end up in a disappointment anyway.

For many STL algorithms, disappointments from element access functions make it impossible to continue the algorithm (semantics of the algorithm becomes undefined). For example, if comparison in std::sort fails, the order of the elements become undefined. For some algorithms (like for_each), it would be possible to continue even if some operations fail, but probably termination as early as possible is a good choice for consistency.

In the version proposed above, the programmer is responsible for creating the disappointment buffer and taking care of its lifetime. An alternative would be for the execution policy, execution context, or the STL algorithm to create the disappointment buffer and then transfer its ownership to the thrown exception, if any (or destroy it after exception reduction, if reduction function is used). The programmer would still pass the necessary information (disappointment type, maximum buffer size and maximum size of buffer elements) to the exception policy. This alternative would make handling the disappointment buffer easier for the programmer, when it is used to store exception objects. However, for other disappointment mechanisms the situation may be trickier, because this approach would require that the mechanism is able to return the disappointment buffer as part of the return value of the algorithm. It would of course be possible to allow both versions, i.e. the programmer chooses whether to pass an already created disappointment buffer or just buffer parameters to the execution policy. However, this could create complications with buffer lifetime management when it is embedded in the exception thrown from the STL algorithm.

Currently, C++17 does not provide any means for converting std::exception_ptr to actual exception class pointers (the only way is to re-throw the exception and catch it through a reference, which causes unacceptable overhead in case of multiple exceptions). There have been suggestions to add an exception_ptr_cast or similar to create ordinary pointers from std::exception_ptr (similar to dynamic_cast). If such a cast is added, it would also be possible to make disappointment buffer to use exception_ptr as the type of pointers to exception objects, instead of making the programmer provide the base class of all possible exceptions. Using exception_ptrs would complicate the memory management however, if dynamic memory management is considered too dangerous during exception handling. Current exception_ptrs manage the lifetime and memory management of the exceptions they contain, possibly using dynamic memory allocation. With pre-allocated memory in the buffer it would be necessary to somehow create exception_ptrs where the exception object is stored in given memory address.

In the proposed solution, a single disappointment buffer type is used both for storing exceptions and other possible future disappointment types. Exceptions have to be cloned polymorphically into the buffer, with the knowledge of the maximum possible exception object size. With many other disappointment mechanisms disappointments are represented by a single type, which makes copying them into the buffer easier. An alternative solution would to make these two cases have their separate disappointment buffer types. This could become even more convenient if exception_ptr_cast is added to the language, making it unnecessary to provide the exception base class for disappointment buffers storing exceptions.

Similarly it would be possible to separate fixed-size buffers and dynamically growing buffers (requiring dynamic memory allocation during exception handling) into separate buffer types.

One alternative would also be to store disappointments as visitable variants of potential disappointments (replacing the need to use exception base class pointers). This would allow

disappointments of any type to be stored in the buffer, as long as possible types are listed in the buffer's template parameter list. However, it would make using exception hierarchies more difficult, since every possible exception class would have to be mentioned (in the proposed solution it is enough to know the maximum size of stored disappointments, not every possible type).

Future Directions: Heterogeneous parallelism

For GPU-based parallelism, supporting the exception mechanism at all is problematic. Therefore alternative methods for disappointment handling may have to be investigated for GPU-based parallelism. There are already proposals (like expected<>) to allow disappointment status to be embedded in the return value. Many operations used by STL algorithms (access through iterators, assignment, copying, comparison operators etc.) are defined to throw exceptions, with no alternative disappointment mechanisms. For programmer-provided predicates and functions, return value based disappointment mechanisms like std::expected<> are a possibility. The disappointment buffer presented in this paper could easily be adapted to contain other kinds of disappointment values as well.

Some GPU architectures (e.g., NVIDIA with NVLINK) would allow GPU to write into CPU resident error buffer. As soon as the first exception is triggered performance is no longer the predominant concern, instead reliably capturing some exception information becomes the primary concern. This approach (GPU write to CPU error buffer) could enable preservation of error information even when post-error the GPU execution state is lost.

Some concern was raised as to what memory the exception_ptr references, and the state of that memory after the parallel execution completely terminates in heterogenous GPU-based parallelism.

Straw polls

From this paper we aim to identify out of all the challenges presented here what is considered most important, and what is most desirable approach:

- Should the proposal be developed further with just existing C++17, or is it reasonable to hope for additional support in std::exception_ptr, so that it could be used as the base type in the buffer?
- Should there be one buffer type/template for all cases, or:
    - One buffer type for exceptions, one for possible other disappointment types?
    - One buffer type for pre-allocated fixed amount of memory, one for dynamically growing size?
- Should algorithms be run as far as possible to collect maximum amount of exceptions, or stop executing algorithms as early as reasonable, or let the programmer decide?
- If a pre-allocated buffer becomes full and further disappointments arise, should the rest be discarded, or should std::terminate() be called?
- When should the buffer be cleared and by whom? By the programmer, by the execution policy, or by the STL algorithm? Or should the buffer be single-use without possibility for clearing it?

Acknowledgement

Thanks for Michael Wong, Carter Edwards, Patrice Roy, Gordon Brown, and everyone on the Heterogenous C++ discussion group for discussion, ideas, and comments that were valuable in drafting this paper.

References

[1] Matti Rintala: *Techniques for Implementing Concurrent Exceptions in C++*, Doctoral dissertation, Tampere University of Technology Publication 1075, ISBN 978-952-15-2915-3, Tampere University of Technology 2012 (PDF version)

[2] Matti Rintala: Handling Multiple Concurrent Exceptions in C++ Using Futures, in Advanced Topics in Exception Handling Techniques, co-editors C. Dony, J. L. Knudsen, A. Romanovsky, A. Tripathi. LNCS 4419, 301 p., ISBN 3-540-37443-4, DOI 10.1007/11818502_4, Springer-Verlag 2006 (PDF version)