Document number: P0797R1
Date: 2018-02-12 (Jacksonville)
Project: Programming Language C++, WG21, SG1,SG14, LEWG, LWG
Authors: Matti Rintala, Michael Wong, Carter Edwards, Patrice Roy, Gordon Brown, Mark Hoemmen
Email: matti.rintala@tut.fi, michael@codeplay.com, mhoemme@sandia.gov
Reply to: Matti Rintala

Handling Concurrent Exceptions with Executors

1 Introduction

This paper describes a mechanism for handling concurrent exceptions (EH) with executors. The mechanism should also work with the parallel algorithms in the C++ Standard Template Library ("parallel STL"). This paper's first version *P0797R0: Exception Handling in Parallel Algorithms [1]*, proposed a mechanism for handling exceptions specifically in parallel STL. Based on comments from the Albuquerque meeting, this proposal has been significantly changed. It has evolved to target handling of multiple concurrent exceptions with the proposed executors [2] [3] and futures. Since executors are proposed to be used in parallel STL, this proposal solves the problem for parallel STL as well. We will discuss the mechanism for parallel STL later in this paper.

2 Revision History

2.1 2018-02-12 [P0797R1] JAX Meeting

- This version is based on feedback comments from ABQ
- Describes a future based concurrent EH handling that matches closely with executors and also handles parallel STL algorithm
- ABQ meeting feedback handling:
  - Cancelling parallel execution is a separate issue; while important, we will not discuss it here
  - Exception reduction is considered beneficial
  - We should leave the door open for non-exception disappointment mechanisms, but currently exceptions are the only "official" mechanism. So the safe way is to not specify other mechanisms (yet), but make sure they remain possible
  - Executors are going forward, and are based on futures, so preferably that's where exception handling should also be. Since parallel STL will be based on executors in the future, integrating parallel exception handling with executors will help solving the problem in parallel STL as well.
  - The case of avoiding dynamic memory allocation during disappointment handling is not attacked directly by this idea, but the door is left open for situations where that is problematic.

2.2 2017-10-15 [P0797R0] ABQ Meeting

- Initial version
- Describes a proposal for a greedy but limited memory scope EH handling mechanism

3 The problem

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

This paper concentrates on handling multiple exceptions arising from parallel executions created using executors. Twoway executors return a future which will become ready when all concurrently initiated executions have finished (either normally or by throwing an exception). Oneway executors have no way to synchronize with the end-results of the executions. The possibility for multiple exceptions has to be dealt with, since only one exception can be stored in the future returned from the executor. Multiple exception handling in other contexts (like exceptions from arbitrary asynchronous executions) may be more complex, and is beyond the scope of this paper. (For a more general discussion of multiple exception handling, see [4].)

If some executions result in exceptions, it is possible that some other executions may not yet have started. Executors do not guarantee how much parallelism they use, so it may be nondeterministic how many executions are actually initiated in parallel simultaneously. This applies both to successful executions, and to executions ending in exceptions. The number of exceptions that may arise concurrently is unknown in advance, and may depend also on how much parallelism the executor uses. A single modern CPU may support hundreds of parallel executions in hardware. This makes memory management for an unknown number of exception objects at least somewhat problematic. Dynamic memory allocation during exception handling is risky, especially with so many possible exception objects.

The nondeterminism in the amount of parallelism used raises additional problems. On one hand, it would be useful to stop invoking new executions as early as possible to avoid wasting time and CPU resources. On the other hand, if some possible exceptions are an indication of a more severe problem than others, it would be useful to select the one exception that best represents the situation. However, this would require executing all the requested executions as far as possible, because otherwise all possible exceptions are not detected.

The possibility of multiple exceptions arising from a single parallel bulk execution is problematic for the return value as well. A future can only contain one exception, so either a single most representative exception should be stored in the future, or an exception containing all received exceptions should be stored, so that the user can later analyze all the exceptions. A combination of these two might also be convenient. The mechanism proposed in this paper accounts for all these variations.

In some environments, exceptions might not be a suitable disappointment handling/signalling mechanisms for various reasons. (For example, exception handling may incur unacceptable performance penalties. It may be technically challenging on some hardware, such as GPUs.) This paper also briefly discusses how other disappointment mechanisms might be used together with executors for concurrent disappointment handling.

4 The State of the Art

The design of the current executors proposal is laid out in Executors Design Document [3], and A Unified Executors Proposal for C++ [2].

The current executor proposal [2] does not require handling for concurrent exceptions from the execution(s) of the user function, only exceptions thrown from the execution function. Both single and bulk oneway executors are defined “not [to] propagate any exception thrown by [the execution]” (1.1.6). On the other hand, “[t]he treatment of exceptions thrown by oneway submitted functions is specific to the concrete executor type” (1.1.6). This is logical, since oneway executors do not return a future through which an exception could be propagated, and their implementation does not necessarily have any return channel for users to get information back from the execution.

For both twoway and "then" versions of single executors, [2] specifies that the executor “stores the result ... or any exception thrown by [the execution], in the associated shared state of the resulting Future” (1.1.7, 1.1.8). Again, this is logical, since there is only one created concurrent execution, and thus one possible exception that could be propagated out of the execution.

Twoway bulk executors (and "then" executors) are a more interesting case, as they may end up with multiple concurrent exceptions. For them, [2] says that “once all invocations … finish execution, [the executor] stores r [(the result object)], or any exception thrown by an invocation ..., in the associated shared state of the resulting Future" (1.1.10, 1.1.11). The proposal does not define what “any exception” means, but presumably it means an arbitrary single exception chosen from the encountered exceptions.

For static_thread_pool executors, [2] specifically defines that “if the submitted function ... exits via an exception, the static_thread_pool calls std::terminate()” (1.7.3.3)

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. This is because nested_exception only allows rethrowing the nested exception, not analyzing its type.

On the other hand, the current Parallelism TS 2 draft [5] contains exception_list (6.2), which can be used for storing (and throwing) multiple exceptions. That type is used to report exceptions from a task_block (8.5). Additionally, [6] says that “exception_list is underspecified” (2.2) and that some of its features are still under consideration.

4.1 Behavior of other parallel programming models

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 may be caught and even rethrown within the parallel region, as they do not escape the parallel region. OpenMP since Version 4 also has various cancellation points (usually blocking locations such as I/O as well as a new cancellation directive) where in-progress exceptions may be checked (though there is no guarantee), so as to enable notification of other threads to begin termination [9].

Khronos’s SYCL standard is a modern C++ programming model based on OpenCL, suitable for heterogenous dispatch, and follows the C++11/14/17/20 progression [7]. Codeplay’s implementation of the SYCL language, called ComputeCpp, implements the original exception system of the Parallelism TS (providing an iterable exception_list of std::exception_ptr objects). OpenCL itself does not have an exception mechanism being based on C99 (although it has a callback on error mechanism), which means SYCL does not handle exceptions from the execution itself, however it does handle concurrent exceptions thrown from the asynchronous SYCL runtime. SYCL handles these by collecting them in an exception_list and then propagating them to a user provided function for handling or reduction.

HPX is similar to SYCL but aimed at distributed computing while following closely ISO C++ progression. It implements both the exception_list concept, but also a future based exception mechanism.

HSA is also investigating an EH mechanism as it is a layer that enables a number of high-level language implementations including C++ and OpenMP. HSA EH does not use a final snapshot of all the exceptions generated, but uses a greedy model that releases exceptions as soon as they are generated.

5 Proposed Solution and discussion

This paper proposes a set of new exception handling properties for the executors property mechanism [2]: *no_exceptions*, *single_exception*, *single_exception_reduction*, and *multiple_exceptions*. These properties control how an executor will handle concurrent exceptions, and executors can choose to support only those properties they can implement. Alike many of the other properties in the executors proposal, the properties described above are mutually exclusive with regards to each other, meaning an executor can only satisfy one at any given time.

Other executor properties also affect how exception handling properties behave. Important other properties are *directionality (oneway, twoway, then),* and *cardinality (single, bulk)*. Other properties (*blocking, continuations, future task submission, forward progress guarantees, thread mapping,* and *memory allocation*) seem (at least for now) to be independent of exception handling properties. For the purposes of exception handling, *twoway* and *then* properties behave equivalently, so if only *twoway* is mentioned below, it also includes *then*.

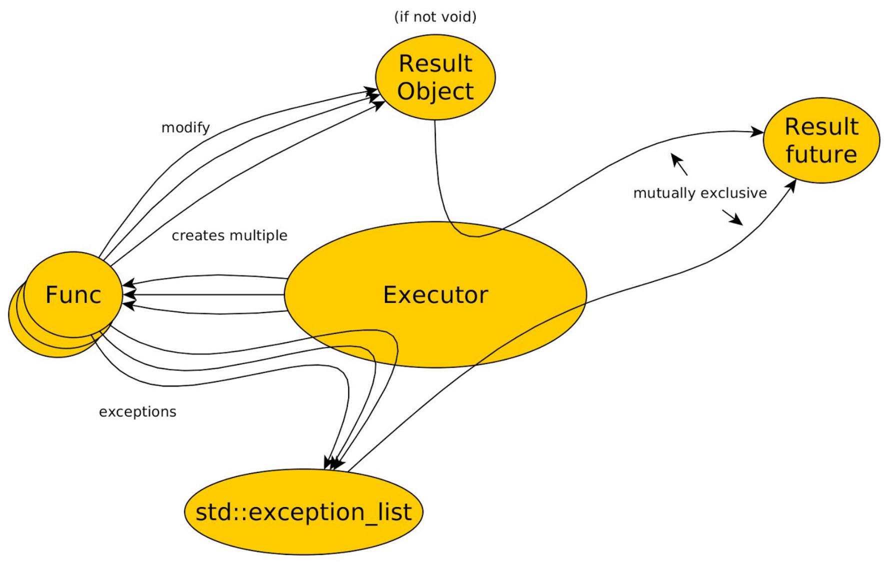
From now on in this paper, executor properties are written in *italics* to make the text clearer.

5.1 Exception Properties

The *no_exceptions* property is the most straightforward. Executors with the *no_exception* property do not support propagating exceptions to the user. (If exceptions are thrown from invoked executions, *no_exception* executors may either call std::terminate, or react to exceptions in a manner specific to the concrete executor type.) Oneway executors might typically only support the *no_exceptions* property, if they have no back channel to report exceptions to the user. The *no_exceptions* property would also suit executors for which it is difficult or too expensive to support exception handling, such as GPU-based executors. Other executors may also support this property, allowing optimizations in cases where users know exceptions will not occur.

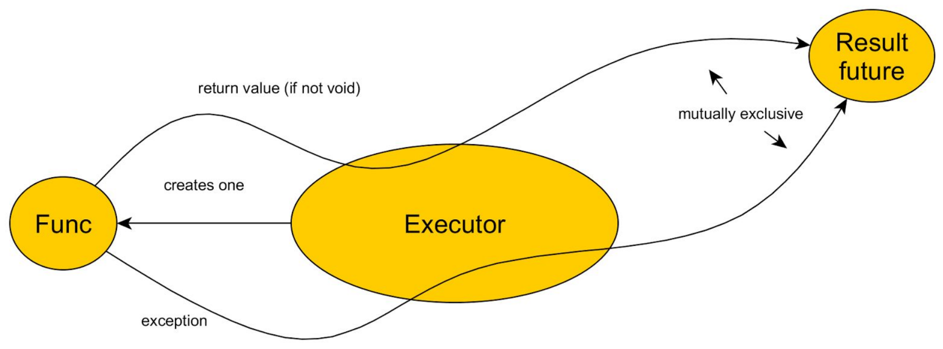
The *multiple_exceptions* property lets the executor propagate out multiple exceptions. The *bulk* executor does this by populating a std::exception_list object with encountered exceptions, and embedding this exception_list into the resulting future as an exception. This makes it impossible to catch any of those exceptions by their type, since the type of the exception in the future is just std::exception_list. Only *twoway* and *then* executors may support the *multiple_exceptions* property, since only those executors provide result futures.

Bulk, twoway, multiple_exceptions



The *single_exception* property lets executors propagate out at most one exception. For *single* executors this property is natural, since they only invoke a single concurrent execution, potentially resulting in a single exception. *Single* executors implement this property simply by embedding the resulting exception in the result future, just as the current executors proposal specifies. Similarly to *multiple_exceptions,* only *twoway* and *then* executors may support the *single_exception* property.

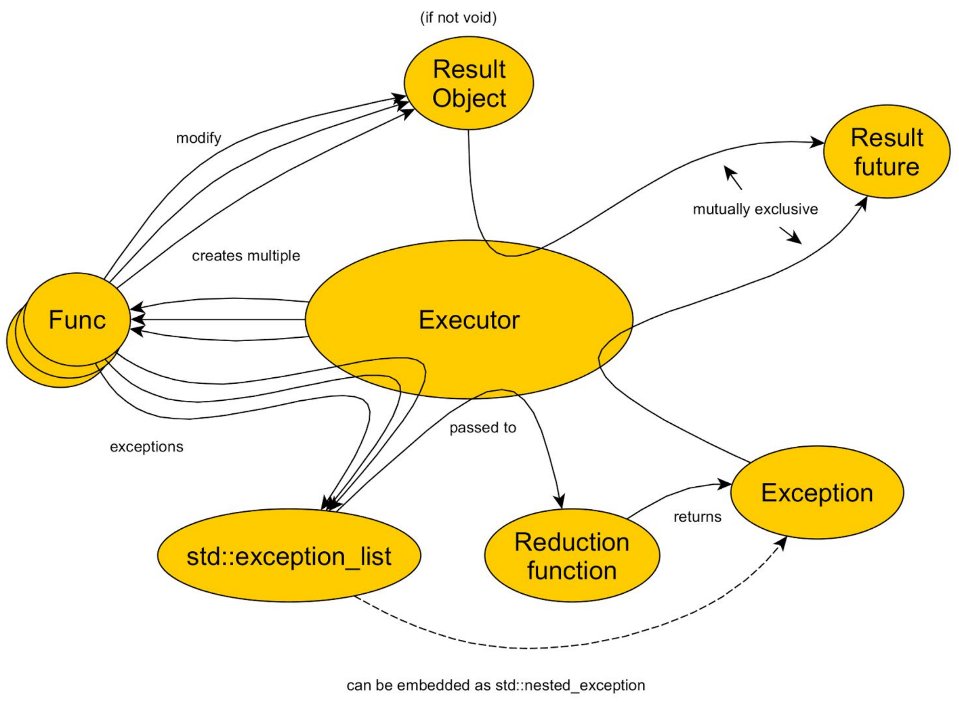
Single, twoway, single_exception



If users want at most a single exception from *bulk* executors, the resulting potentially multiple exceptions must be reduced to a single exception. The *single_exception_reduction* property achieves this. Users give a *reduction function* to the *single_exception_reduction* property as a parameter (similarly to custom allocators with the memory allocation property in the current executor proposal). The reduction function is a Callable that takes an std::exception_list as a parameter, selects or creates a suitable single exception that represents the situation however it sees fit, and returns the single exception as an std::exception_ptr. That resulting exception is then embedded in the executor's result future.

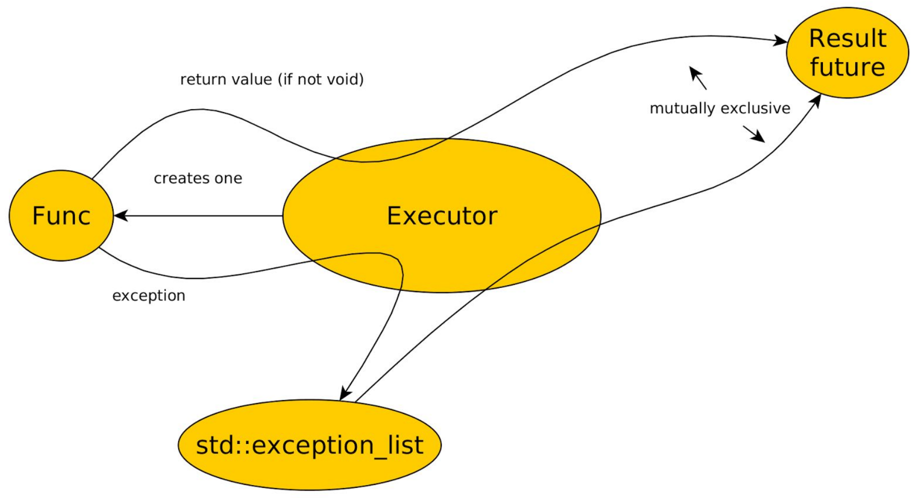
If users wish to keep the entire original exception list, they may provide a reduction function that uses std::nested_exception to embed the exception_list into the selected single exception.

Bulk, twoway, single_exception_reduction



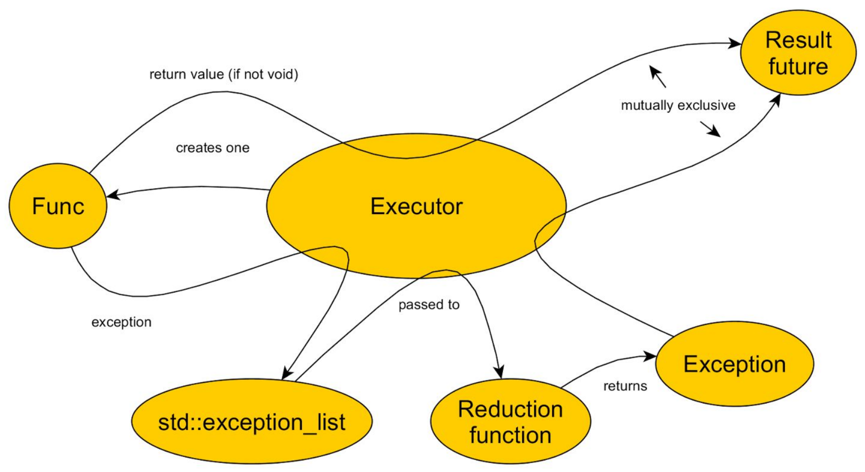
To keep the exception properties as orthogonal to other properties as possible, it is possible to use the *multiple_exceptions* property with *single* executors as well (even if only one exception can result from a single execution). In this case, a std::exception_list with a single embedded exception is embedded in the result future.

Single, twoway, multiple_exceptions



Finally, again for orthogonality, it is possible to use the *single_exception_reduction* property with *single* executors as well (even though there's only one potential exception, and thus no need for reduction). In this case the only exception is stored in a std::exception_list, passed to the provided reduction function, and the resulting exception is embedded in the result future.

Single, twoway, single_exception_reduction



Summary of constraints on which properties can go with what executors:

- Users may require the *no_exceptions, single_exception_reduction*, and *multiple_exceptions* properties from any *twoway* or *then* executor without having to know whether the executor is a *single* or *bulk* executor.
- Users may only associate the *single_exception* property with *single* executors, since with *bulk* executors a reduction function is needed to reduce multiple exceptions into one.
- *Oneway* executors only support the *no_exceptions* property.

We propose that the default exception handling property for both *single* and *bulk oneway* executors is *no_exceptions*. For *single twoway* executors it is *single_exception*. For *bulk twoway* executors, the default is *multiple_exceptions*.

The table below summarizes how exception handling properties tie together with combinations of other properties (✔ = combination is supported, (✔) = combination is supported but not typical, ❌ = combination is normally not supported):

	no_exceptions	*multiple_exceptions*	*single_exception*	*single_exception_reduction*
single, oneway	✔	❌	❌	❌
bulk, oneway	✔	❌	❌	❌
single, twoway/then	✔	(✔)	✔	(✔)
bulk, twoway/then	✔	✔	❌	✔

Below is a code example showing how the proposed exception handling could be used:

```
// User-provided reduction function
std::exception_ptr my_reduction(std::exception_list l);

template<class Executor>
void do_parallel(Executor const& ex)
{
      // ...create task, shape, and parameter and result factories
      auto result_fut =
             execution::require(ex, execution::bulk, execution::twoway,
                   execution::single_exception_reduction(my_reduction)
             ).bulk_twoway_execute(task, shape, result_fact,
                                   param_fact);
      try { result_fut.wait(); }
      catch (my_exception const& e) { ... }
      // ...
}

// Use multiple_exceptions to get a list of exceptions
template<class Executor>
void parallel_stl(Executor const& ex, std::vector<int>& vec)
{
      auto ex2 = execution::require(ex, execution::multiple_exceptions);
      try
      {
            sort(std::execution::par.on(ex2),
                  vec.begin(), vec.end(), &comparison_that_throws);
            for_each(std::execution::par.on(ex2),
                     vec.begin(), vec.end(),
                     [](auto e){ if (...) throw ...; });
      }
      catch (std::exception_list const& e)
      {
         // Iterate list to react to exceptions
      }
}
```

5.2 Exception Reduction

In this proposal, exception reduction is proposed to be performed with a user-provided function, which takes an exception_list containing all encountered exceptions as its parameter, and returns an exception_ptr pointing to the exception, which should be propagated to the user. This section briefly discusses some topics related to exception reduction. For a more thorough discussion, see [1] (sections 5.5 - 5.7).

The idea of exception reduction is to come up with a single exception that best represents the total "exceptional" situation that manifests as a set of concurrent exceptions. It is quite likely that there is no single strategy for this that would apply to all use cases. For example, in some cases exception reduction may be as simple as ranking the exception types and choosing the exception with the highest rank. In other cases, it might be that the "importance" of an exception depends also on how many exceptions of that type have been received. And it is of course also possible that the "most representative" exception is not among the exceptions in the list, but rather would be a completely new exception created by the reduction function. Finally, even if simple ranking is enough to choose the outcome of reduction, it could still be necessary to collect and combine information from all exceptions of the highest rank to embed that information in the resulting exception. [1] contains two concrete example uses cases for exception reduction (sections 6.2 and 6.3).

One further reduction strategy is to find the most derived common base class for all the exceptions, and replace the set of exceptions with a single exception object of that type. This kind of reduction can be convenient as it provides a general way of reducing an arbitrary set of exceptions. On the other hand, this kind of reduction loses information about the types of individual exceptions. It does not allow certain exception types to have a higher precedence than others. If a fatal exception and a minor exception are reduced, the result is their common base class, which abstracts the types of individual exceptions away. Catching this base class exception object does not reveal whether a fatal exception has occurred. Further discussion on this strategy can also be found in [1] (section 5.7).

In this proposal, it is expected that reduction functions are called only once for each set of executions, after all encountered exceptions have been collected into an exception_list. This strategy makes sure that reduction functions have all the information for performing analysis on exceptions. However, another option would be to allow intermediate partial reductions to be performed on subsets of exceptions, and final reduction to be based on the results of these intermediate reductions, allowing parallel reduction. These alternatives are discussed further in Alternatives considered, and it is a topic for future discussion.

5.3 Effect on parallel STL

C++17 added 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 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).

If parallel STL algorithm execution policies are based on executors, then the exception handling properties proposed in this paper can be used to control exceptions resulting from parallel STL algorithms as well. Using the properties the user can specify whether there can be any exceptions at all, and whether to receive a single exception (through reduction) or multiple exceptions (in an exception_list). Similarly the parallel STL algorithms can use the exception properties to adapt their behaviour (for example, to use *oneway* executors if *no_exceptions* property is associated with the provided executor).

A note concerning the use of exception_list for multiple exceptions in parallel STL: 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. However with the introduction of executors it's now possible to provide a safe default whilst also providing other options for users who chose it.

5.4 Use of other disappointment mechanisms

Since it is possible that all platforms cannot easily support exception handling (or exception handling causes too much overhead), there are proposals for other "disappointment" handling mechanisms (std::expected<>, etc.) [8]. Even though this paper concentrates on handling of concurrent exceptions, possibility of using other disappointment mechanisms is discussed briefly.

If exception handling is not supported by the executor, the executor can signal this by only supporting the *no_exceptions* property. For *oneway* executors, signalling disappointment is not an issue since there's no back channel for the results of execution.

For *twoway* and *then* versions of *single no_exceptions* executors it is possible to use the result future to store a return type supporting disappointments (like std::expected<>) to pass information about the single potential disappointment back to the user. The user can do this by specifying an appropriate return type, so no special support in the executors is needed.

Twoway and *then* versions of *bulk* executors use a shared result object (created from the user-provided result factory) to collect results of the multiple executions, and this result object is embedded in the result future. Again, since the result object is provided by the user, other disappointment mechanisms can be used by storing the disappointments in the result object. Again, no support from the executors is needed, since the user controls the functions executed in parallel as well as the result object.

6 Alternatives considered

All exception handling properties in this paper (*no_exceptions, single_exception, single_exception_reduction,* and *multiple_exceptions*) are different approaches to handling multiple encountered exceptions. Alternatively, it would be possible to have just one exception handling property, which is always given a reduction function, and let the chosen reduction function to decide how to react to multiple exceptions. I.e., for *no_exceptions,* reduction function could call std::terminate(). For *multiple_exceptions,* it would return the exception_list. For *single_exception* it would pick the only exception in the exception list and return it (and the *single_exception_reduction* case already uses a reduction function). This proposal uses four separate exception handling properties, because this allows executors to support only some subset of exception handling properties. Additionally, and executor can be queried for its exception handling property, and decisions can be made based on that information. These would become impossible or at least much more complicated, if there was only one exception handling property. Also multiple properties allow an implementation to provide different executor types, which are optimized or specialized for a particular exception handling property.

This proposal uses the proposed std::exception_list to collect encountered exceptions and to pass them to reduction functions. In the case of the *multiple_exceptions* property, an exception_list is also embedded in the result future. As mentioned in [6] (section 2.2), the current interface on exception_list is quite limited, and probably not suitable for general exception reduction (especially since the result of exception reduction might in some cases still be an exception_list, so users should be able to create exceptions_lists and fill them with exceptions). Further analysis is needed for the set of ideal operations in exception_list. If std::exception_list cannot provide necessary operations, a new exception container type would be needed for the mechanism proposed in this paper.

Somewhat similarly to the exception_list problem, the std::exception_ptr type in the current standard is quite limited in functionality, and does not provide enough support for exception analysis needed in exception reduction. Currently, C++ does not provide any means for converting std::exception_ptr to a normal pointer pointing to the exception object (the only way is to re-throw the exception and catch it through a reference, which causes unacceptable overhead). There have been suggestions to add an exception_ptr_cast or similar to create ordinary pointers from std::exception_ptr (similar to dynamic_cast). This would allow reduction functions to iterate through encountered exceptions, query their types, access the actual exception objects, and base reduction on that information. If std::exception_ptr does not allow these operations, it would be necessary to store exceptions as a collection of std::exception*, which would limit exception support to types derived from std::exception.

Canceling parallel executions (both not initiating remaining un-started executions and possibly interrupting already started in-progress executions) in case of disappointments is a separate issue and is not discussed in this paper. However, cancellation affects also encountered exceptions. On one hand, it is reasonable not to initiate new executions if the whole operation is already known to fail (because of already encountered exceptions). On the other hand, not initiating the remaining executions could hide exceptions that would have been considered more important than the ones already encountered. For this reason, it should be considered whether the user (and possibly the reduction function) should be notified if there were parallel executions that were not run to completion or not started at all.

The proposal is currently based on the idea that exception reduction is performed once when all encountered exceptions have been collected. This allows exception reduction to use all available information to decide the result of reduction, and minimizes amount of nondeterminism in reduction. However, this also prevents intermediate (possibly parallel) partial reduction along the way, which could sometimes be useful. If partial reduction is allowed, in the extreme executors could perform intermediate reduction on just pairs of exceptions, making reduction into a user-provided comparison function choosing from two exceptions. This would allow maximum parallelism in reduction, but would limit information in each reduction step to two (arbitrary) exceptions. This would only work, if exceptions form a strict weak ordering so that the "most important" one can be determined by pairwise comparison. On the other hand, allowing intermediate reductions could have a positive effect on memory requirements, since all encountered exceptions wouldn't necessarily be propagated to a single place for reduction. There has been discussion back and forth on this issue, and it is one clear topic for future work. One possible solution could be to allow intermediate partial reduction, but add a boolean flag parameter telling reduction functions whether final or intermediate reduction is taking place. This would allow reduction functions to adapt their strategy accordingly (for example, intermediate reductions could mainly collect information for the final reduction).

The current proposal doesn't impose a limit to the number of concurrent exceptions which are handled and stored in the exception_list. However, concurrently dynamically allocating memory for exceptions can potentially be costly, or even not possible in the case of many GPUs. For these cases it's desirable to specify a fixed limit on the number of exceptions, or more specifically the memory available to store exceptions. One approach to this would be to allow the exception_list to be constructible using a fixed size limit where it would preallocate memory for storing exceptions. Then have an execution policy, an executor property or an execution context property which allows users to specify a fixed size which is then used when constructing the exception_list. This raises the concern about how to handle an overflow of exceptions, i.e. what to do when the fixed size limit is reached. A way to handle this would be to simply drop any exceptions after the limit has been reached and perhaps increment an atomic counter recording the number of exceptions which were dropped. Another option could be to have memory reserved in the exception_list for an exception which encapsulates the number of exceptions which were dropped.

7 Future Directions

Further study is needed to improve the interface of std::exception_ptr and the proposed std::exception_list so that they provide enough functionality for exception reduction.

More study is also needed to determine how executor exception handling and cancellation of executions can best be combined, so that unnecessary overhead is avoided, but the user is informed about potentially hidden exceptions.

This current paper presents certain combinations of the cardinality and directionality executor properties (*oneway*, *twoway*, *single*, *bulk*) and the exception handling properties (*no_exceptions*, *single_exception*, *single_exception_reduction*, *multiple_exceptions*) as supported or not supported. As the exact nature of how properties interoperate with each other and how the side effects of one property on another is reflected in the design of executors is still to be decided, this concept of supported or not supported is mainly theoretical to represent the valid combinations. How this is reflected in the executors interface shall follow the design of the executors proposal.

Finally future work is needed for deciding whether to allow partial intermediate reductions or require the reduction function to be called only once for each set of executions.

8 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:

- Is it reasonable to hope for additional support in std::exception_ptr, so that it could be used in exception reduction?
- Is it enough to perform exception reduction only after all encountered exceptions have been collected, or should there be a way to signal that parallel partial reduction is also allowed?
- How important is it to have the option for both a dynamic and fixed size exception_list.

9 Acknowledgement

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

10 References

[1] Matti Rintala, Michael Wong, Carter Edwards, Patrice Roy, Gordon Brown: *Exception Handling in Parallel Algorithms.* P0797R0

[2] Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, Carter Edwards, Gordon Brown, et al: *A Unified Executors Proposal for C++.* P0443R4

[3] Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, Carter Edwards, Gordon Brown, Michael Wong: *Executors Design Document* P0761R1

[4] 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)

[5] Jared Hoberock: *Working Draft, Technical Specification for C++ Extensions for Parallelism Version 2*. N4706

[6] Alisdair Meredith: *Rebase the Parallelism TS onto the C++17 Standard.* P0776R1

[7] Khronos OpenCL Working Group — SYCL subgroup: Khronos Group SYCL 1.2.1 Specification. (PDF version)

[8] Vicente Botet, JF Bastien. p0323r4 std::expected. (PDF version)

[9] Wong et al. Towards an Error Model for OpenMP. https://link.springer.com/chapter/10.1007%2F978-3-642-13217-9_6