Document number: P0797R2
Date: 2019-06-17
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@tuni.fi, michael@codeplay.com, mhoemme@sandia.gov
Reply to: Matti Rintala

Handling Exceptions in Executors

1 Introduction and summary

This paper describes a mechanism for handling exceptions (EH) emerging from multiple concurrent/asynchronous executions with executors. The authors realize that multiple asynchronous exceptions may arise from other sources as well (especially in the future versions of the standard), so the attempt is to propose a relatively simple solution that would still address as many foreseeable use cases as possible. 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, the proposal was significantly changed in the next version *P0797R1: Handling Concurrent Exceptions with Executors* to work with executors as proposed in P0443R4. Since then the development of exceptions has bifurcated to alternatively use the sender/receiver model, and this version takes that development into account. Some parts of this proposal could be also used to control how single exceptions thrown from non-bulk single executors are handled.

Summary of the proposal:

Add a set of (non-API-changing) properties to executors for defining whether executors support exception handling and how they react to exceptions thrown from executed user code (don't support, terminate, propagate, collect, reduce, etc.). Executors can extend this set with their own executor-specific exception handling properties, if needed.

2 Revision History

2.1 2019-07-15 [P0797R2] CGN Meeting

- This version is based on feedback comments from JAX
- Takes also into account the proposed sender/receiver model
- The proposed exception handling properties are expanded into a conceptual hierarchy
- Abandon exception reduction as a primary mechanism, but still allow that as executor-dependent extension
- Add a property allowing an executor to not support exception handling at all

2.2 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.

2.3 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 executions created using executors. In the future-based model [2], twoway executors propagate their results into a returned future. In the sender/receiver model [3], twoway executors propagate their results to a receiver using its value or error channels. 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 error value can be propagated the future or the receiver's error channel. 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 multiple things can go wrong and cause an exception to be thrown in an algorithm, there are several sources of non-determinism affecting which exception(s) are actually encountered. Parallelism/concurrency may cause non-determinism on *when* a problem is noticed and an exception thrown, thus affecting the relative order of exceptions. Moreover, earlier exception handling can cause some later code not to be executed, which means that the set of exceptions may be different depending on the order the exceptions were triggered.

On the other hand, in many algorithms, some design choices are typically left to the implementation, and only the outcome of the algorithm is specified. Thus even without parallelism, there's non-determinism in *what* an algorithm actually does, and in what order. This means that running a different implementation of the same algorithm for the same input may trigger different exceptions. This becomes even more complicated if the algorithm provides customization points, in which case running even the same algorithm with a different customization may trigger a different exception (or set of exceptions).

In general, this means that although it's useful to know that an algorithm has failed (resulted in exception(s) instead of a successful result), it may be less useful to know what individual exceptions have caused this failure. This is because even the same failure may manifest in many different sets of exceptions, depending on run-time parallelism, customization, and implementation freedom in the algorithm itself.

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 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.

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.

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, and how to deal with execution environments which are incapable of exception propagation.

4 The State of the Art

The original design of the executors proposal was laid out in Executors Design Document [5], and A Unified Executors Proposal for C++ [6]. Then a compromise proposal introducing the sender/receiver executor model was presented in [3]. The original executor proposal was split so that only one-directional executors remained in [6], dependent (two-way and then-execution) was moved to [2], and the property mechanism was separated into [7].

The original executor proposal [6] 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. In the proposed sender/receiver model, “[t]he behavior on an exception escaping the passed task is executor-defined.”

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. The sender/receiver model could report the exception by calling receiver's *set_error* with the exception as a *std::exception_ptr* parameter.

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. The sender/receiver model doesn't seem to (at least to the eyes of this author) currently specify what happens if bulk execution throws exceptions, and the sender/receiver model's approach to bulk execution seems to be currently under active discussion.

For static_thread_pool executors, [6] 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. When a *task_block* in Parallelism TS 2 [8] encounters (multiple) exceptions, it stores them in a *std::exception_list* and propagates that exception forward.

Concurrency TS [9] provides *std::when_all* and *std::when_any*, which can be used to wait for a collection of futures, which of course can contain exceptions. These two functions return a future containing a collection of futures when all (*std::when_all*) or any (*std::when_any*) of the futures become ready. Several of the futures may contain exceptions, but they remain in the individual futures and are not propagated to the future returned by *std::when_all/std::when_any.*

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 [10].

Khronos’s SYCL standard is a modern C++ programming model based on OpenCL, suitable for heterogeneous dispatch, and follows the C++11/14/17/20 progression [11]. 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 closely following 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.

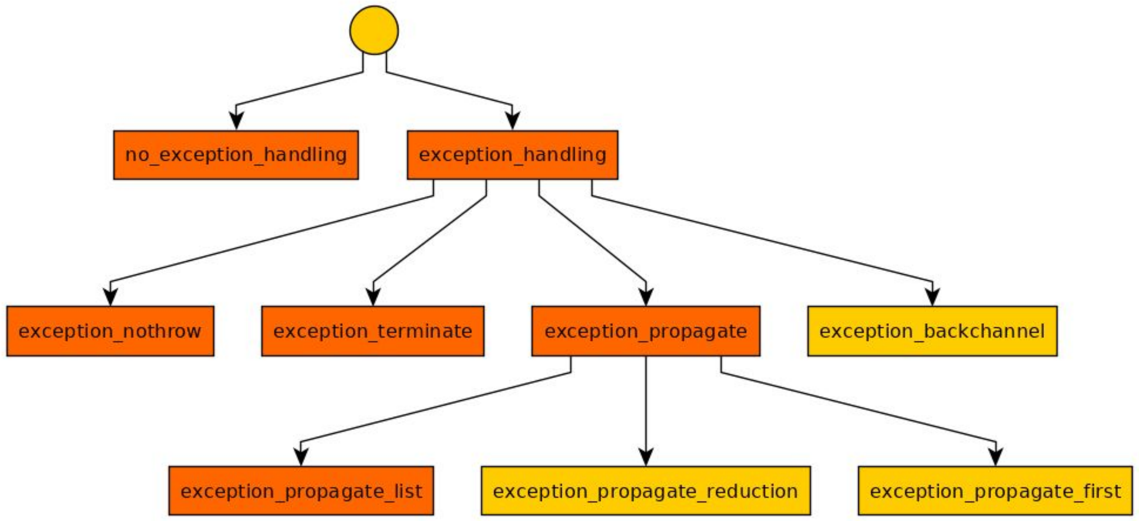
5 Proposed Solution and discussion

Concurrency and asynchrony make it possible to have multiple simultaneous disappointments active at the same time. A mechanism handling those has to take into account at least the following things:

- The current C++ exception handling model can have at most one exception propagated propagated on any level. Exception handling may be nested (destructors are allowed to trigger exceptions, if they do not propagate out of the destructor), but if the exception handling mechanism handling an uncaught exception directly invokes a function that exits via an exception, the function std::terminate is called. So, in the end only one exception may be propagated out of an executor (an arbitrary number of disappointments can of course be stored as values at the same time, and this includes exception objects at the end of std::exception_ptr, for example).
- Bulk executors create executions in batches, and it is under discussion whether the remaining not started executions should still be started and run to completion even if earlier executions have thrown exceptions. If both alternatives are possible, it may increase the amount of uncertainty over how many (and which) multiple exceptions may be thrown during the bulk execution.
- Somewhat similarly to above, implementations of parallel algorithms (using executors) have often freedom in how to use parallelism to boost up performance. There have also been discussions on adding customization points so that executors may customize parallel algorithms to better utilize execution capabilities of the executor. This freedom adds up even more uncertainty and possibly non-determinism in what exceptions and how many may occur in parallel.
- In some execution environments, supporting exception handling at all may be problematic (GPUs, SIMD, ...). Either implementing exception handling could be technically very difficult, or it could cause unacceptable overhead to the execution.

5.1 Exception Handling Properties

The following diagram shows proposed hierarchy of exception handling properties for executors. These properties control only what is allowed in exception handling and how executors behave on encountering exceptions, requiring any of these properties does not change the API of the executor. The properties drawn in red are the ones this paper actually proposes to be standardized. Properties drawn in yellow are examples of how individual executors might want to provide additional exception handling properties as an extension (i.e. these are only included as an illustration). If an executor supports an exception handling property, it should also support properties higher up in that branch of the hierarchy.



5.1.1. Property *no_exception_handling*

An executor with this property does not promise to support exception handling in the executed user code at all. This means that if the provided user code contains any exception handling code, the program is ill-formed. This property is necessary for executors whose execution environment cannot support exception handling or exception handling would impose unacceptable bloat or overhead (for example some GPU or FPGA based executors). Alternatively this property could be relaxed to mean that if exceptions are thrown in user code, the behaviour is undefined or implementation-dependent. Then a GPU executor not supporting exception handling might ignore try-catch blocks while compiling code to be offloaded to the GPU, and potentially could also optimize out all code branches that always end up throwing exceptions (this would make it possible to call library functions that can theoretically throw exceptions, but are called in a context where exceptions cannot occur).

It should be noted that since exception handling is a mandatory part of C++, this property cannot be used as a default in any executor, since it would be quite questionable to require users to query for a property in order to do something that is regularly supported by the language. A GPU executor not supporting exception handling should by default not have this property, and just refuse to execute anything. To actually use such a GPU executor would require first preferring (or requiring) *no_exception_handling*, which produces an executor capable of executing code that does not contain exception handling.

It is understood that this property may be somewhat controversial, since it allows creation of executors that do not fully implement the C++ language. However, the need for being able to use executors without exception handling capabilities has been brought up many times. This property would likely always be preferred, not required, since an executor capable of exception handling is of course also capable of running code that does not contain exception handling.

5.1.2 Property *exception_handling*

This property allows users to require that exception handling is possible in executed user code, but at the same time not specify how the executor should actually react to exceptions thrown out of user code. As discussed above, by default all executors should provide this capability, unless the *no_exceptiong_handling* property is preferred/required.

5.1.3. Property *exception_nothrow*

This property is a slight refinement of *exception_handling*, and specifies that all user code executed by the executor has to be nothrow, i.e. exceptions have to be handled inside user code and are not allowed to propagate out of it. This property is not absolutely necessary, since property *exception_handling* also covers this behaviour, but it could provide some optimization possibilities, if the executor knows that executed user code does not propagate out exceptions.

5.1.4 Property *exception_terminate*

Executors with this property call *std::terminate* if exceptions are thrown out of executed user code. This is what C++17 parallel STL specifies, so it should probably be included as an option.

5.1.5 Property *exception_propagate*

Executors with this property will propagate an exception, if exceptions are thrown out of user code. This property will allow the actual propagated exception to be implementation-defined. I.e., if an exception (or exceptions) are thrown out of user code, the executor will propagate *some exception* either to the returned future or the attached receiver's error channel. It is executor-dependent whether this exception is the exception thrown by user code (or one of them), or an exception created by the executor. Even with this much ambiguity, the user knows that an execution that fails because of exception(s) will result in an exception in the future or receiver.

5.1.6 Property *exception_propagate_list*

Executors with this property will propagate an exception of type *std::exception_list*, if exceptions are thrown out of user code. This exception list *may* contain at least *some* of the exceptions thrown out of user code. It is undefined what exceptions are included in the list, how many, and in what order (and this may depend on the executor, run-time properties like available execution environments, etc.). This property ensures that users are guaranteed to get an exception of type *std::exception_list*, and can thus catch such exceptions, whereas *exception_propage* leaves open the type of the propagated exception (which can be refined with executor-dependent properties, see below). In discussions some have raised concerns that it should be possible to distinguish between exceptions raised by the executor itself and exceptions thrown from user code. If that is considered necessary, it would be possible to change this property so that the propagated exception is a special exception type signifying propagated user exceptions, and that exception could contain an embedded exception list.

5.1.7 Properties *exception_backchannel, exception_propagate_first, exception_propagate_reduction*

These are examples of properties that some executors might introduce as extensions of the hierarchy to specify a different or more specific exception handling strategy. In this example, *exception_propagate_first* could propagate the first encountered exception as such (reasonable choice for a sequential executor that will always only encounter one exception). Similarly *exception_propagate_reduction* could be provided by an executor that performs exception analysis and reduction (see section 5.2 below) on encountered exceptions, and propagates the result of the reduction. Property *exception_backchannel* is included as an illustration for executors that propagate thrown exceptions using some executor-specific backchannel (this could be useful, for example, in one-way executors with no regular return channel).

5.2 Exception Reduction

In the previous version of this proposal, exception reduction was proposed to be performed with a user-provided function, which would take an exception list containing all encountered exceptions as its parameter, and return an exception pointer, which would then be propagated to the user. Although exception reduction is no longer at the core of this proposal, this section briefly discusses some topics related to exception reduction, since the need for reduction of multiple exceptions still exists. For a more thorough discussion, see [4] (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. Sections 6.2 and 6.2 of [4] give two concrete example uses cases for exception reduction.

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 [4] (section 5.7).

There are at least two ways to do the reduction. All encountered exceptions can be collected and then reduced together. This gives the reduction function maximum amount of information to do the reduction, but on the other hand it forces reduction to be performed as a single (possibly massive) step in one place. Another way is to allow pairwise reduction, in which case the reduction function is reduced (sic) to a function taking two exceptions and returning an exception. This pairwise reduction would then be called transitively like in *std::reduce* to come up with a single exception. This latter strategy assumes that exception reduction is possible by seeing only two exceptions at any time.

As mentioned before, the previous version of this paper suggested a user-provided exception reduction function which would be passed to an executor via a property or some other means. However, the same need for reduction arises in other places where multiple exceptions occur, and all of them do not necessarily have a suitable property mechanism. However, even the proposed simple solution of throwing a std::exception_list and allowing executors to embed individual encountered exceptions inside that exception (in an exception-dependent manner) still allows exception reduction, if needed. The user can surround the code potentially producing exception lists with an extra try-block which catches *std::exception_list*. In the catch-block it is possible to do exception reduction and throw the resulting exception. Or an exception reduction performed by the executor could be provided by some executors as a executor-dependent property extending the property hierarchy of this proposal.

In some discussions it has been brought up that reduction wouldn't necessarily have to be performed on exceptions only; results of the successful executions could also participate in reduction. This would allow ignoring minor exceptions, if the successful executions were enough for a result, etc. However, this kind of “reduction of all results” is not included in this proposal. Exceptions are meant to be thrown only in exceptional circumstances and they are not supposed to represent “partial results,” which could be ignored. If there is a need to do reduction based on both successful and failed executions, then other disappointments mechanisms like *std::expected* are better suited to express that kind of failure, and reduction based on those can be done by the user.

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. In this proposal, the same behaviour can still be achieved with the property *exception_terminate.*

The sequential STL algorithms allow exceptions to be used to signal failure to complete the algorithm. These exceptions may be thrown from *element access functions*, including 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 contrast to successful completion).

A note concerning the use of exception_list for multiple exceptions: 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. Lately there has been discussions on whether it would be useful to somewhat relax that strict choice, and allow some kind of exception handling in Parallel STL as well. This proposal is allows executors to either stick to the original strict rule or perform more complex exception handling.

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, and parallel STL algorithms can use exception handling properties to adapt their own behaviour. For example, an implementation of *std::sort* might start multiple independent (i.e., non-bulk) parallel executions, and query the executor's exception handling properties to handle its own (possible multiple) exceptions in a compatible way.

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.) [12]. 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 the properties it supports. For *oneway* executor, signalling disappointment is not an issue since there's no back channel for the results of execution, so no result is transmitted back.

For *twoway* executors (and the *then* versions) 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. Similarly in the sender/receiver model all user disappointments could be propagated to the value channel in the form of *std::expected* or similar. Executors would only have to be aware of disappointments if they should cause cancellation of not yet started executions, or if user disappointments should be directed to the receiver's error channel instead of the value channel.

Twoway and *then* versions of *bulk* executors use a shared result object (created from the user-provided result factory) to collect the results of multiple executions. This result object is embedded in the result future or propagated to the receiver's value channel. Again, since the result object is provided by the user, other disappointment mechanisms can be used by storing the disappointments in the result object. When determining the final result, both successful results and disappointments can be taken into account. Again, no support from the executors is needed, since the user controls the functions executed in parallel as well as the result object. In the sender/receiver model, executor support would again be needed only if disappointments should be propagated into the error channel of the receiver.

6 Future Directions

Currently executors and the sender/receiver model is under heavy development, so work is needed to adapt this proposal to the resulting executor model.

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 analysing exceptions stored as values and possibly performing 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 the possibility of losing exceptions from executions that were never started or were cancelled.

7 Design questions

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 the extensible family of exception handling properties an idea that should be developed further?

- If yes to above, what properties should be standardized, what left up to individual executors to add, if necessary? What to call the properties?
- Should executors be allowed to forbid the use of any exception handling in user code executed under the executor? (Property *no_exception_handling*)
- If yes to above, is the proposed strategy good for *no_exception_handling*?: Make user code containing exception handling illegal? Allow but ignore possible EH-code (try/catch/throw) assuming it is not executed (optimization possibilities for off-loading). Make it undefined what happens if an exception is actually thrown in the code executed under an executor not supporting exception handling?

8 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.

9 References

[1] Matti Rintala, Michael Wong, Carter Edwards, Patrice Roy, and Gordon Brown. *Exception Handling in Parallel Algorithms.* P0797R0
[2] Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, Carter Edwards, Gordon Brown, et.al. *Dependent Execution for a Unified Executors Proposal for C++.* P1244R0
[3] Lee Howes, Eric Niebler, Kirk Shoop, Bryce Lelbach, David S. Hollman. *The Compromise Executors Proposal: A lazy simplification of P0443.* P1194R0
[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, Michael Garland, Chris Kohlhoff, Chris Mysen, Carter Edwards, Gordon Brown, Michael Wong. *Executors Design Document*. P0761R2
[6] Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, Carter Edwards, Gordon Brown, David Hollman, et al. *A Unified Executors Proposal for C++.* P0443R10
[7] David Hollman, Chris Kohlhoff, Bryce Lelbach, Jared Hoberock, Gordon Brown, and Michał Dominiak. *A General Property Customization Mechanism.* P1393R0
[8] Jared Hoberock. *Working Draft, Technical Specification for C++ Extensions for Parallelism Version 2*. N4706
[9] *The C++ Extensions for Concurrency, ISO/IEC TS 19571:2016*
[10] Wong et al. *Towards an Error Model for OpenMP*.
https://link.springer.com/chapter/10.1007%2F978-3-642-13217-9_6
[11] Khronos OpenCL Working Group — SYCL subgroup. *Khronos Group SYCL 1.2.1 Specification*. (PDF version)
[12] Vicente Botet and JF Bastien. *std::expected*. P0323R4 (PDF version)