TECHNICAL SPECIFICATION

ISO/IEC TS 19570

First edition
2015-07-01

Programming Languages — Technical Specification for C++ Extensions for Parallelism

Langages de programmation — Spécification technique pour les extensions C++ relatives au parallélisme

Reference number
ISO/IEC TS 19570:2015(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the WTO principles in the Technical Barriers to Trade (TBT) see the following URL: Foreword - Supplementary information

ISO/IEC TS 19570 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

Contents

1 General [parallel.general]

1.1 Scope [parallel.general.scope]

1 This Technical Specification describes requirements for implementations of an interface that computer programs written in the C++ programming language may use to invoke algorithms with parallel execution. The algorithms described by this Technical Specification are realizable across a broad class of computer architectures.

2 This Technical Specification is non-normative. Some of the functionality described by this Technical Specification may be considered for standardization in a future version of C++, but it is not currently part of any C++ standard. Some of the functionality in this Technical Specification may never be standardized, and other functionality may be standardized in a substantially changed form.

3 The goal of this Technical Specification is to build widespread existing practice for parallelism in the C++ standard algorithms library. It gives advice on extensions to those vendors who wish to provide them.

1.2 Normative references [parallel.general.references]

1 The following referenced document is indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

— ISO/IEC 14882:—[1], *Programming Languages — C++*

2 ISO/IEC 14882:— is herein called the *C++ Standard*. The library described in ISO/IEC 14882:— clauses 17-30 is herein called the *C++ Standard Library*. The C++ Standard Library components described in ISO/IEC 14882:— clauses 25, 26.7 and 20.7.2 are herein called the *C++ Standard Algorithms Library*.

3 Unless otherwise specified, the whole of the C++ Standard's Library introduction (C++14 §17) is included into this Technical Specification by reference.

1.3 Namespaces and headers [parallel.general.namespaces]

1 Since the extensions described in this Technical Specification are experimental and not part of the C++ Standard Library, they should not be declared directly within namespace `std`. Unless otherwise specified, all components described in this Technical Specification are declared in namespace `std::experimental::parallel::v1`.

[*Note:* Once standardized, the components described by this Technical Specification are expected to be promoted to namespace `std`. — *end note*]

2 Unless otherwise specified, references to such entities described in this Technical Specification are assumed to be qualified with `std::experimental::parallel::v1`, and references to entities described in the C++ Standard Library are assumed to be qualified with `std::`.

3 Extensions that are expected to eventually be added to an existing header `<meow>` are provided inside the `<experimental/meow>` header, which shall include the standard contents of `<meow>` as if by

1. To be published. Section references are relative to N3937.

```
#include <meow>
```

1.4 Terms and definitions [parallel.general.defns]

1 For the purposes of this document, the terms and definitions given in the C++ Standard and the following apply.

2 A *parallel algorithm* is a function template described by this Technical Specification declared in namespace `std::experimental::parallel::v1` with a formal template parameter named `ExecutionPolicy`.

3 Parallel algorithms access objects indirectly accessible via their arguments by invoking the following functions:

- All operations of the categories of the iterators that the algorithm is instantiated with.
- Functions on those sequence elements that are required by its specification.
- User-provided function objects to be applied during the execution of the algorithm, if required by the specification.
- Operations on those function objects required by the specification. [*Note:* See clause 25.1 of *C++ Standard Algorithms Library*. — *end note*]

These functions are herein called *element access functions*.
[*Example:* The `sort` function may invoke the following element access functions:

- Methods of the random-access iterator of the actual template argument, as per 24.2.7, as implied by the name of the template parameters `RandomAccessIterator`.
- The `swap` function on the elements of the sequence (as per 25.4.1.1 [sort]/2).
- The user-provided `Compare` function object.

— *end example*]

1.5 Feature-testing recommendations [parallel.general.features]

1 An implementation that provides support for this Technical Specification shall define the feature test macro(s) in Table 1.

Table 1 — Feature Test Macro(s)

Name	Value	Header

`__cpp_lib_experimental_parallel_algorithm`	201505	`<experimental/algorithm>`
		`<experimental/exception_list>`
		`<experimental/execution_policy>`
		`<experimental/numeric>`

2 Execution policies [parallel.execpol]

2.1 In general [parallel.execpol.general]

1 This clause describes classes that are *execution policy* types. An object of an execution policy type indicates the kinds of parallelism allowed in the execution of an algorithm and expresses the consequent requirements on the element access functions.

[*Example:*

```
std::vector<int> v = ...

// standard sequential sort
std::sort(v.begin(), v.end());

using namespace std::experimental::parallel;

// explicitly sequential sort
sort(seq, v.begin(), v.end());

// permitting parallel execution
sort(par, v.begin(), v.end());

// permitting vectorization as well
sort(par_vec, v.begin(), v.end());

// sort with dynamically-selected execution
size_t threshold = ...
execution_policy exec = seq;
if (v.size() > threshold)
{
  exec = par;
}

sort(exec, v.begin(), v.end());
```

— *end example*]

[*Note:* Because different parallel architectures may require idiosyncratic parameters for efficient execution, implementations of the Standard Library may provide additional execution policies to those described in this Technical Specification as extensions. — *end note*]

2.2 Header `<experimental/execution_policy>` synopsis [parallel.execpol.synopsis]

```
namespace std {
namespace experimental {
namespace parallel {
inline namespace v1 {
  // 2.3, Execution policy type trait
  template<class T> struct is_execution_policy;
  template<class T> constexpr bool is_execution_policy_v = is_execution_policy<T>::value;
```

```
  // 2.4, Sequential execution policy
  class sequential_execution_policy;

  // 2.5, Parallel execution policy
  class parallel_execution_policy;

  // 2.6, Parallel+Vector execution policy
  class parallel_vector_execution_policy;

  // 2.7, Dynamic execution policy
  class execution_policy;
}
}
}
}
```

2.3 Execution policy type trait [parallel.execpol.type]

```
template<class T> struct is_execution_policy { see below };
```

1 `is_execution_policy` can be used to detect parallel execution policies for the purpose of excluding function signatures from otherwise ambiguous overload resolution participation.

2 `is_execution_policy<T>` shall be a UnaryTypeTrait with a BaseCharacteristic of `true_type` if `T` is the type of a standard or implementation-defined execution policy, otherwise `false_type`.

[*Note:* This provision reserves the privilege of creating non-standard execution policies to the library implementation. — *end note*]

3 The behavior of a program that adds specializations for `is_execution_policy` is undefined.

2.4 Sequential execution policy [parallel.execpol.seq]

```
class sequential_execution_policy{ unspecified };
```

1 The class `sequential_execution_policy` is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and require that a parallel algorithm's execution may not be parallelized.

2.5 Parallel execution policy [parallel.execpol.par]

```
class parallel_execution_policy{ unspecified };
```

1 The class `parallel_execution_policy` is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be parallelized.

2.6 Parallel+Vector execution policy [parallel.execpol.vec]

```
class parallel_vector_execution_policy{ unspecified };
```

1 The class `class parallel_vector_execution_policy` is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be vectorized and parallelized.

2.7 Dynamic execution policy [parallel.execpol.dynamic]

```
class execution_policy
{
  public:
    // 2.7.1, execution_policy construct/assign
    template<class T> execution_policy(const T& exec);
    template<class T> execution_policy& operator=(const T& exec);

    // 2.7.2, execution_policy object access
    const type_info& type() const noexcept;
    template<class T> T* get() noexcept;
    template<class T> const T* get() const noexcept;
};
```

1 The class `execution_policy` is a container for execution policy objects. `execution_policy` allows dynamic control over standard algorithm execution.

[*Example:*

```
std::vector<float> sort_me = ...

using namespace std::experimental::parallel;
execution_policy exec = seq;

if(sort_me.size() > threshold)
{
  exec = std::par;
}

std::sort(exec, std::begin(sort_me), std::end(sort_me));
```

— *end example*]

2 Objects of type `execution_policy` shall be constructible and assignable from objects of type `T` for which `is_execution_policy<T>::value` is `true`.

2.7.1 `execution_policy` construct/assign [parallel.execpol.con]

1 `template<class T> execution_policy(const T& exec);`

2 *Effects:* Constructs an `execution_policy` object with a copy of `exec`'s state.

3 *Remarks:* This constructor shall not participate in overload resolution unless `is_execution_policy<T>::value` is `true`.

4 `template<class T> execution_policy& operator=(const T& exec);`

5 *Effects:* Assigns a copy of `exec`'s state to `*this`.

6 *Returns:* `*this`.

2.7.2 `execution_policy` object access [parallel.execpol.access]

1 `const type_info& type() const noexcept;`

2 *Returns:* `typeid(T)`, such that `T` is the type of the execution policy object contained by `*this`.

3 `template<class T> T* get() noexcept;`
`template<class T> const T* get() const noexcept;`

4 *Returns:* If `target_type() == typeid(T)`, a pointer to the stored execution policy object; otherwise a null pointer.

5 *Requires:* `is_execution_policy<T>::value` is `true`.

2.8 Execution policy objects [parallel.execpol.objects]

```
constexpr sequential_execution_policy      seq{};
constexpr parallel_execution_policy        par{};
constexpr parallel_vector_execution_policy par_vec{};
```

1 The header `<experimental/execution_policy>` declares a global object associated with each type of execution policy defined by this Technical Specification.

3 Parallel exceptions [parallel.exceptions]

3.1 Exception reporting behavior [parallel.exceptions.behavior]

1 During the execution of a standard parallel algorithm, if temporary memory resources are required and none are available, the algorithm throws a `std::bad_alloc` exception.

2 During the execution of a standard parallel algorithm, if the invocation of an element access function exits via an uncaught exception, the behavior of the program is determined by the type of execution policy used to invoke the algorithm:

- If the execution policy object is of type `class parallel_vector_execution_policy`, `std::terminate` shall be called.
- If the execution policy object is of type `sequential_execution_policy` or `parallel_execution_policy`, the execution of the algorithm exits via an exception. The exception shall be an `exception_list` containing all uncaught exceptions thrown during the invocations of element access functions, or optionally the uncaught exception if there was only one.

  [*Note:* For example, when `for_each` is executed sequentially, if an invocation of the user-provided function object throws an exception, `for_each` can exit via the uncaught exception, or throw an `exception_list` containing the original exception. — *end note*]

  [*Note:* These guarantees imply that, unless the algorithm has failed to allocate memory and exits via `std::bad_alloc`, all exceptions thrown during the execution of the algorithm are communicated to the caller. It is unspecified whether an algorithm implementation will "forge ahead" after encountering and capturing a user exception. — *end note*]

  [*Note:* The algorithm may exit via the `std::bad_alloc` exception even if one or more user-provided function objects have exited via an exception. For example, this can happen when an algorithm fails to allocate memory while creating or adding elements to the `exception_list` object. — *end note*]
- If the execution policy object is of any other type, the behavior is implementation-defined.

3

3.2 Header `<experimental/exception_list>` synopsis [parallel.exceptions.synopsis]

```
namespace std {
namespace experimental {
namespace parallel {
inline namespace v1 {

  class exception_list : public exception
  {
    public:
      typedef unspecified iterator;

      size_t size() const noexcept;
      iterator begin() const noexcept;
      iterator end() const noexcept;
```