TECHNICAL SPECIFICATION

ISO/IEC TS 19570

Second edition
2018-11

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 19570:2018(E)

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Fax: +41 22 749 09 47
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

Foreword [parallel.foreword]

1 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.

2 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`).

3 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`).

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

5 For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see `www.iso.org/iso/foreword.html`.

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

7 This second edition cancels and replaces the first edition (ISO/IEC 19570:2015) which has been technically revised.

8 The main changes compared to the previous edition are as follows:

(8.1) — Eliminate previously standardized functionality.

(8.2) — Introduce task block.

(8.3) — Introduce vector and wavefront policies.

(8.4) — Introduce a template library for parallel for loops.

(8.5) — Introduce data-parallel vector types.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at `www.iso.org/members.html`.

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

1 Scope [parallel.scope]

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

2 There is a possibility of a subset of the functionality described by this document being standardized in a future version of C++, but it is not currently part of any C++ standard. There is a possibility of some of the functionality in this document never being standardized, or of it being standardized in a substantially changed form.

3 The goal of this document is to build widespread existing practice for parallelism in the C++ programming language. It gives advice on extensions to those vendors who wish to provide them.

2 Normative references [parallel.references]

1 The following documents are referred to in the text in such a way that some or all of their content constitutes requirements 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.

(1.1) — ISO/IEC 14882:2017, *Programming languages — C++*

2 ISO/IEC 14882:2017 is herein called the C++ Standard. References to clauses within the C++ Standard are written as "C++17 §20". The library described in ISO/IEC 14882:2017 clauses 20-33 is herein called the *C++ Standard Library*. The C++ Standard Library components described in ISO/IEC 14882:2017 clauses 28, 29.8 and 23.10.10 are herein called the *C++ Standard Algorithms Library*.

3 Unless otherwise specified, the whole of the C++ Standard's Library introduction (C++17 §20) is included into this document by reference.

3 Terms and definitions [parallel.defns]

1 For the purposes of this document, the terms, definitions, and symbols given in ISO/IEC 14882:2017 apply.

2 ISO and IEC maintain terminological databases for use in standardization at the following addresses:

(2.1) — ISO Online browsing platform: available at https://www.iso.org/obp

(2.2) — IEC Electropedia: available at http://www.electropedia.org

4 General [parallel.general]

4.1 Namespaces and headers [parallel.general.namespaces]

1 Since the extensions described in this document 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 document are declared in namespace `std::experimental::parallelism_v2`.

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

2 Each header described in this document shall import the contents of `std::experimental::parallelism_v2` into `std::experimental` as if by

```
namespace std::experimental {
  inline namespace parallelism_v2 {}
}
```

3 Unless otherwise specified, references to such entities described in this document are assumed to be qualified with `std::experimental::parallelism_v2`, and references to entities described in the C++ Standard Library are assumed to be qualified with `std::`.

4 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

```
#include <meow>
```


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

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

Table 1 — Feature-test macro(s)

Title	Subclause	Macro Name	Value	Header
Task Block	8	`__cpp_lib_experimental_parallel_task_block`	201711	`<experimental/exception_list>` `<experimental/task_block>`
Vector and Wavefront Policies	6.2	`__cpp_lib_experimental_execution_vector_policy`	201711	`<experimental/algorithm>` `<experimental/execution>`
Template Library for Parallel For Loops	7.2.2, 7.2.3, 7.2.4	`__cpp_lib_experimental_parallel_for_loop`	201711	`<experimental/algorithm>`
Data-Parallel Vector Types	9	`__cpp_lib_experimental_parallel_simd`	201803	`<experimental/simd>`

5 Parallel exceptions [parallel.exceptions]

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

```
namespace std::experimental {
inline namespace parallelism_v2 {

  class exception_list : public exception {
  public:
    using iterator = unspecified;

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

    const char* what() const noexcept override;
  };
}
}
```

1 The class `exception_list` owns a sequence of `exception_ptr` objects.

2 `exception_list::iterator` is an iterator which meets the forward iterator requirements and has a value type of `exception_ptr`.

`size_t size() const noexcept;`

3 *Returns:* The number of `exception_ptr` objects contained within the `exception_list`.

4 *Complexity:* Constant time.

`iterator begin() const noexcept;`

5 *Returns:* An iterator referring to the first `exception_ptr` object returned within the `exception_list`.

`iterator end() const noexcept;`

6 *Returns:* An iterator that is past the end of the owned sequence.

`const char* what() const noexcept override;`

7 *Returns:* An implementation-defined NTBS.

6 Execution policies [parallel.execpol]

6.1 Header <experimental/execution> synopsis [parallel.execpol.synopsis]

```
#include <execution>

namespace std::experimental {
inline namespace parallelism_v2 {
namespace execution {
  // 6.2, Unsequenced execution policy
  class unsequenced_policy;

  // 6.3, Vector execution policy
  class vector_policy;

  // 6.4, Execution policy objects
  inline constexpr unsequenced_policy unseq{ unspecified };
  inline constexpr vector_policy vec{ unspecified };
}
}
}
```


6.2 Unsequenced execution policy [parallel.execpol.unseq]

```
class unsequenced_policy { unspecified };
```


1 The class `unsequenced_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, e.g., executed on a single thread using instructions that operate on multiple data items.

2 The invocations of element access functions in parallel algorithms invoked with an execution policy of type `unsequenced_policy` are permitted to execute in an unordered fashion in the calling thread, unsequenced with respect to one another within the calling thread. [*Note:* This means that multiple function object invocations may be interleaved on a single thread. — *end note*]

3 [*Note:* This overrides the usual guarantee from the C++ Standard, C++17 §4.6 that function executions do not overlap with one another. — *end note*]

4 During the execution of a parallel algorithm with the `experimental::execution::unsequenced_policy` policy, if the invocation of an element access function exits via an uncaught exception, `terminate()` will be called.

6.3 Vector execution policy [parallel.execpol.vec]

```
class vector_policy { unspecified };
```

1 The class `vector_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. Additionally, such vectorization will result in an execution that respects the sequencing constraints of wavefront application (7.1). [*Note:* The implementation thus makes stronger guarantees than for `unsequenced_policy`, for example. — *end note*]

2 The invocations of element access functions in parallel algorithms invoked with an execution policy of type `vector_policy` are permitted to execute in unordered fashion in the calling thread, unsequenced with respect

to one another within the calling thread, subject to the sequencing constraints of wavefront application (7.1) for the last argument to `for_loop`, `for_loop_n`, `for_loop_strided`, or `for_loop_strided_n`.

3 During the execution of a parallel algorithm with the `experimental::execution::vector_policy` policy, if the invocation of an element access function exits via an uncaught exception, `terminate()` will be called.

6.4 Execution policy objects [parallel.execpol.objects]

```
inline constexpr execution::unsequenced_policy unseq { unspecified };
inline constexpr execution::vector_policy vec { unspecified };
```

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

7 Parallel algorithms [parallel.alg]

7.1 Wavefront Application [parallel.alg.wavefront]

1 For the purposes of this subclause, an *evaluation* is a value computation or side effect of an expression, or an execution of a statement. Initialization of a temporary object is considered a subexpression of the expression that necessitates the temporary object.

2 An evaluation A *contains* an evaluation B if:

(2.1) — A and B are not potentially concurrent (C++17 §4.7.1); and

(2.2) — the start of A is the start of B or the start of A is sequenced before the start of B; and

(2.3) — the completion of B is the completion of A or the completion of B is sequenced before the completion of A.

[*Note:* This includes evaluations occurring in function invocations. — *end note*]

3 An evaluation A is *ordered before* an evaluation B if A is deterministically sequenced before B. [*Note:* If A is indeterminately sequenced with respect to B or A and B are unsequenced, then A is not ordered before B and B is not ordered before A. The ordered before relationship is transitive. — *end note*]

4 For an evaluation A ordered before an evaluation B, both contained in the same invocation of an element access function, A is a *vertical antecedent* of B if:

(4.1) — there exists an evaluation S such that:

(4.1.1) — S contains A, and

(4.1.2) — S contains all evaluations C (if any) such that A is ordered before C and C is ordered before B,

(4.1.3) — but S does not contain B, and

(4.2) — control reached B from A without executing any of the following:

(4.2.1) — a `goto` statement or `asm` declaration that jumps to a statement outside of S, or

(4.2.2) — a `switch` statement executed within S that transfers control into a substatement of a nested selection or iteration statement, or

(4.2.3) — a `throw` [*Note:* Even if caught — *end note*] , or

(4.2.4) — a `longjmp`.

[*Note:* Vertical antecedent is an irreflexive, antisymmetric, nontransitive relationship between two evaluations. Informally, A is a vertical antecedent of B if A is sequenced immediately before B or A is nested zero or more levels within a statement S that immediately precedes B. — *end note*]

5 In the following, X_i and X_j refer to evaluations of the same expression or statement contained in the application of an element access function corresponding to the i^{th} and j^{th} elements of the input sequence. [*Note:* There can be several evaluations X_k, Y_k, etc. of a single expression or statement in application k, for example, if the expression or statement appears in a loop within the element access function. — *end note*]

6 *Horizontally matched* is an equivalence relationship between two evaluations of the same expression. An evaluation B_i is *horizontally matched* with an evaluation B_j if:

(6.1) — both are the first evaluations in their respective applications of the element access function, or

(6.2) — there exist horizontally matched evaluations A_i and A_j that are vertical antecedents of evaluations B_i and B_j, respectively.

[*Note: Horizontally matched* establishes a theoretical lock-step relationship between evaluations in different applications of an element access function. — *end note*]

7 Let f be a function called for each argument list in a sequence of argument lists. *Wavefront application* of f requires that evaluation A_i be sequenced before evaluation B_j if $i < j$ and:

(7.1) — A_i is sequenced before some evaluation B_i and B_i is horizontally matched with B_j, or

(7.2) — A_i is horizontally matched with some evaluation A_j and A_j is sequenced before B_j.

[*Note: Wavefront application* guarantees that parallel applications i and j execute such that progress on application j never gets ahead of application i. — *end note*] [*Note:* The relationships between A_i and B_i and between A_j and B_j are *sequenced before*, not *vertical antecedent*. — *end note*]

7.2 Non-Numeric Parallel Algorithms [parallel.alg.ops]

7.2.1 Header <experimental/algorithm> synopsis [parallel.alg.ops.synopsis]

```
#include <algorithm>

namespace std::experimental {
inline namespace parallelism_v2 {
namespace execution {
  // 7.2.5, No vec
  template<class F>
    auto no_vec(F&& f) noexcept -> decltype(std::forward<F>(f)());

  // 7.2.6, Ordered update class
  template<class T>
    class ordered_update_t;

  // 7.2.7, Ordered update function template
  template<class T>
    ordered_update_t<T> ordered_update(T& ref) noexcept;
}

// Exposition only: Suppress template argument deduction.
template<class T> struct type_identity { using type = T; };
template<class T> using type_identity_t = typename type_identity<T>::type;

// 7.2.2, Reductions
template<class T, class BinaryOperation>
  unspecified reduction(T& var, const T& identity, BinaryOperation combiner);
template<class T>
  unspecified reduction_plus(T& var);
template<class T>
  unspecified reduction_multiplies(T& var);
template<class T>
  unspecified reduction_bit_and(T& var);
template<class T>
  unspecified reduction_bit_or(T& var);
template<class T>
  unspecified reduction_bit_xor(T& var);
template<class T>
  unspecified reduction_min(T& var);
```



```
template<class T>
  unspecified reduction_max(T& var);

// 7.2.3, Inductions
template<class T>
  unspecified induction(T&& var);
template<class T, class S>
  unspecified induction(T&& var, S stride);

// 7.2.4, For loop
template<class I, class... Rest>
  void for_loop(type_identity_t<I> start, I finish, Rest&&... rest);
template<class ExecutionPolicy,
         class I, class... Rest>
  void for_loop(ExecutionPolicy&& exec,
                type_identity_t<I> start, I finish, Rest&&... rest);
template<class I, class S, class... Rest>
  void for_loop_strided(type_identity_t<I> start, I finish,
                        S stride, Rest&&... rest);
template<class ExecutionPolicy,
         class I, class S, class... Rest>
  void for_loop_strided(ExecutionPolicy&& exec,
                        type_identity_t<I> start, I finish,
                        S stride, Rest&&... rest);
template<class I, class Size, class... Rest>
  void for_loop_n(I start, Size n, Rest&&... rest);
template<class ExecutionPolicy,
         class I, class Size, class... Rest>
  void for_loop_n(ExecutionPolicy&& exec,
                  I start, Size n, Rest&&... rest);
template<class I, class Size, class S, class... Rest>
  void for_loop_n_strided(I start, Size n, S stride, Rest&&... rest);
template<class ExecutionPolicy,
         class I, class Size, class S, class... Rest>
  void for_loop_n_strided(ExecutionPolicy&& exec,
                          I start, Size n, S stride, Rest&&... rest);
}
}
```


7.2.2 Reductions [parallel.alg.reductions]

1 Each of the function templates in this subclause (7.2.2) returns a *reduction object* of unspecified type having a *reduction value type* and encapsulating a *reduction identity* value for the reduction, a *combiner* function object, and a *live-out object* from which the initial value is obtained and into which the final value is stored.

2 An algorithm uses reduction objects by allocating an unspecified number of instances, known as *accumulators*, of the reduction value type. [*Note:* An implementation can, for example, allocate an accumulator for each thread in its private thread pool. — *end note*] Each accumulator is initialized with the object's reduction identity, except that the live-out object (which was initialized by the caller) comprises one of the accumulators. The algorithm passes a reference to an accumulator to each application of an element-access function, ensuring that no two concurrently executing invocations share the same accumulator. An accumulator can be shared between two applications that do not execute concurrently, but initialization is performed only once per accumulator.

3 Modifications to the accumulator by the application of element access functions accrue as partial results. At some point before the algorithm returns, the partial results are combined, two at a time, using the reduction

object's combiner operation until a single value remains, which is then assigned back to the live-out object. [*Note:* In order to produce useful results, modifications to the accumulator should be limited to commutative operations closely related to the combiner operation. For example if the combiner is `plus<T>`, incrementing the accumulator would be consistent with the combiner but doubling it or assigning to it would not. — *end note*]

```
template<class T, class BinaryOperation>
  unspecified reduction(T& var, const T& identity, BinaryOperation combiner);
```

4 *Requires:* `T` shall meet the requirements of `CopyConstructible` and `MoveAssignable`. The expression `var = combiner(var, var)` shall be well-formed.

5 *Returns:* A reduction object of unspecified type having reduction value type `T`, reduction identity `identity`, combiner function object `combiner`, and using the object referenced by `var` as its live-out object.

```
template<class T>
  unspecified reduction_plus(T& var);
template<class T>
  unspecified reduction_multiplies(T& var);
template<class T>
  unspecified reduction_bit_and(T& var);
template<class T>
  unspecified reduction_bit_or(T& var);
template<class T>
  unspecified reduction_bit_xor(T& var);
template<class T>
  unspecified reduction_min(T& var);
template<class T>
  unspecified reduction_max(T& var);
```


6 *Requires:* `T` shall meet the requirements of `CopyConstructible` and `MoveAssignable`.

7 *Returns:* A reduction object of unspecified type having reduction value type `T`, reduction identity and combiner operation as specified in Table 2 and using the object referenced by `var` as its live-out object.

Table 2 — Reduction identities and combiner operations

Function	Reduction Identity	Combiner Operation
reduction_plus	T()	x + y
reduction_multiplies	T(1)	x * y
reduction_bit_and	(~T())	x & y
reduction_bit_or	T()	x | y
reduction_bit_xor	T()	x ^ y
reduction_min	var	min(x, y)
reduction_max	var	max(x, y)

[*Example:* The following code updates each element of `y` and sets `s` to the sum of the squares.

```
extern int n;
extern float x[], y[], a;
float s = 0;
for_loop(execution::vec, 0, n,
  reduction(s, 0.0f, plus<>()),
  [&](int i, float& accum) {
    y[i] += a*x[i];
```

