Revision: 2015-09-14 N1966
Reply to: Clark Nelson

Programming languages — C — Extensions for parallel programming

This is the latest working draft of the CPLEX study group.

Contents

Page

Tables

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75% of the member bodies casting a vote.

In other circumstances, particularly when there is an urgent market requirement for such documents, a technical committee may decide to publish other types of normative document:

— an ISO Publicly Available Specification (ISO/PAS) represents an agreement between technical experts in an ISO working group and is accepted for publication if it is approved by more than 50% of the members of the parent committee casting a vote;

— an ISO Technical Specification (ISO/TS) represents an agreement between the members of a technical committee and is accepted for publication if it is approved by 2/3 of the members of the committee casting a vote.

An ISO/PAS or ISO/TS is reviewed every three years with a view to deciding whether it can be transformed into an International Standard.

ISO/TS *CPLEXTS* was prepared by Technical Committee ISO/IEC JTC1/SC22/WG14.[1)]

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

[1)] FYI: This is the only paragraph in the Foreword that has anything in it that's not just boilerplate.

Introduction

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights other than those mentioned above. ISO [and/or] IEC shall not be held responsible for identifying any or all such patent rights.

Programming languages — C — Extensions for parallel programming

1 Scope

The following are within the scope of this technical specification:

— Extensions to the C language to simplify writing a parallel program.

The following are outside the scope of this technical specification:

— Support for writing a concurrent program.

2 Normative references

The following normative documents contain provisions which, through reference in this text, constitute provisions of this technical specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this technical specification are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

ISO/IEC 9899:2011(E), *Programming languages — C*

ISO/IEC 14882:2014(E), *Programming languages — C++*

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

3.1
thread of execution
flow of control within a program, including a top-level statement or expression, and recursively including every function invocation it executes[2)]

3.2
OS thread
service provided by an operating system for executing multiple threads of execution concurrently

NOTE 1 There is typically significant overhead involved in creating a new OS thread.

3.3
thread
thread of execution, or OS thread

NOTE 1 This word, when used without qualification, is ambiguous.

3.4
execution agent
entity, such as an OS thread, that may execute a thread of execution in parallel with other execution agents[3)]

3.5
task
thread of execution within a program that can be correctly executed asynchronously with respect to independent threads of execution from the program

3.6
concurrent program
program that uses multiple concurrent interacting threads of execution, each with its own progress requirements

EXAMPLE 1 A program that has separate server and client threads is a concurrent program.

3.7
parallel program
program whose computation involves independent tasks, which may be distributed across multiple computational units to be executed simultaneously

NOTE 1 If sufficient computational resources are available, a parallel program may execute significantly faster than an otherwise equivalent serial program.

[2)] FYI: Adapted from the C++ standard.

[3)] FYI: Adapted from the C++ standard.

4 Document conventions

1 *[C++: Text that is specific to C++ is enclosed in square brackets and presented in oblique sans-serif type.]*

2 Definitions of terms and grammar non-terminals defined in the C *[C++: or C++]* standard are not duplicated in this document. Terms and grammar non-terminals defined in this document are referenced in the index. The "cplex_" prefix of library identifiers is omitted from the index entry.

3 According to the ISO editing directives, the use of footnotes "shall be kept to a minimum." Almost all of the footnotes in this document are not intended to survive to final publication. Most footnotes are classified by an abbreviation:

FYI: A point of information.

TODO: Calls attention to an area that needs more work.

DFEP: Departure from existing practice.

5 Task execution

1 A task is permitted to execute in either the invoking OS thread or an OS thread implicitly created by the implementation to support task execution. Independent tasks executing in the same OS thread are indeterminately sequenced with respect to one another. Independent tasks executing in different OS threads are unsequenced with respect to one another. [4)]

2 When execution of an independent task completes, execution *joins* with its parent task. The completion of a task synchronizes with the completion of the associated task block, or with the next execution of a sync statement within the associated task block.

[4)] TODO: The expectation is that there is a direct mapping from a thread managed by the C11 `thrd` functions to an OS thread.

6 Reduction types

6.1 Introduction

1 A *reduction type* describes a member object with a particular type, called the *proxied type*, and an associated combiner operation, along with other optional aspects, to support common parallel computations.

6.2 Reduction specifiers

Syntax

reduction-specifier:
  `_Reduction` *identifier*$_{opt}$ `{` *reduction-aspect-list* `}`
  `_Reduction` *identifier*

reduction-aspect-list:
  reduction-aspect
  reduction-aspect-list `,` *reduction-aspect*

reduction-aspect:
  `_Type :` *type-name*
  `_Combiner :` *combiner-operation*
  `_Initializer :` *initializer*
  `_Finalizer :` *constant-expression*
  `_Order :` *reduction-order-constraint*

combiner-operation:
  constant-expression
  builtin-combiner-operation

builtin-combiner-operation: one of
  `*= +=`
  `&= ^= |=`
  `_And _Or`
  `_Min _Max`
  `_Last`

reduction-order-constraint:
  `_Commutative`
  `_Associative`

Constraints

1 The type and combiner aspects shall be present in every reduction specifier. Any given kind of aspect shall not be present more than once in a reduction specifier.

2 The proxied type of a reduction shall be one of the following: an unqualified arithmetic type, a pointer to object type, or a complete structure or union type.

3 If the combiner is a constant expression, then it shall be an address constant referring to a

function taking two arguments, both of pointer-to-proxied type. If it is a compound assignment operator, then it shall be one that is usable with an lvalue of the proxied type.[5)] If it is `_And` or `_Or`, the proxied type shall be an integer type. If it is `_Min` or `_Max`, the proxied type shall be an arithmetic type.

4 The initializer of a reduction shall be suitable to initialize an object of the proxied type having static storage duration, or shall be an address constant referring to a function. If it is an address constant, it shall refer to a function taking one argument of pointer-to-proxied type.

5 The finalizer of a reduction shall be an address constant referring to a function taking one argument of pointer-to-proxied type.

Semantics

6 A reduction is a type containing a proxied member of an associated proxied type. Each concurrently-executing task that refers to an object of reduction type has its own distinct proxied member object (called its *view*) of the object.

NOTE 1 Thus, when they are used as intended, reduction objects can be updated from different tasks without causing data races.

7 At any point within a parallel computation, the value of a view reflects a sub-computation on the reduction object. At some point after the completion of a set of tasks, partial results are combined, two at a time, using the combiner operation of the reduction type to merge one view into another. The resulting value reflects the union of the sub-computations on the two original views.

8 A reduction object is *serially consistent* when no other task that could access it in parallel is executing. A serially consistent reduction object has a single view, called the *root view*, reflecting the entire set of computations on the reduction object since its creation.

9 If the combiner operation is a function pointer, the combination is performed by executing:

`(*` *combiner*`)(&` *into_view*`, &` *from_view*`);`

Otherwise the combination is performed according to Table 1. In all cases the object designated *into_view* is expected to be modified to reflect the combined sub-computations. The object designated *from_view* is unused after being combined with *into_view* except as the argument to the finalizer.

10 At some unspecified point before a task refers to a reduction object for the first time, the view used by the task is allocated, and initialized using the initializer of the reduction's type. If the initializer is a function pointer, the initialization is performed by executing:

`(*` *initializer*`)(&` *view*`);`

Otherwise, the view is initialized as if it were an object with static storage duration, using the specified initializer. If the initializer is not specified, and the specified combiner is in Table 2, the view is initialized with the corresponding value from Table 2. [6)]

[5)] For example, if the proxied type is a floating type, the operator shall not be `|=`.

[6)] TODO: It's not clear that what we have is complete considering dynamically-allocated reduction objects.

Table 1 – Combination method for built-in combiners

Specified combiner	Combination method
`*=`	*into_view* `*=` *from_view* `;`
`+=`	*into_view* `+=` *from_view* `;`
`&=`	*into_view* `&=` *from_view* `;`
`^=`	*into_view* `^=` *from_view* `;`
`|=`	*into_view* `|=` *from_view* `;`
`_And`	*into_view* `=` *into_view* `&&` *from_view* `;`
`_Or`	*into_view* `=` *into_view* `||` *from_view* `;`
`_Min`	`if (` *from_view* `<` *into_view* `)` *into_view* `=` *from_view* `;`
`_Max`	`if (` *from_view* `>` *into_view* `)` *into_view* `=` *from_view* `;`
`_Last`	*into_view* `=` *from_view* `;`

Table 2 – Default initializers for built-in combiners

Specified combiner	Default initializer
`*=`	the value 1, converted to the proxied type
`&=`	the bitwise complement of converting zero to the proxied type
`_And`	the value 1, converted to the proxied type
`_Min`	the maximum value representable by the proxied type
`_Max`	the minimum value representable by the proxied type

11 It is unspecified whether a view is created for a task that does not access a specific reduction object. A task's view of a reduction object can be shared with other tasks that do not execute concurrently; it is not necessary that each task have a distinct view. Any initializer function is invoked only once for each distinct view, regardless how many tasks share that view. The initializer aspect of a reduction type is not used to initialize the root view of a reduction object; for purposes of initialization, the root view behaves like a member of the reduction object.

12 If the type of a reduction object has a finalizer, after a view has been used as the *from_view* argument to the combiner operation, the view is finalized by executing:

`(*` *finalizer*`)(&` *view*`);`

The finalizer is not applied to the root view.

NOTE 2 A single view is never passed to concurrent invocations of the initializer, combiner operation, or finalizer of a reduction object.

13 Views are presented to the combiner operation in pairings that depend on the order aspect of the reduction type. In the following, for any point P at which the reduction object is serially consistent, let S represent the sequence of modifications that would be applied to the reduction object in the serialization of the program:

`_Commutative:` View combinations can be paired arbitrarily. The value of the root view at P reflects all of the operations in S, but applied in an unspecified order.

NOTE 3 Operations that are sensitive to operand order (e.g., string append) or to operation grouping (e.g., addition in the presence of overflow) might yield nondeterministic results that differ from the serialization.

`_Associative`: Views are presented to the combiner operation such that the values of *into_view* and *from_view* reflect consecutive subsequences of S, respectively called SL and SR. The value computed by the combiner operation (stored in *into_view*) reflects the concatenation of SL and SR, which comprises a contiguous subsequence of S. The value of the root view at P reflects all of the operations in S, but applied in unspecified groupings.

NOTE 4 Operations that are sensitive to operation grouping (e.g., addition in the presence of overflow) might yield nondeterministic results that differ from the serialization.

14 If the combiner aspect of a reduction type is `_Last`, the default for the order aspect is `_Associative`. Otherwise, the default for the order aspect is `_Commutative`. [7)]

15 Two reduction types declared in separate translation units are compatible if all of the following conditions are satisfied:

a) Neither is declared with a tag, or they are declared with the same tag.

b) Their proxied types are compatible.

c) Their combiner operations are the same (either the same builtin combiner operation or the same function).

d) Neither specifies a finalizer, or their finalizers are specified with equal values.

e) Their order aspects are specified or defaulted to be the same.

f) If either is specified with an initializer that is the address of a function, then the other is specified with an initializer that is the address of the same function; otherwise, corresponding scalar components of the proxied type are initialized with equal values, and in corresponding components with union type, members with compatible types are initialized.

6.3 Reduction conversions

1 An lvalue with reduction type is implicitly converted, through a run-time view lookup, to an lvalue with its corresponding proxied type. This conversion is suppressed if the address of the lvalue is taken in a context where the result is immediately converted, implicitly or explicitly, to a pointer to the original reduction type.

EXAMPLE 1 Consider this code:

```
_Reduction int_add { _Type: int, _Combiner: += };
_Reduction int_add x, y;
x = y; // int assignment: both operands converted by view lookup
void f(_Reduction int_add *, int *);
f(&x, &y);  // view lookup performed on y, but not on x
int *pi = &x;
f(pi, &x);  // error
```

The last line of the example is an error because `pi`, as an expression, is not taking the address of a reduction-converted lvalue. The expression that takes that address is in the previous line. The reduction lvalue conversion can be suppressed during translation, but not necessarily reversed during execution. As a further example:

[7)] DFEP: Neither OpenMP nor Cilk supports specifying the order constraint for reduction. OpenMP reductions provide only the guarantees of `_Commutative`; Cilk reductions provide the guarantees of `_Associative`.

```
f((_Reduction int_add *)pi, &x);
```

This is not an error, but the first argument passed to the function need not point to the reduction object, so undefined behavior results if it used as if it did.

6.4 Integration with the C standard

Change paragraph 7 of subclause 6.2.1 “Scopes of identifiers”:

> Structure, union, reduction, and enumeration tags have scope that begins just after the appearance of the tag in a type specifier that declares the tag. ...

Change the list item of paragraph 1 of subclause 6.2.3 “Name spaces of identifiers”:

— the *tags* of structures, unions, reductions, and enumerations (disambiguated by following any of the keywords `struct`, `union`, `_Reduction`, or `enum`);

Add a new item to the list in paragraph 20 of subclause 6.2.5 “Types”:

— A *reduction type* describes a member object with a particular type, called the *proxied type*, and an associated combiner operation, along with other optional aspects, to support common parallel computations.

Change the grammar rule in subclause 6.4.1 “Keywords”, by adding new alternatives:

keyword: one of
...
`_Task_parallel`
`_Block`
`_Spawn`
`_Sync`
`_Call`
`_Reduction`

Add a new item to the list in paragraph 1 of subclause 6.5.16.1 “Simple assignment”:

— the left operand has atomic, qualified, or unqualified pointer to some reduction type, and the right operand expression is the taking of the address of some object having a qualified or unqualified version of the same reduction type (whose type is therefore a pointer to the reduction’s proxied type), and the type pointed to by the left has all the qualifiers of the type pointed to by the right;

Change the grammar rule in subclause 6.7.2 “Type specifiers”, by adding a new alternative:

type-specifier:
...
reduction-specifier

Change paragraphs 2 through 6 of subclause 6.7.2.3 “Tags”:

> Where two declarations that use the same tag declare the same type, they shall both use the same choice of `struct`, `union`, `_Reduction`, or `enum`.

A type specifier of the form

`_Reduction` *identifier*

without a reduction aspect list, or

`enum` *identifier*

without an enumerator list shall only appear after the type it specifies is complete.

All declarations of structure, union, reduction, or enumerated types that have the same scope and use the same tag declare the same type.

Two declarations of structure, union, reduction, or enumerated types which are in different scopes or use different tags declare distinct types. Each declaration of a structure, union, reduction, or enumerated type which does not include a tag declares a distinct type.

A type specifier of the form

struct-or-union $identifier_{opt}$ `{` *struct-declaration-list* `}`

or

`_Reduction` $identifier_{opt}$ `{` *reduction-aspect-list* `}`

or

`enum` $identifier_{opt}$ `{` *enumerator-list* `}`

or

`enum` $identifier_{opt}$ `{` *enumerator-list* `, }`

declares a structure, union, reduction, or enumerated type. The list defines the structure content, union content, reduction content, or enumeration content. ...

Change paragraph 9 of 6.7.2.3 “Tags”:

If a type specifier of the form

struct-or-union identifier

or

`_Reduction` *identifier*

or

`enum` *identifier*

occurs other than as part of one of the above forms, and a declaration of the identifier as a tag is visible, then it specifies the same type as that other declaration, and does not redeclare the tag.

Change paragraph 1 of subclause 6.7.6.3 “Function declarations (including prototypes)”:

> A function declarator shall not specify a return type that is a function type or an array type or a reduction type.

Add a new paragraph following paragraph 8 of subclause 6.7.6.3 “Function declarations (including prototypes)”:

> A declaration of a parameter as a reduction type shall be adjusted to be a pointer to the same reduction type.

Add a new paragraph following paragraph 8 of subclause 6.7.9 “Initializers”:

> Any initializer for an object of reduction type initializes its root view. [8]

[C++:

6.5 Integration with the C++ standard

Add new entries to table 3 in subclause 2.11 “Keywords”:[9]

Change paragraph 3 of subclause 3.3.2 “Point of declaration”:

Changes to subclause 3.4.4 “Elaborated type specifiers”:

Add a new item to the list in paragraph 1 of subclause 3.9.2 “Compound types”:

Add a new paragraph following paragraph 3 of subclause 4.10 “Pointer conversions”:

Change the grammar rule in paragraph 1 of subclause 7.1.6 “Type specifiers”:

Change paragraphs 2 and 3 of subclause 7.1.6.3 “Elaborated type specifiers”:

Change paragraph 5 of subclause 8.3.5 “Functions”:

Change to subclause 8.5 “Initializers”:

]

[8] TODO: At the time it was incorporated into this document, the reduction proposal added: ”If an object of reduction type is not initialized explicitly, then the root view is initialized as every other view is initialized.” But this conflicted with the direction CPLEX approved (I think).

[9] TODO: This section is currently just an outline, and needs to be filled in.

7 Captures

7.1 Introduction

1 A spawn capture allows a spawn statement to make a copy of a variable prior to the start of asynchronous execution. A reduction capture allows a task block or parallel loop to temporarily associate a reduction object with an existing object, to simplify parallel computation of a reduction.

7.2 Spawn captures

Syntax

spawn-capture:
    `_Capture` (*spawn-capture-list*)

spawn-capture-list:
    spawn-capture-item
    spawn-capture-list , *spawn-capture-item*

spawn-capture-item:
    identifier
    identifier `=` *expression*

Constraints

1 If no expression is present in a spawn capture item, the identifier shall be a name that is already in scope at the beginning of the spawn capture item, and the effective expression is taken to be the same as the identifier. Otherwise, the effective expression is the expression in the spawn capture item.

2 The effective expression shall have complete object type.

Semantics

3 Each spawn capture item declares a new object named by the item's identifier, having automatic storage duration. The type of the declared object is that of the effective expression. The scope of the name extends from the end of the spawn capture item until the end of the spawn statement with which it is associated.

4 The declared object is initialized with the value of the effective expression. The initialization of the declared object occurs before asynchronous execution of the spawned compound statement.

5 Change the first sentence of paragraph 3 of subclause 6.3.2.1: [10)]

> Except when it is the effective expression in a spawn capture item, or is the operand of the `sizeof` operator, the `_Alignof` operator, or the unary `&` operator, or is a string literal used to initialize an array, an expression that has type "array of type" is converted

[10)] TODO: This change is necessary only to be able to capture/copy a whole array. Do we want to allow that? OpenMP seems to support it.

to an expression with type "pointer to type" that points to the initial element of the array object and is not an lvalue. ...

7.3 Reduction captures

Syntax

reduction-capture:
  `_Reduction` `(` *reduction-capture-list* `)`

reduction-capture-list:
  reduction-capture-item
  reduction-capture-list **,** *reduction-capture-item*

reduction-capture-item:
  declaration-specifiers declarator
  declaration-specifiers declarator `:` *expression*

Constraints

1 The declaration specifiers in a reduction capture item shall specify a reduction type, and shall not specify static or thread storage duration.

2 If no expression is present in a reduction capture item, the identifier in the declarator shall be a name that is already in scope at the beginning of the reduction capture item, and the effective expression is taken to be the same as the identifier. Otherwise, the effective expression is the expression in the reduction capture item.

3 The effective expression shall be a modifiable lvalue, and shall have a type that is assignable to the proxied type of the item's reduction type. [11)]

Semantics

4 Each reduction capture item declares a new object, named by the identifier in the declarator. The type of the object is the type specified by the declaration specifiers. The scope of the name extends from the end of the reduction capture item until the end of the task block or loop with which it is associated.

5 Before execution of the task block or loop, the new reduction object is initialized with the value of the object designated by the effective expression. Upon completion of the task block or loop, the value of the reduction object is assigned back to the object designated by the effective expression.

6 Change the first sentence of paragraph 2 of subclause 6.3.2.1:

> Except when it is the expression in a reduction capture item, or is the operand of the `sizeof` operator, the `_Alignof` operator, the unary `&` operator, the `++` operator, the `--` operator, or the left operand of the `.` operator or an assignment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue); this is called *lvalue conversion*. ...

[11)] TODO: Would it be better to require the effective expression type to match the proxied type?

8 Counted loops

8.1 Introduction

1 A *counted loop* is a `for` statement *[C++: or range-based* `for` *statement]* that is required to satisfy additional constraints. The purpose of these constraints is to ensure that the loop's iteration count can be computed before the loop body is executed.

2 There shall be no `return`, `break`, `goto` or `switch` statement that might transfer control into or out of a counted loop.

8.2 Constraints on a counted `for` statement

8.2.1 Introduction

1 The syntax of a `for` statement includes three *control clauses* between parentheses, separated by semicolons. The first of these is called the initialization clause; the second is called the condition clause or controlling expression; the third is called the *loop-increment*.

2 When a constraint limits the form of an expression, parentheses are allowed around the expression or any required subexpression.

8.2.2 Constraints on the form of the control clauses

1 *[C++: The condition shall be an expression.*

NOTE 1 A condition with declaration form is useful in a context where a value carries more information than just whether it is zero or nonzero. This is not believed to be useful in a counted loop.

]

2 The controlling expression shall be a comparison expression with one of the following forms:[12)]

relational-expression `<` *shift-expression*
relational-expression `>` *shift-expression*
relational-expression `<=` *shift-expression*
relational-expression `>=` *shift-expression*
equality-expression `!=` *relational-expression*

3 Exactly one of the operands of the comparison operator shall be an identifier designating an induction variable, as described below. This induction variable is known as the *control variable*. The operand that is not the control variable is called the *limit expression*. *[C++: Any implicit conversion applied to that operand is not considered part of the limit expression.]*

4 The loop-increment shall be an expression with the following form:[13)]

loop-increment:
  single-increment
  loop-increment `,` *single-increment*

[12)]DFEP: OpenMP does not (yet) allow comparison with `!=`.

[13)]DFEP: OpenMP and "classic" Cilk allow only a single induction variable: the loop control variable. Allowing multiple induction variables is implemented in Intel's compiler.

single-increment:
 identifier `++`
 identifier `--`
 `++` *identifier*
 `--` *identifier*
 identifier `+=` *initializer-clause*
 identifier `-=` *initializer-clause*
 identifier `=` *identifier* `+` *multiplicative-expression*
 identifier `=` *identifier* `-` *multiplicative-expression*
 identifier `=` *additive-expression* `+` *identifier*

5 *[C++: Each comma in the grammar of loop-increment shall represent a use of the built-in comma operator.]* The identifier in each grammatical alternative for single-increment names an *induction variable*. If *identifier* occurs twice in a grammatical alternative for *single-increment*, the same variable shall be named by both occurrences. If a grammatical alternative for *single-increment* contains a subexpression that is not an identifier for the induction variable, that is called the *stride expression* for that induction variable.

6 An induction variable shall not be designated by more than one *single-increment*.

NOTE 2 The control variable is identified by considering the loop's condition and loop-increment together. If exactly one operand of the condition comparison is a variable, it is the control variable, and must be incremented. If both operands of the condition comparison are variables, only one is allowed to be incremented; that one is the control variable. It is an error if neither operand of the condition comparison is a variable.

NOTE 3 There is no additional constraint on the form of the initialization clause of a counted `for` loop.[14)]

8.2.3 Other statically checkable constraints

1 Each induction variable shall have unqualified integer, *[C++: enumeration, copy-constructible class,]* or pointer type, and shall have automatic storage duration.

2 Each stride expression shall have integer *[C++: or enumeration]* type.

3 The *iteration count* is computed according to Table 3. If the controlling expression uses a relational operator, and is true when the value of the control variable is less than (respectively, greater than) the value of the limit expression, then the operator in the single-increment for the control variable shall not be `--` (respectively, `++`). The iteration count is computed after the loop initialization is performed, and before the control variable is modified by the loop. *[C++: The iteration count expression shall be well-formed.]*

4 The type of the difference between the limit expression and the control variable is the *subtraction type*, *[C++: which shall be integral. When the condition operation is !=, (limit)-(var) and (var)-(limit) shall have the same type.]* Each stride expression shall be convertible to the subtraction type. *[C++: The loop odr-uses whatever operator-functions are selected to compute these differences.]*

[C++:

5 *For each induction variable V, one of the expressions from Table 4 shall be well-formed, depending on*

[14)] DFEP: OpenMP and "classic" Cilk require that the control variable be initialized. This relaxation is implemented in Intel's compiler.

Table 3 – Method of computing the iteration count

Form of condition	**Form of single-increment**			
	id* ++** **++ *id	***id* --** **-- *id***	***id* += *stride*** ***id* = *id* + *stride*** ***id* = *stride* + *id***	***id* -= *stride*** ***id* = *id* - *stride***
id* < *lim ***lim* > *id***	$((lim) - (id))$	ERROR	$((lim) - (id) - 1)/$ $(stride) + 1$	$((lim) - (id) - 1)/$ $(stride) + 1$
id* > *lim ***lim* < *id***	ERROR	$((id) - (lim))$	$((id) - (lim) - 1)/$ $-(stride) + 1$	$((id) - (lim) - 1)/$ $-(stride) + 1$
id* <= *lim ***lim* >= *id***	$((lim) - (id))$ $+1$	ERROR	$((lim) - (id))/$ $(stride) + 1$	$((lim) - (id))/$ $(stride) + 1$
id* >= *lim ***lim* <= *id***	ERROR	$((id) - (lim))$ $+1$	$((id) - (lim))/$ $-(stride) + 1$	$((id) - (lim))/$ $-(stride) + 1$
id* != *lim ***lim* != *id***	$((lim) - (id)$	$((id) - (lim))$	$((stride) < 0)$? $((id) - (lim) - 1)/$ $-(stride) + 1$: $((lim) - (id) - 1)/$ $(stride) + 1$	$((stride) < 0)$? $((lim) - (id) - 1)/$ $-(stride) + 1$: $((id) - (lim) - 1)/$ $(stride) + 1$

Legend:

Name	**In the form of an expression**	**In the iteration count expression**
id	The name of the control variable.	An expression with the type and value of the control variable.
lim	The limit expression.	An expression with the type and value of the limit expression.
stride	The stride expression.	An expression with the type and value of the stride expression for the control variable.

Table 4 – Method of advancing an induction variable

Single-increment operator	**Expression**
`++ += +`	V `+=` X
`-- -= -`	V `-=` X

the operator used in its single-increment. In the table, X stands for some expression with the same type as the subtraction type. The loop odr-uses whatever `operator+=` *and* `operator-=` *functions are selected by these expressions.]*

8.2.4 Dynamic constraints

1 If an induction variable is modified within the loop other than as the side effect of its single-increment operation, the behavior of the program is undefined.

[C++: If evaluation of the iteration count, or a call to a required `operator+=` *or* `operator-=` *function, terminates with an exception, the behavior of the program is undefined.]*

2 If X and Y are values of the control variable that occur in consecutive evaluations of the loop condition in the serialization, then the behavior is undefined if $((limit) - X) - ((limit) - Y)$,

evaluated in infinite integer precision, does not equal the stride.

NOTE 1 In other words, the control variable must obey the rules of normal arithmetic. Unsigned wraparound is not allowed.

3 If the condition expression is true on entry to the loop, then the behavior is undefined if the computed iteration count is not greater than zero. If the computed iteration count is not representable as a value of type `unsigned long long`, the behavior is undefined.

8.2.5 Evaluation relaxations

1 The stride expressions shall not be evaluated if the iteration count is zero; otherwise, the stride and limit expressions are evaluated exactly once.[15)]

2 Within each iteration of the loop body, the name of each induction variable refers to a local object, as if the name were declared as an object within the body of the loop, with automatic storage duration and with the type of the original object. *[C++: If the loop body throws an exception that is not caught within the same iteration of the loop, the behavior is undefined, unless otherwise specified.]*

[C++:

8.3 Constraints on a counted range-based `for` statement

1 In a counted range-based `for` statement ([stmt.ranged] 6.5.4), the type of the `__begin` variable, as determined from the begin-expr, shall satisfy the requirements of a random access iterator.

NOTE 1 Intel has not yet implemented support for a parallel range-based `for` statement.

]

[15)] DFEP: Neither OpenMP nor Cilk specifies how many times these expressions must be evaluated.

9 Parallel loops

1 A *parallel loop* is a `for` statement with loop qualifiers. The grammar of the iteration statement (6.8.5, paragraph 1) is modified to read:

iteration-statement:
  `while (` *expression* `)` *statement*
  `do` *statement* `while (` *expression* `) ;`
  loop-qualifiers$_{opt}$ `for (` *expression*$_{opt}$ `;` *expression*$_{opt}$ `;` *expression*$_{opt}$ `)` *statement*
  loop-qualifiers$_{opt}$ `for (` *declaration expression*$_{opt}$ `;` *expression*$_{opt}$ `)` *statement*

[C++: The grammar of iteration-statement (6.5 [stmt.iter], paragraph 1) is modified to read:

iteration-statement:
  `while (` *expression* `)` *statement*
  `do` *statement* `while (` *expression* `) ;`
  loop-qualifiers$_{opt}$ `for (` *for-init-statement condition*$_{opt}$ `;` *expression*$_{opt}$ `)` *statement*
  loop-qualifiers$_{opt}$ `for (` *for-range-declaration* `:` *for-range-initializer* `)` *statement*

]

2 The following rules are added to the grammar:

loop-qualifiers:
  `_Task_parallel` *qualifier-clauses*$_{opt}$

qualifier-clauses:
  loop-parameters qualifier-clauses$_{opt}$
  reduction-capture qualifier-clauses$_{opt}$

loop-parameters:
  `_Options (` *expression* `)`

3 A parallel loop is a counted loop, and shall satisfy all the constraints of a counted loop.

4 In a parallel loop with the `_Task_parallel` loop qualifier, each iteration is executed as a task, independent of all other iterations of that execution of the loop. At the end of the loop, execution joins with all of these tasks.

5 If loop parameters are specified as part of the loop qualifiers, the contained expression shall have type "pointer to `cplex_loop_params_t`", as defined in header `<cplex.h>`.[16)]

6 The *serialization* of a parallel loop is obtained by deleting the loop qualifiers from the loop.

[16)] DFEP: This syntax for specifying tuning parameters for a loop is a CPLEX invention.

10 Task statements

10.1 Introduction

1 The grammar of a statement (6.8, paragraph 1) *[C++: (clause 6, paragraph 1)]* is modified to add task-statement as a new alternative.

Syntax

task-statement:
task-block-statement
task-spawn-statement
task-sync-statement
task-call-statement

10.2 The task block statement

Syntax

task-block-statement:
`_Task_parallel _Block` *reduction-capture*$_{opt}$ *compound-statement*

Constraints

1 There shall be no `return`, `break`, `goto` or `switch` statement that might transfer control into or out of a task block statement.

Semantics

2 Defines a task block, within which tasks can be spawned. At the end of the contained compound statement, execution joins with all child tasks spawned directly or indirectly within the compound statement. [17)]

3 For a given statement, the *associated task block* is defined as follows. For a statement within a task spawn statement, there is no associated task block, except within a nested task block statement or parallel loop. For a statement within a task block statement or parallel loop, the associated task block is the smallest enclosing task block statement or parallel loop. Otherwise, for a statement within the body of a function declared with the spawning function specifier, the associated task block is the same as it was at the point of the task spawning call statement that invoked the spawning function. For a statement in any other context, there is no associated task block.

NOTE 1 Task blocks can be nested lexically and/or dynamically. Determination of the associated task block is a hybrid process: lexically within a function, and dynamically across calls to spawning functions.[18)] Code designated for execution in another thread by means other than a task statement (e.g. using `thrd_create`) is not part of any task block.

[17)] TODO: What about C++ EH?

[18)] DFEP: In Cilk, this determination can be done entirely lexically. In OpenMP, this determination can be done entirely dynamically.

10.3 The task spawn statement

Syntax

task-spawn-statement:
`_Task_parallel _Spawn` *spawn-capture*$_{opt}$ *compound-statement*

Constraints

1 A task spawn statement shall have an associated task block.

2 There shall be no `return`, `break`, `goto` or `switch` statement that might transfer control into or out of a task spawn statement.

Semantics

3 The contained compound statement is executed as a task, independent of the continued execution of the associated task block.

10.4 The task sync statement

Syntax

task-sync-statement:
`_Task_parallel _Sync ;`

Constraints

1 A task sync statement shall have an associated task block.

Semantics

2 Execution joins with all child tasks of the associated task block of the task sync statement.

10.5 The task spawning call statement

Syntax

task-call-statement:
`_Task_parallel _Call` *expression-statement*

Constraints

1 A task spawning call statement shall have an associated task block. [19)]

Semantics

2 The contained expression statement is executed normally. Any called spawning function is allowed to spawn tasks; any such tasks are associated with the associated task block of the task

[19)] TODO: Using the same keyword pair for the statement prefix and the function specifier introduces a case where three tokens of lookahead (and hopefully no more) are sometimes needed to disambiguate a declaration from a statement. Is this what we want?

spawning call statement, and are independent of the statements of the task block following the task spawning call statement.

NOTE 1 A call to a task spawning function need not be the “outermost” operation of the expression statement. A task spawning call statement might invoke more than one spawning function, or might invoke none.

10.6 The spawning function specifier

Syntax

1 A new alternative is added to the grammar of function specifier (6.7.4 paragraph 1):

function-specifier:
```
    _Task_parallel _Call
```

Constraints

2 If a spawning function specifier appears on any declaration of a function, it shall appear on every declaration of that function. A function declared with a spawning function specifier shall be called only from a task spawning call statement. [20)]

[20)]TODO: Is a spawning function specifier part of the function's type? If so, are function-pointer conversions allowed?

11 Parallel loop hint parameters `<cplex.h>`

11.1 Introduction

1 The header `<cplex.h>` defines several types and several macros.

2 The `cplex_loop_params_t` type is a structure type with an unspecified number of members for specifying parameters for tuning hints for a parallel loop. A program whose output depends on the value specified for any tuning hint parameter is not considered a correct program.

NOTE 1 There is no guarantee that setting any tuning hint parameter will improve the performance of the program.

3 The `cplex_sched_kind_t` type is an enumerated type with at least the following enumeration constants, each with nonzero value:

```
cplex_sched_static
cplex_sched_dynamic
cplex_sched_guided
```

4 The `cplex_workload_t` type is an enumerated type with at least the following enumeration constants, each with nonzero value:

```
cplex_workload_balanced
cplex_workload_unbalanced
```

5 The `cplex_affinity_t` type is an enumerated type with at least the following enumeration constants, each with nonzero value:

```
cplex_affinity_close
cplex_affinity_spread
```

6 When an object of type `cplex_loop_params_t` is used as the loop parameter of a parallel loop, the loop is described as being associated with the object. If the associated object is modified during the execution of the loop, the behavior is undefined. When executing a parallel loop associated with an object of type `cplex_loop_params_t`, for any parameter for which the corresponding member has the value zero, an unspecified default value is used.

7 Each parameter is represented by a pair of macros: one to set the value of the parameter in the parameter block, and one to get the value of the parameter from the parameter block.

NOTE 2 Because these methods are specified as macros, not functions, taking the address of any of them need not be supported. However, an implementation is also free to provide functions with these names.

EXAMPLE 1 Hint parameters for a parallel loop can be specified as follows:

```
#include <cplex.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    cplex_loop_params_t hints = { 0 };
    if (argc > 1) {
        cplex_set_num_threads(&hints, atoi(argv[1]));
```

```
    }
    cplex_set_chunk_size(&hints, 1000);
    _Task_parallel _Options(&hints) for (long i = 0; i < 1000000; i++) {
        do_something_with(i);
    }
}
```

11.2 The num_threads parameter

Synopsis

```
#include <cplex.h>
void cplex_set_num_threads(cplex_loop_params_t *hints, int num_threads);
int cplex_get_num_threads(cplex_loop_params_t *hints);
```

Description

1 The `cplex_set_num_threads` macro sets to `num_threads` the recommended number of execution agents to be used to execute the iterations of a parallel loop associated with the object pointed to by `hints`.

11.3 The chunk_size parameter

Synopsis

```
#include <cplex.h>
void cplex_set_chunk_size(cplex_loop_params_t *hints, int chunk_size);
int cplex_get_chunk_size(cplex_loop_params_t *hints);
```

Description

1 The `cplex_set_chunk_size` macro sets to `chunk_size` the recommended maximum number of iterations of a parallel loop associated with the object pointed to by `hints` to be grouped together to be executed sequentially in a single thread of execution.

11.4 The schedule_kind parameter

Synopsis

```
#include <cplex.h>
void cplex_set_schedule_kind(cplex_loop_params_t *hints,
                               cplex_sched_kind_t kind);
cplex_sched_kind_t cplex_get_schedule_kind(cplex_loop_params_t *hints);
```

Description

1 The `cplex_set_schedule_kind` macro sets to `kind` the recommended scheduling algorithm for a parallel loop associated with the object pointed to by `hints`.

NOTE 1 Setting the `schedule_kind` parameter to a particular value may (but need not) select the corresponding OpenMP loop-scheduling algorithm.

11.5 The `workload_balance` parameter

Synopsis

```
#include <cplex.h>
void cplex_set_workload_balance(cplex_loop_params_t *hints,
                                cplex_workload_t kind);
cplex_workload_t cplex_get_workload_balance(cplex_loop_params_t *hints);
```

Description

1 The `cplex_set_workload_balance` macro sets to `kind` the workload-balancing characteristic for a parallel loop associated with the object pointed to by `hints`.

2 For a loop with a balanced workload, each iteration should be assumed to execute in approximately the same amount of time. A loop with an unbalanced workload should be assumed to have iterations taking widely varying amounts of time.

NOTE 1 This parameter is semantically a statement about the associated loop, whereas the `schedule_kind` parameter is semantically a request to the implementation. Setting this parameter to `cplex_workload_balanced` may have an effect similar to setting the schedule to `cplex_schedule_static`. Setting this parameter to `cplex_workload_unbalanced` may have an effect similar to setting the schedule to `cplex_schedule_dynamic` or `cplex_schedule_guided`.

11.6 The `affinity` parameter

Synopsis

```
#include <cplex.h>
void cplex_set_affinity(cplex_loop_params_t *hints, cplex_affinity_t kind);
cplex_affinity_t cplex_get_affinity(cplex_loop_params_t *hints);
```

Description

1 The `cplex_set_affinity` macro sets to `kind` the recommended affinity for a parallel loop associated with the object pointed to by `hints`.

2 The affinity of a loop indicates whether the loop benefits from being executed by co-located hardware threads, or whether performance is likely to improve if the software threads are spread over multiple cores.

Bibliography

[1] *Intel© Cilk™ Plus Language Extension Specification,* Intel Corporation: <https://www.cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_1.2.htm>

[2] *OpenMP Application Program Interface,* OpenMP Architecture Review Board: <http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf>

Index