Intel® Cilk™ Plus Language Specification

Part 1 of the Intel® Cilk™ Plus Language Specification version 0.9

Document Number: 324396-001US

Revision: 1.0

World Wide Web: http://www.intel.com

More information about Intel® Cilk™ Plus can be found at the following web site:
http://cilk.com

Feedback on the specification is encouraged and welcome, please send to:
cilkfeedback@intel.com

Disclaimer and Legal Information

Contents

1 Intel® Cilk™ Plus Language Extension Specification

This document is part of the Intel® Cilk™ Plus Language Specification version 0.9. The language specification comprises a set of technical specifications describing the language and the run-time support for the language. Together, these documents provide the detail needed to implement a compliant compiler. At this time the language specification contains these parts:

- Part 1. The Intel® Cilk™ Plus Language Specification, document number 324396-001US.
- Part 2. The Intel® Cilk™ Plus Application Binary Interface, document number 324512-001US.

This document defines the Intel® Cilk™ Plus extension to C and C++. The language extension is supported by a run time user mode work stealing task scheduler which is not directly exposed to the application programmer. However, some of the semantics of the language and some of the guarantees provided require specific behavior of the task scheduler. The programmer visible parts of the language include the following constructs:

- Three keywords - _Cilk_spawn, _Cilk_sync and _Cilk_for, to express tasking
- Hyper variables, that provide local views to global variables
- Array notations
- Elemental functions
- Pragma simd

An implementation of the language may take advantage of all parallelism resources available in the hardware. On a typical CPU, these include at least multiple cores and vector units. Some of the language constructs, e.g. _Cilk_spawn, utilize only core parallelism, some, e.g. #pragma simd, utilize only vector parallelism, and some, e.g. elemental functions, utilize both. The defined behavior of every deterministic Cilk program is the same as the behavior of a similar C or C++ program known as the "serialization." While execution of a C or C++ program may be considered as a linear sequence of statements, execution of a tasking program is in general a directed acyclic graph. Parallel control flow may yield a new kind of undefined behavior, a "data race," whereby parts of the program that may execute in parallel access the same memory location in an indeterminate order, with at least one of the accesses being a write access. In addition, throwing an exception may result in code being executed that would not have been executed in a serial execution.

2 Keywords for Tasking

Cilk Plus adds the following new keywords:

- `_Cilk_sync`
- `_Cilk_spawn`
- `_Cilk_for`

A program that uses these keywords other than as defined in the grammar extension below is ill-formed.

2.1 Keyword Aliases

The header `<cilk/cilk.h>` shall define the following aliases for the Cilk keywords:

```
# define cilk_spawn _Cilk_spawn
# define cilk_sync  _Cilk_sync
# define cilk_for   _Cilk_for
```

2.2 Grammar

The three keywords are used in the following new productions:

jump-statement:
    `_Cilk_sync` *;*

The call production of the grammar is modified to permit the keyword _Cilk_spawn before the expression denoting the function to be called:

postfix-expression:
    `_Cilk_spawn`*opt* *postfix-expression (expression-list*opt *)*

Consecutive `_Cilk_spawn` tokens are not permitted. The postfix-expression following `_Cilk_spawn` is called a "spawned function." The spawned function may be a normal function call, a member-function call, or the function-call (parentheses) operator of a function object (functor) or a call to a lambda expression. Overloaded operators other than the parentheses operator may be spawned only by using the function-call notation (e.g., `operator+(arg1,arg2)`). There shall be no more than one `_Cilk_spawn` within a full expression. A function that contains a spawn statement is called a "spawning function."

A program is considered ill formed if the `_Cilk_spawn` form of this expression:

- does not appear as the entire body of an expression statement,
- Does not appear as the entire right hand side of an assignment expression that is the entire body of an expression statement, or
- Does not appear as the entire *initializer-clause* in a simple declaration..

(A `_Cilk_spawn` expression may be permitted in more contexts in the future.)

A statement with a `_Cilk_spawn` on the right hand side of an assignment or declaration is called an "assignment spawn" or "initializer spawn", respectively and the object assigned or initialized by the spawn is called the "receiver."

The iteration-statement is extended by adding another form of `for` loop:

cilk-for-init-decl:
    decl-specifier-seq init-declarator

grainsize-pragma:
    `# pragma cilk grainsize =` *expression*

iteration-statement:
    `_Cilk_for` *(cilk-for-init-decl ; condition; expression)*
        statement
    `_Cilk_for` *(assignment-expression ; condition ; expression)*
        statement

2.3 Semantics

2.3.1 Tasking Execution Model

A "strand" is a serially-executed sequence of instructions that does not contain a spawn point or sync point (as defined below). At a spawn point, one strand (the initial strand) ends and two strands (the new strands) begin. The initial strand runs in series with each of the new strands but the new strands may run in parallel with each other. At a sync point, one or more strands (the initial strands) end and one strand (the new strand) begins. The initial strands may run in parallel with one another but each of the initial strands runs in series with the new strand. A single strand can be subdivided into a sequence of shorter strands in any manner that is convenient for modeling the computation. A "maximal strand" is one that cannot be included in a longer strand.

The strands in an execution of a program form a directed acyclic graph (DAG) in which spawn points and sync points comprise the vertices and the strands comprise the directed edges, with time defining the direction of each edge. (In an alternative DAG representation, sometimes seen in the literature, the strands comprise the vertices and the dependencies between the strands comprise the edges.)

2.3.2 Serialization rule

The behavior of a deterministic Intel® Cilk™ Plus program is defined in terms of its "serialization" as defined below. If the serialization has undefined behavior, the Intel® Cilk™ Plus program also has undefined behavior.

The strands in an execution of a program are ordered according to the order of execution of the equivalent code in the program's serialization. Given two strands, the "earlier" strand is defined as the strands that would execute first in the serial execution of the same program with the same inputs, even though the two strands may execute in either order or concurrently in the actual parallel execution. Similarly, the terms "earliest," "latest," and "later" are used to designate strands according to their serial ordering. The terms "left," "leftmost," "right," and "rightmost" are equivalent to "ealier," "earliest," "later," and "latest," respectively.

2.3.2.1 The serialization of a pure C or C++ program is itself.

If a C or C++ program has defined behavior and does not use the tasking keywords or library functions, it is a Cilk Plus program with the same defined behavior.

2.3.2.2 The serializations of `_Cilk_spawn` and `_Cilk_sync` are empty.

If a Cilk Plus program has defined deterministic behavior, then that behavior is the same as the behavior of the C or C++ program derived from the original by removing all instances of the keywords `_Cilk_spawn`, and `_Cilk_sync`

2.3.2.3 The serialization of `_Cilk_for` is `for`.

If a Cilk Plus program has defined deterministic behavior, then that behavior is the same as the behavior of the C or C++ program derived from the original by replacing each instance of the `_Cilk_for` keyword with `for`.

2.4 Cilk blocks

A Cilk block is a region of the program subject to special rules. Cilk blocks may be nested. The body of a nested Cilk block is not part of the outer Cilk block. Cilk blocks never partially overlap. The body of a spawning function is a Cilk block. A `_Cilk_for` statement is a Cilk block and the body of the `_Cilk_for` loop is a (nested) Cilk block.

Every Cilk block includes an implicit `_Cilk_sync` executed on exit from the block, including abnormal exit due to an exception. Destructors for automatic objects with scope ending at the end of the Cilk block are invoked before the implicit `_Cilk_sync`. The receiver is assigned or initialized to the return value before executing the implicit `_Cilk_sync` at the end of a function. An implicit or explicit `_Cilk_sync` within a nested Cilk block will synchronize with `_Cilk_spawn` statements only within that Cilk block, and not with `_Cilk_spawn` statements in the surrounding Cilk block.

The scope of a label defined in a Cilk block is limited to that Cilk block.

Programmer note: Therefore, `goto` may not be used to enter or exit a Cilk block.

2.5 _Cilk_for Loops

2.5.1 Syntactic constraints

To simplify the grammar, some restrictions on `_Cilk_for` loops are stated here in text form. The three items inside parentheses in the grammar, separated by semicolons, are the "initialization", "condition", and "increment".

A program that contains a `return`, `break`, or `goto` statement that would transfer control into or out of a `_Cilk_for` loop is ill-formed.

The initialization must declare or initialize a single variable called the “control variable.” In C mode only, the control variable may be previously declared, but must be reinitialized, i.e., assigned, in the initialization item. In C++ mode, the control variable must be declared and initialized within the initialization item of the `_Cilk_for` loop. No storage class may be specified for the variable within the initialization item. The variable must have integral, pointer, or class type. The variable may not be `const` or `volatile`. The variable must be initialized. Initialization may be explicit, using assignment or constructor syntax, or implicit via a nontrivial default constructor. Within each iteration of the loop body, the control variable is considered a unique variable whose address is no longer valid when the iteration completes. If the control variable is declared before the loop initialization, then the address of the variable at the end of the loop is the same as the address of the variable before the loop initialization and the value of the control variable is the same as for the serialization of the program.

The condition shall have one of the following two forms:

var OP shift-expression
shift-expression OP var

where *var* is the control variable, optionally enclosed in parentheses. The operator denoted "*OP*" shall be one of !=, <=, <, >=, or >. The *shift-expression* that is not the control variable is called the "loop limit."

The loop increment shall have one of the following forms, where "var" is the loop control variable, optionally enclosed in parentheses, and "incr" is a *logical-or-expression* with integral or enum type. The table indicates the "stride" corresponding to the syntactic form.

syntax	**stride**
`++var`	`+1`
`var++`	`+1`
`--var`	`-1`
`var--`	`-1`
`var += incr`	`+(incr)`
`var -= incr`	`-(incr)`

The notion of "stride" exists for exposition only and does not need to be computed. In particular, for the case of `var -= incr`, a program may be well formed even if `incr` is unsigned.

2.5.2 Requirements on types and operators

The type of `var` shall be copy constructible. (For the purpose of specification, all C types are considered copy constructible.) The initialization, condition, and increment parts of a `_Cilk_for` must be defined such that the total number of iterations (loop count) can be determined before beginning the loop execution. Specifically, the parts of the `_Cilk_for` loop shall meet all of the semantic requirements of the corresponding serial for statement. In addition, a `_Cilk_for` adds the following requirements on the types for `var`, `limit`, and `stride` (and by extension `incr`), depending on the syntactic form of the condition. (In the following table, `first` is the value of `var` immediately after initialization.)

condition syntax	**requirements**	**loop count**

`var < limit`
`limit > var`
	`(limit)-(first)` shall be well-formed and shall yield an integral `difference_type`; `stride` > 0	`((limit)-(first))/stride`

`var > limit`
 	
`limit < var`
	`(first)-(limit)` shall be well-formed and shall yield an integral `difference_type;` `stride < 0`	`((first)-(limit))/-stride`

`var <= limit`
 	
`limit >= var`
	`(limit)-(first)` shall be well-formed and shall yield an integral `difference_type;` `stride` > 0	`((limit)-(first)+1)/stride`

`var >= limit`
 	
`limit <= var`
	`(first)-(limit)` shall be well-formed and shall yield an integral `difference_type;` `stride < 0`	`((first)-(limit)+1)/-stride`

`var != limit`
 	
`limit != var`
	both `(limit)-(first)` and `(first)-(limit)` shall be well-formed and yield the same integral `difference_type`; `stride != 0`	if `stride` is positive, then `((limit)-(first))/stride`, else `((first)-(limit))/-stride`

The `incr` expression shall have integral or enum type. If the loop increment uses operator ++ or +=, the expression `(var) += (difference_type)(incr)` shall be well-formed. If the loop increment uses operator `--` or `-=`, the expression `(var) -= (difference_type)(incr)` shall be well-formed. The loop is a *use* of the required operator function.

2.5.3 Dynamic constraints

If the `stride` does not meet the requirements in the table above, the behavior is undefined. If this condition can be determined statically, the compiler is encouraged (but not required) to issue a warning. (Note that the incorrect loop might occur in an unexecuted branch, e.g., of a function template, and thus should not cause a compilation failure in all cases.)

If the control variable is modified other than as a side effect of evaluating the loop increment expression, the behavior of the program is undefined.

If X and Y are values of var that occur in consecutive evaluations of the loop condition in the serialization, then `((limit)-X)-((limit)-Y)` evaluated in infinite integer precision shall equal `stride`. If the condition expression is true on entry to the loop, then the loop count shall be non-negative.

Programmer note: Unsigned wraparound is not allowed.

The increment and limit expressions may be evaluated fewer times than in the serialization. If different evaluations of the same expression yield different values, the behavior of the program is undefined.

The copy constructor for the control variable may be executed more times than in the serialization.

If evaluation of the increment or limit expression, or a required `operator+=` or `operator-=` throws an exception, the behavior of the program is undefined.

If the loop body throws an exception that is not caught within the same iteration of the loop, it is unspecified which other loop iterations execute. If multiple loop iterations throw exceptions that are not caught in the loop body, the `_Cilk_for` statement throws the exception that would have occurred first in the serialization of the program.

2.5.4 Grainsize pragma

A `_Cilk_for` *iteration-statement* may optionally be preceded by a *grainsize-pragma*. The grainsize pragma must immediately precede a `_Cilk_for` loop and may not appear anywhere else in a program, except that other pragmas that appertain to the `_Cilk_for` loop may appear between the *grainsize-pragma* and the `_Cilk_for` loop. The expression in the grainsize pragma must evaluate to a type convertible to `signed long`. The presence of the pragma provides a hint to the runtime specifying the number of serial iterations desired in each chunk of the parallel loop. The grainsize expression is evaluated at runtime. If there is no grainsize pragma, or if the grainsize evaluates to `0`, then the runtime will pick a grainsize using its own internal heuristics. If the grainsize evaluates to a negative value, the behavior is unspecified. (The meaning of negative grainsizes is reserved for future extensions.) The grainsize pragma applies only to the `_Cilk_for` statement that immediately follows it – the grain sizes for other `_Cilk_for` statements are not affected.

2.6 Spawn

The `_Cilk_spawn` keyword suggests to the implementation that an executed statement or part of a statement may be run in parallel with following statements. A consequence of this parallelism is that the program may exhibit undefined behavior not present in the serialization. Execution of a `_Cilk_spawn` keyword is called a “spawn.” Execution of a `_Cilk_sync` statement is called a “sync.” A statement that contains a spawn is called a “spawning statement.”

The "following sync" of a `_Cilk_spawn` refers to the next `_Cilk_sync` executed (dynamically, not lexically) in the same Cilk block. Which spawn the sync follows is implied from context. The following sync may be the implicit `_Cilk_sync` at the end of a Cilk block.

A "spawn point" is a C sequence point at which a control flow fork is considered to have taken place. Any operations within the spawning expression that are not required by the C/C++ standards to be sequenced after the spawn point shall be executed before the spawn point. The strand that begins at the statement immediately following the spawning statement (in execution order) is called the "continuation" of the spawn. The sequence of operations within the spawning statement that are sequenced after the spawn point comprise the "child" of the spawn. The scheduler may execute the child and the continuation in parallel. Informally, the "parent" is the Cilk block containing the initial strand, the spawning statements, and their continuations but excluding the children of all of the spawns. The children of the spawns within a single Cilk block are "siblings" of one another.

The spawn points associated with different spawning statements are as follows:

- The body of a `_Cilk_for` loop is a spawning statement with spawn point at the end of the loop condition test.
- An expression statement containing a single `_Cilk_spawn` has a spawn point at the sequence point at the call to the spawned function. Any unnamed temporary variables created prior to the spawn point are not destroyed until after the spawn point (i.e., the destructors are invoked in the child).
- A declaration statement in which an identifier is initialized or assigned with a result of a function call that is being spawned has a spawn point at the sequence point at the call to the spawned function. A declaration statement may consist of multiple comma separated declarations. Each of them may or may not have a spawn, and there can be at most one spawn per expression. The conversion of the function return value, if necessary, and the assignment or initialization of the receiver takes place after the spawn point (i.e., in the child). Any unnamed temporary variables created prior to the spawn point are not destroyed until after the spawn point (i.e., their destructors are invoked in the child).

For example, in the following two statements:

```
x[g()] = _Cilk_spawn f(a + b);
a++;
```

The call to function `f` is the spawn point and the statement `a++;` is the continuation. The expression, `a + b`, the initialization of the temporary variable holding the result of `a+b`, and the evaluation of `x[g()]` take place before the spawn point. The execution of `f`, the assignment to `x[g()]`, and the destruction of the temporary variable holding `a+b` take place in the child.

If a statement is followed by an implicit sync, that sync is the spawn continuation.

Programmer note: The sequencing may be more clear if

```
  x[g()] = _Cilk_spawn f(a + b);
```

is considered to mean

```
  {
    // Evaluate arguments and receiver address before spawn point
    T tmp = a + b;   // T is the type of a + b
    U &r = x[g()];   // U is the type of x[0]
    _Cilk_spawn  { r = f(tmp); tmp.~T(); }
  }
```

A setjmp/longjmp call pair within the same Cilk block has undefined behavior if a spawn or sync is executed between the setjmp and the longjmp. A setjmp/longjmp call pair that crosses a Cilk block boundary has undefined behavior. A `goto` statement is not permitted to enter or exit a Cilk block.

2.7 Sync

A sync statement indicates that all children of the current Cilk block must finish executing before execution may continue within the Cilk block. The new strand coming out of the `_Cilk_sync` is not running in parallel with any child strands, but may still be running in parallel with parent and sibling strands (other children of the calling function).

There is an implicit sync at the end of every Cilk block. If a spawning statement appears within a try block, a sync is implicitly executed at the end of that try block, as if the body of the try were a Cilk block. If a Cilk block has no children at the time of a sync, then the sync has no observable effect. (The compiler may elide an explicit or implicit sync if it can statically determine that the sync will have no observable effect.)

Programmer note: Because implicit syncs follow destructors, writing `_Cilk_sync` at the end of a function may produce a different effect than the implicit sync. In particular, if an assignment spawn or initializer spawn is used to modify a local variable, the function will generally need an explicit `_Cilk_sync` to avoid a race between assignment to the local variable by the spawned function and destruction of the local variable by the parent function.

2.8 Exceptions

There is an implicit `_Cilk_sync` before a `throw`, after the exception object has been constructed.

A function that is not synched may continue executing until the exception is recognized at a later sync.

When several exceptions are pending and not yet caught, later exception objects (in the serial execution order of the program) are destructed in an unspecified order before the earliest exception is caught.

3 Hyperobjects

3.1 Description

Cilk defines a category of objects called “hyperobjects”. Hyperobjects allow thread-safe access to shared objects by giving each parallel strand a separate instance of the object.

Parallel code uses a hyperobject by performing a “hyperobject lookup” operation. The hyperobject lookup returns a reference to an object, called a “view,” that is guaranteed not to be shared with any other active strands in the program. The sequencing of a hyperobject lookup within an expression is not specified. The runtime system creates a view when needed, using callback functions provided by the hyperobject type. When strands synchronize, the hyperobject views are merged into a single view, using another callback function provided by the hyperobject type.

The view of a hyperobject visible to a program may change at any spawn or sync (including the implicit spawns and syncs within a `_Cilk_for` loop). The identity (address) of the view does not change within a single strand. The view of a given hyperobject visible within a given strand is said to be *associated* with that view. A hyperobject has the same view before the first spawn within a Cilk block as after a sync within the same Cilk block, even though the thread ID may not be the same (i.e., hyperobject views are not tied to threads). A hyperobject has the same view upon entering and leaving a `_Cilk_for` loop and within the first iteration (at least) of the `_Cilk_for` loop. A special view is associated with a hyperobject when the hyperobject is initially created. This special view is called the “leftmost view” or “earliest view” because it is always visible to the leftmost (earliest) descendent in the depth-first, left-to-right traversal of the program’s spawn tree. The leftmost view is given an initial value when the hyperobject is created.

Programmer note: If two expressions compute the same address for a view, then they have not been scheduled in parallel. This property yields one of the simplest ways by which a program can observe the runtime behavior of the scheduler.

Implementation note: An implementation can optimize hyperobject lookups by performing them only when a view has (or might have) changed. This optimization can be facilitated by attaching implementation-specific attributes to the hyperobject creation, lookup, and/or destruction operations.

3.2 Reducers

The vast majority of hyperobjects belong to a category known as "reducers." Each reducer type provides a `reduce` callback operation that merges two views in a manner specific to the reducer. For a pair of views V_1 and V_2, the result of calling `reduce(`V_1, V_2`)` is notated as $V_1 \otimes V_2$. Each reducer also provides an `identity` callback operation that initializes a new view.

The `reduce` callback for a "classical" reducer implements an operation $\otimes$ such that $(a \otimes b) \otimes c == a \otimes (b \otimes c)$ (i.e., $\otimes$ is associative). The view-initialization callback for such a reducer sets the view to an identity value I such that $I \otimes v == v$ and $v \otimes I == v$ for any value v of `value_type`. Given an associative $\otimes$ and an identity, I, the triplet (`value_type`, $\otimes$, I) describes a mathematical *monoid*. For example, (`int,+,0`) is a monoid, as is (list,concatenate,empty). If each individual view, R, of a classical reducer is modified using only expressions that are equivalent to $R \leftarrow R \otimes v$ (where v is of `value_type`), then the reducer computes the same value in the parallel program as would be computed in the serialization of the program. (In actuality, the "$\otimes$" in the expression "$R \leftarrow R \otimes v$" can represent a set of mutually-associative operations. For example, += and -= are mutually associative.) For example, a spawned function or `_Cilk_for` body can append items onto the view of a list reducer with monoid (list,concatenate,empty). At the end of the parallel section of code, the reducer's view contains the same list items in the same order as would be generated in a serial execution of the same code.

Given a set of strands entering a sync, $S_1,S_2,S_3,\ldots S_n$, associated with views $V_1,V_2,V_3,\ldots V_n$, respectively such that S_i is earlier in the serial ordering than S_{i+1}, a single view, W, emerges from the sync with value $W \leftarrow V_1 \otimes V_2 \otimes V_3 \otimes \ldots V_n$, such that the left-to-right order is maintained but the grouping (associativity) of the operations is unspecified. The timing of this "reduction" is unspecified – in particular, subsequences typically will be computed asynchronously as child tasks complete. Every view except the one emerging from the sync is destroyed after the merge. If any of the strands does not have an associated view, then the invocation of the `reduce` callback function can be elided (i.e., the missing view is treated as an identity).

A strand is never associated with more than one view for a given reducer, but multiple strands can be associated with the same view if those strands are not scheduled in parallel (at run time). Specifically, for a given reducer, the association of a strand to a view of the reducer obeys the following rules:

- The strand that initializes the reducer is associated with the leftmost view.
- If two strands execute in series (i.e., both strands are part of a larger strand), then both are associated with the same view.
- The child strand of a spawn is associated with the same view as the strand that entered the spawn.
- If the continuation strand of a spawn is scheduled in parallel with the child, then the continuation strand is associated with a new view,

initialized using `identity`. The implementation may create the new view at any time up until the first hyperobject lookup following the spawn. If the continuation strand does not perform a hyperobject lookup, then the implementation is not required to create a view for that strand.

- If the continuation strand of a spawn is *not* scheduled in parallel with the child strand (i.e., the child and the continuation execute in series), then the continuation strand is associated with the same view as the child strand.
- The strand that emerges from a sync is associated with the same view as the leftmost strand entering the sync.

Even before the final reduction, the leftmost view of a reducer will contain the same value as in the serial execution. Other views, however, will contain partial values that are different from the serial execution.

If ⊗ is not associative or if `identity` does not yield a true identity value then the result of a set of reductions will be non-deterministic (i.e., it will vary based on runtime scheduling). Such "non-classical" reducers are nevertheless occasionally useful. Note that, for a classical reducer, the ⊗ operator needs to be associative, but not commutative.

3.3 Hyperobjects in C++

3.3.1 C++ hyperobject syntax

The syntax described here is the syntax used in the Intel products. Intel is considering a different syntax for future, either in addition to or instead of the syntax described below.

At present, reducers are the only kind of hyperobject supported. In C++, every reducer hyperobject has a hyperobject type, which is an instantiation of the `cilk::reducer` class template. The `cilk::reducer` class template has a single template type parameter, `Monoid`, which must be a class type.

To define a reducer, a program defines a monoid class with public members representing the monoid, (*T*, ⊗, *identity*) as follows:

`value_type`	is a typedef for *T*
`reduce(left,right)`	evaluate '*left = *left ⊗ *right'
`identity(p)`	construct *identity* object at `p`
`destroy(p)`	call the destructor on the object at `p`
`allocate(size)`	return a pointer to `size` bytes of raw memory
`deallocate(p)`	deallocate the raw memory at `p`

The arguments, `left`, `right`, and `p` are of type pointer-to-`value_type` and the argument, `size`, is of type `std::size_t`. If any of the above functions do not modify the state of the monoid (most monoids carry no state), then those functions may be declared `static` or `const`. The monoid type may derive from an instantiation of `cilk::monoid_base<T>`, which defines `value_type` and provides default implementations for `identity`, `destroy`, `allocate`, and `deallocate`. The derived class needs to define `reduce` and override only those functions for which the default is incorrect.

For a given monoid, *M*, the type `cilk::reducer<M>` defines a hyperobject type. The `cilk::reducer` class template provides constructors, a destructor, and (`const` and non-`const` versions of) `value_type& operator()` and `value_type& view()`, both of which return a reference to the current view.

A hyperobject is created by defining an instance of `cilk::reducer<M>`:

```
cilk::reducer<M> hv(args...);
```

Where *args* is a list of `M::value_type` constructor arguments used to initialize the leftmost view of `hv`. A hyperobject lookup is performed by invoking the member function, `view()` or member `operator()` on the hyperobject, as in the following examples:

```
hv.view().append(elem);
hv().append(elem);
```

In these examples, `append` is an operation to be applied to the current view of `hv`, and is presumably consistent with the associative operation defined in the monoid, `M`.

Modifying a hyperobject view in a way that is not consistent with the associative operation in the monoid can lead to subtle bugs. For example, addition is not associative with multiplication, so performing a multiplication on the view of a summing reducer will almost certainly produce incorrect results. To prevent this kind of error, it is common to wrap reducers in proxy classes that expose only the valid associative operations. All of the reducers included in the standard reducer library have such wrappers.

3.3.2 C++ hyperobject behavior

An object of type `M::value_type` is constructed by the `reducer` constructor. This object is called the initial view or leftmost view of the hyperobject. When a hyperobject goes out of scope, the destructor is called on the leftmost view. It is unspecified whether `M::allocate` and `M::deallocate` are called to allocate and deallocate the leftmost view (they are not called in the current Intel implementation).

The implementation may create a view at any spawn that has been scheduled in parallel, or may lazily defer creation until the first access within a strand. The implementation creates a view by calling `M::allocate` followed by `M::identity`. (This is in addition to the initial view created by construction of the hyperobject.) The calls to `M::allocate` and `M::identity` are part of the strand for the purpose of establishing the absence of a data race.

At any sync or at the end of any spawned (child) function, the runtime may merge two views by calling `M::reduce(`*left,right*`)`, where *right* is the earliest remaining view that is later than *left*. The `M::reduce` function is expected to store the merged result in the *left* view. After the merge, the runtime destroys the *right* view by calling `M::destroy` followed by `M::deallocate`. Every view except the leftmost view is passed exactly once as the second argument to `reduce`. The calls to `M::reduce`, `M::destroy` and `M::deallocate` happen after completion of both of the strands that formerly owned the left and right views.

If a monoid member function executes a hyperobject lookup (directly or through a function call), the behavior of the program is undefined.

For purposes of establishing the absence of a data race, a hyperobject view is considered a distinct object in each parallel strand. A hyperobject lookup is considered a read of the hyperobject.

3.4 Hyperobjects in C

3.4.1 C hyperobject syntax

The syntax described here is the syntax used in the Intel products. Intel is considering a different syntax for future, either in addition to or instead of the syntax described below.

The C mechanism for defining and using hyperobjects depends on a small number of typedefs and preprocessor macros provided in the Cilk library. C does not have the template capabilities of C++ and thus has a less abstract hyperobject syntax. Unlike C++, each C hyperobject variable is unique – there is no named type that unites similar hyperobjects. There is, however, an implicit “hyperobject type” defined by the operations that comprise the hyperobjects’ monoid. The provided macros facilitate creating reducer variables, which are the only type of hyperobject currently supported. The terms “reducer” and “hyperobject” are used interchangeably in this section.

To define a C reducer, the program defines three functions representing operations on a monoid (*T*, ⊗, *identity*):

```
void T_reduce(void* r, void* left, void* right);
void T_identity(void* r, void* view);
void T_destroy(void* r, void* view);
```

The names of these functions are for illustration purposes only and must be chosen, as usual, to avoid conflicts with other identifiers. The purposes of these functions are as follows:

`T_reduce`	Evaluate *(T*) left = *(T*) left ⊗ *(T*) right
`T_identity`	Initialize a T value to *identity*
`T_destroy`	Clean up (destroy) a T value

The `r` argument to each of these functions is a pointer to the actual reducer variable and is usually ignored. Since most C types do not require cleanup on destruction, `T_destroy` often does nothing. As a convenience, the Cilk library makes this common implementation available as a library function, `__cilkrts_hyperobject_noop_destroy`.

A reducer, `hv`, is defined and given an initial value, `init`, using the `CILK_C_DECLARE_REDUCER` and `CILK_C_INIT_REDUCER` macros as follows:

```
CILK_C_DECLARE_REDUCER(T) hv =
    CILK_C_INIT_REDUCER(T_identity, T_reduce, T_destroy,
                        init);
```

The `init` expression is used to initialize the leftmost reducer view. The `CILK_C_DECLARE_REDUCER` macro defines a `struct` and can be used in a `typedef` or `extern` declaration as well:

```
extern CILK_C_DECLARE_REDUCER(T) hv;
```

The `CILK_C_INIT_REDUCER` expands to a static initializer for a hyperobject of any type. After initialization, the leftmost view of the reducer is available as `hv.value`.

If a reducer is local to a function, it must be registered before first use using the `CILK_C_REGISTER_REDUCER` macro and unregistered after its last use using the `CILK_C_UNREGISTER_REDUCER` macro:

```
CILK_C_REGISTER_REDUCER(hv);
/* use hv here */
CILK_C_UNREGISTER_REDUCER(hv);
```

For the purpose of registration and unregistration, “first use” and “last use” are defined with respect to the serialization of the program. The reducer view immediately before unregistration must be the same (have the same address) as the reducer view immediately after registration. In practice, this means that any spawns after the registration have been synced before the unregistration and that no spawns before the registration have been synced before the unregistration. Registration and unregistration are optional for reducers declared in global scope. The `value` member of the reducer continues to be available after unregistration, but a hyperobject lookup on an unregistered reducer results in undefined behavior unless the reducer is registered again.

A hyperobject lookup is performed using the `REDUCER_VIEW` macro:

```
REDUCER_VIEW(hv) += expr;
```

As in the case of a C++ reducer, modifying a reducer other than through the correct associative operations can cause bugs. Unfortunately, C does not have sufficient abstraction mechanisms to prevent this kind of error. Nevertheless, the Cilk library provides wrapper macros to simplify the declaration and initialization, though not the safety, of library-provided reducers in C. For example, you can define and initialize a summing reducer this way:

```
CILK_C_DECLARE_REDUCER(long) hv =
    REDUCER_OPADD_INIT(long, 0);
```

A C reducer can be declared, defined, and accessed within C++ code, but a C++ reducer cannot be used within C code.

3.4.2 C hyperobject behavior

The macro `CILK_C_DECLARE_REDUCER(`*`T`*`)` defines a `struct` with a data member of type *T*, `value`. The macro `CILK_C_INIT_REDUCER(`*`I`*`,`*`R`*`,`*`D`*`,`*`V`*`)` expands to a *braced-init-list* appropriate for initializing a variable, *`hv`*, of structure type declared with `CILK_C_DECLARE_REDUCER(`*`T`*`)` such that *`hv`*, can be recognized by the runtime system as a C reducer with value type *T*, identity function *I*, reduction function *R*, destroy function *D*, and initial value *V*.

Invoking `CILK_C_REGISTER_REDUCER(`*`hv`*`)` makes a call into the runtime system that registers *`hv`*`.value` as the initial, or leftmost, view of the C hyperobject *`hv`*. The macro `CILK_C_UNREGISTER_REDUCER(`*`hv`*`)` makes a call into the runtime system that removes hyperobject *`hv`* from the runtime system's internal map. Attempting to access *`hv`* after it has been unregistered will result in undefined behavior. If a hyperobject is never registered, the leftmost view will be associated with the program strand before the very first spawn in the program and will follow the leftmost branch of the execution DAG. This association is typically useful only for hyperobjects in global scope.

The implementation may create a view at any spawn that has been scheduled in parallel, or may lazily defer creation until the first access within a strand. The implementation creates a view by allocating it with `malloc`, then calling the identity function specified in the reducer initialization. (This is in addition to the initial view created by construction of the reducer.) The call to the identity function is part of the strand for the purpose of establishing the absence of a data race.

At any sync or at the end of any spawned (child) function, the runtime may merge two views by calling the reduction function (specified in the reducer initialization) on the values *left* and *right*, where *right* is the earliest remaining view that is later than *left*. The reduction function is expected to store the merged result in the *left* view. After the merge, the runtime destroys the *right* view by calling the destroy function for the hyperobject, then deallocates it using `free`. Every view except the leftmost view is passed exactly once as the second argument the reduction function. The calls to reduction and destroy functions happen after completion of both of the strands that formerly owned the left and right views.

If a monoid function executes a hyperobject lookup, the behavior of the program is undefined.

For purposes of establishing the absence of a data race, a hyperobject view is considered a distinct object in each parallel strand. A hyperobject lookup is considered a read of the hyperobject.

4 Array notations

4.1 Description

This section provides a specification for the array notation portion of the Intel® Cilk™ Plus language extension. For brevity, this portion will sometime be referred to as CEAN (C/C++ Extension for Array Notation). CEAN is intended to allow users to directly express high level parallel vector array operations in their programs. This assists the compiler in performing data dependence analysis, vectorization, and auto-parallelization. From the users' point of view, they will see more predictable vectorization, improved performance and better hardware resource utilization. The CEAN extension is based on the standard C/C++ languages with some restrictions, plus features that are designed for easy expression of array operations and simplified parallel mapping of elemental functions over independent input/output streams.

4.2 Data Types

4.2.1 Scalar Data Types

CEAN supports the following basic scalar data types defined in C and C++:

- char, unsigned char (uchar), short, unsigned short (ushort), int, unsigned int (uint), long, unsigned long (ulong), float, double, size_t, complex

A program that uses unsupported data types in array expressions is ill formed.

4.2.2 Compound Data Types

4.2.2.1 Arrays

CEAN uses standard C/C++ arrays as a base type for data-parallel operations. This includes C99 variable-length arrays (VLAs). CEAN introduces a *section operator* ":" (colon) which allows a user to express data-parallel operations over multiple elements in an array. The section operator is described fully in the next section, Operations.

4.2.2.2 Structures

CEAN allows users to perform data-parallel operations on any array type, including arrays of structures and arrays of C++ objects. Examples are found in the next section, Operations.

4.3 Operations

4.3.1 The Section Operator

The section operator ":" selects multiple array elements for a data-parallel operation.

The general syntax is as follows:

```
   <array base>[<lower bound>:<length>:<stride>][<lower
bound>:<length>:<stride>]....
```

Each *subscript triplet* lower_bound:length:stride represents a range of length values, starting at lower_bound and separated by a distance of stride: *<lower bound>, <lower bound + <stride>>, …, <lower bound> + (<length> – 1) * <stride>.*

The stride is optional and defaults to 1. (The expression is still termed a "triplet".) If the length is less than 1, the array section is undefined. The entire triplet can also be expressed as a single colon ":" which is shorthand for "the entire array dimension". If this shorthand is used, the array base type must have a declared size (which can be variable in the case of C99).

The stride can be negative, indicating an operation on the array elements lower in index than the lower bound. If the stride is 0, the expression is undefined.

Examples of valid multidimensional expressions are: "A[:][:]", "A[1:5:2][:]", "A[1:5][2:4]".

An *array section* is an array expression where one or more elements in the subscript list are subscript triplets.

For example, A[0:3][0:4] refers to 12 elements in the two-dimensional array A, starting at row 0, column 0, and ending at row 2, column 3.

A[0:2:3] refers to elements 0 and 3 of the one-dimensional array A.

A[:] refers to the entire array A.

Multidimensional array notation requires the array base type to have a declared size, as in standard C/C++. This size can be variable in the case of C99.

Examples of section expressions:

```
int *p;
int A[n][m];
p[:] = ... // not valid. p has no declared size.
A[:][:] = ... // The entire 2D array A.
p[10][10] = ... // not valid. p has no declared size.
p[1:5] = ... // p[1],p[2], ... p[5].
```

Figure 1. Example section expressions.

The *shape* of an array section is defined as an *n*-tuple of pairs: $((length_0, stride_0),(length_1, stride_1),\ldots,(length_{n-1}, stride_{n-1}))$ where *n* is the number of dimensions in the array section.

Every array expression has a *rank* given by the number of dimensions with subscript triplets, if any (which is different than the number of dimensions of the array definition itself.) For example, A[3:4][0:10], A[3][0:10], A[3:4][0], A[:][:], and A[3][0] are rank 2, 1, 1, 2, and 0 respectively. Every subscript triplet of the array section has a *relative rank*, defined as its ordinal number among the other triplets, from left to right, starting at 1. For example, in the array sections: A[1][0:10][0], A[0:10][1][2], A[2][x][0:10], the subscript triplet "0:10" has the relative rank 2, 1, and 3 respectively.

An array section may not be used inside the conditional tests in an "if", "for", or "while" statement. These restrictions may be relaxed in the future.

4.3.2 Operations on Array Sections

4.3.2.1 Assignment

An array section assignment is a parallel operation that applies to every element of the array section on the left-hand side. The right-hand side of the assignment is evaluated before the assignment is performed. The expression on the left-hand side of the assignment cannot affect the expression on the right-hand side. The compiler will allocate temporary storage to ensure this is the case, if the left-hand and right-hand sides overlap.

For example, in the assignment:

```
A[4:3] = A[3:3];
```

the 3 elements A[3], A[4], A[5] are copied to A[4], A[5], A[6] respectively. The compiler will allocate temporary storage to ensure that the writes to A[4] and A[5] do not interfere with the read operations of the same locations.

The bases of the various array sections in an assignment are allowed to have different dimension sizes, as long as they contain the same number of dimensions.

The array section operator cannot be used in a subscript expression where the subscript operator has been overloaded with a C++ function.

The array expression on the left-hand side of an assignment defines an array context where the expressions on the right hand side are evaluated. The array context has a rank (and its triplets also have a relative rank) defined as the rank of the expression on the left-hand side. The right-hand side expression must have the same rank and size as the array context. A rank 0 scalar value is duplicated to fill the array context.

Figure 2 shows examples of well-formed and ill-formed array section assignments.

```
// Copy elements 10->19 in A to elements 0->9 in B.
B[0:10] = A[10:10];

// Error. Triplets 0:10 and 0:100 are not the same size.
B[0:10] = A[0:100];

// y is assumed to be equal to x. Elements 10->10+x-1 in A are copied to
// elements 0->x-1 in B.
B[0:x] = A[10:y];

// Temporary storage is allocated so that the "move semantics" are
// preserved, and the original values of A[0:10] end up in A[1:10]. In
// other words, the copy operation does not interfere with itself.
A[1:10] = A[0:10];

// Transpose row 0, columns 0-9 of A, into column 0, rows 0-9 of B.
B[0:10][0] = A[0][0:10];

// Copy the specified array slice in the 2nd and 3rd dimensions of A into
// the 1st and 4th dimensions of B.
B[0:10][0][0][0:5] = A[3][0:10][0:5][5]

// Error. The corresponding triplets with the same relative rank
// (0:9, 0:5) and (0:5, 0:9) do not have the same number of elements.
B[0:9][0:5] = A[0:5][0:9];

// OK since the triplets on both sides have the same number of elements.
// The 5 elements in A are scattered to the even-index elements in B
// (0,2,4...8). The values of the odd-index elements (1,3,5...7) are not
```

Figure 2. Section assignments.

4.3.2.2 Arithmetic Operations

The following arithmetic operations can be applied to array sections:

```
+, -, *, /, %, <, ==, >, <=, !=, >=, ++, --,
|, &, ^, &&, ||, !, -(unary), +(unary)
```

Figure 3. Valid arithmetic operations.

When applied to an array section or sections, the C/C++ arithmetic operators are applied to each element of the array section(s). The precedence of operations is the same as with scalar C/C++ operators, as are the restrictions on data types (for example, '%' can only be applied to integers). The order of application among elements is unspecified. Note that multiplication is performed element-wise and does not correspond to traditional vector/matrix multiplication. The conditional operators (<,==,>) are applied element-wise in arithmetic expressions, and result in 0 (false) or 1 (true) values as in standard C/C++.

Figure 4 (next page) shows examples of operator usage.

```
// Set all elements of A to 1.0.
A[:] = 1.0;

// Set 3x3 shape in A to the scalar value B[0].
A[0:3][0:3] = B[0];

// Error: the number of dimensions (rank) must match,
// or be equal to 0.
A[0:2][0:2] = B[0:2];

// Element-wise addition of all elements in A and B, resulting in C.
C[:] = A[:] + B[:];

// Element-wise multiplication of all elements in A and B,
// resulting in C      .
C[:] = A[:] * B[:];

// Add elements 10->19 from A with elements 0->9 from B and place in
// elements 20->29 in C.
C[20:10] = A[10:10] + B[0:10];

// Element-wise addition of the first 10 elements in column 2 of A and
// column 3 of B, resulting in column 0 of C.
C[0:10][0] = A[0:10][2] + B[0:10][3];

// Add one to elements 1->10 of A, resulting in temporary storage,
// which will be copied into elements 0->9. The temporary storage is
// needed to avoid the copy operation interfering with itself.
A[0:10] = A[1:10] + 1;

// Matrix addition of the 2x2 matrices in A and B starting at A[3][3]
// and B[5][5].
C[0:2][0:2] = A[3:2][3:2] + B[5:2][5:2];

// Add the array slice along the 1st and 2nd dimensions of B to the
// elements in the array slice along the 2nd and 3rd dimensions of A,
// placing them in an array slice in C.
C[0:9][0][0:9] = A[0][0:9][0:9] + B[0:9][0:9][4];

// Element-wise addition of first 10 elements in A and B,
// resulting in A.
A[0:10] = A[0:10] + B[0:10];

// Element-wise negation of first 10 elements in A, resulting in A.
```

Figure 4. Examples of section operations.

4.3.2.3 Reduction Operation

The reduction operation accumulates all the values in the array using the given user function, or one of the specified operations. A list of reductions follows:

```
type fn(type in1, type in2);  // declaration of scalar reduction
                              // function
type in[N], out;              // array input and scalar output result

// accumulate successive elements in in with a user function fn,
// resulting in a single value out
out = __sec_reduce(fn, identity_value, in[x:y:z]);

out = __sec_reduce_add(in[x:y:z]); // out = sum of all values
out = __sec_reduce_mul(in[x:y:z]); // out = product of all values
out = __sec_reduce_all_zero(in[x:y:z]); // 1 if all values are zero
out = __sec_reduce_all_nonzero(in[x:y:z]); // 1 if all values nonzero
out = __sec_reduce_any_nonzero(in[x:y:z]); // 1 if any values nonzero
out = __sec_reduce_max(in[x:y:z]); // maximum value of values in in
out = __sec_reduce_min(in[x:y:z]); // minimum value of values in in
out = __sec_reduce_max_ind(in[x:y:z]); // index of maximum value
out = __sec_reduce_min_ind(in[x:y:z]); // index of minimum value
```

If a user function is provided, an identity value (starting value for the reduction) must also be provided. The reduction will be expanded as:

```
     result = identity_value;
     for (idx = x; idx < x+y; idx += z) {
       result = fn(result, in[idx]);
     }
```

For example, the two reduction operations given here are functionally equivalent:

```
     double add(double in1, double in2) { return in1+in2; }
     out = __sec_reduce(add, 0, in[x:y:z]); // accumulate using
add()
     out = __sec_reduce_add(in[x:y:z]); // accumulate using
built-in +
```

Figure 5. Complete list of reduction operations.

The compiler may produce more optimized code when the specialized reduction functions are used.

If the left-hand side of the assignment is of a different type than the array elements, an implicit cast will be performed, if valid. There is no specific order of elements with which the reduction function will be called. It is suggested, but not required, that the function be associative.

4.3.2.4 Scatter and Gather Operations

The gather operation takes an array section of indices, which must be of an integral type. For every value in an index array section *index[x:y:z]*, the corresponding element in an input array section *in[]* is collected and placed in the next element in the result array section *out[a:b:c]*.

The scatter operation takes each element in the input array section *in[a:b:c]*, and places it in the locations in the result array *out[]* which are given by the array of indices *index[x:y:z].*

```
unsigned int index[N];
type out[M],in[O];

// gather elements from in[], given by index[x:y:z], into out[]
out[a:b:c] = in[index[x:y:z]];

// scatter elements from in[] into various locations in out[],
// given by index[x:y:z]
out[index[x:y:z]] = in[a:b:c];
```

See Figure 6 below for an example.

The shapes of [a:b:c] and [x:y:z] must be the same. The values of the bounds themselves do not have to be the same (i.e. a != x and b != y).

Scatter/gather is implicitly performed when assignments with different stride values are performed, such as:

```
        B[0:5:2] = A[0:5] + C[0:5:3];
```

4.3.2.5 Array Implicit Index

In writing CEAN code it is sometimes useful to explicitly reference the indices of

Figure 6. Scatter/gather example.

the individual elements in a section. For example, the user may wish to fill an array with a function of the element index, rather than a single value.

Conceptually, an array section operation can be thought of as expanding into a loop with implicit index variables *i, j, k, etc.* for each rank of the section. The __sec_implicit_index(int i) function will return the value of the implicit index variable for the '*i*th' rank. The first rank is numbered 1, and therefore *i* must range between 1 and the number of array dimensions, inclusive.

```
int A[10], B[10][10];

// A[0] = 0, A[1] = 1, A[2] = 2,...
A[:] = __sec_implicit_index(1);

// B[i][j] = i+j;
B[:][:] = __sec_implicit_index(1) + __sec_implicit_index(2);
```

Figure 7. Implicit index example.

4.3.2.6 Map Operation

If a scalar ("standard") C/C++ function is called with array section arguments, the function will be "mapped", or called with successive elements in the array. In the example below, the given scalar function *fn* is mapped over the array sections *in* and *in2* given by the shape *[x:y:z]*. The function fn will be called with (in[x], in2[x]), (in[x+1], in2[x+1]), etc. The results of the function calls are collected into the array section *out[]*. All the array section arguments in a scalar function map call must have the same rank. Any scalar arguments are passed

```
type fn(type arg1, type2 arg2);  // declaration of scalar function
type in[N], out[N];
type2 in2[N];
out[x:y:z] = fn(in[x:y:z], in2[x:y:z]);
```

8. A sample function map operation.

to all the function calls. The element type of an array section argument must match the corresponding parameter type of the function declaration.

Mapped function calls are executed in parallel for all the elements with no specific ordering. The mapped functions may have side effects. When there are conflicts during parallel execution, such as setting an error code, the programmer is responsible for correctness.

4.3.3 Sections of Array Parameters

CEAN supports a "vector kernel" style of programming, where vector code is encapsulated within a function, with a parameterized vector length.

This differs from *maps* of array sections in that the function is not called in parallel; any parallel operations must be implemented within the body of the function itself.

The following example illustrates how to combine CEAN vectorization inside a function body with OpenMP* threading for parallel function calls. The vector length *m* is 256 in this example.

```
void saxpy_vec(int m, float a, float restrict (&x)[m], float (&y)[m])
{
   y[:] += a * x[:];
}

void main(void)
{
      int a[2048], b[2048] ;
#pragma omp parallel
      for (int i = 0; i < 2048; i += 256)
           saxpy_vec(256, 2.0, &(a[i]), &(b[i]));
```

Figure 9. Passing an array section to a function.

By writing the function explicitly with array arguments, the programmer can write code with easily customizable vector lengths and runtime model choices.

Please note that functions cannot return array section values. It may be helpful to return a pointer to the array as in standard C/C++ and take sections of the return value in the callee

5 Elemental Functions

5.1 Description

This section describes the “elemental functions” portion of the Cilk Plus language. Elemental functions are a data parallel programming construct. The use of elemental functions consists of the following three steps:

1. The programmer writes a function that uses scalar syntax in standard C/C++ to describe the operation to be performed on a single element.

2. The programmer adds a __declspec(vector) with optional clauses to be described below, so that the compiler projects the scalar operation onto a short vector implementation designed to operate on a vector of element instead of a single one.

3. Lastly, the programmer writes the invocation of the function to provide arrays of arguments instead of single arguments.

The function is invoked with vectors of arguments iteratively until the whole array of arguments is processed. Each such invocation is a single *instance* of the function.

5.2 Semantics

The semantics of the elemental functions are that the execution order among the instances is unsequenced.

The execution of the elemental functions depends on the language construct used at the invocation site, as follows.

1. If the function is called from a C/C++ for loop, then the compiler may replace the scalar function by the short vector function and iterate in the loop fewer times. While in this context the replacement is always correct, whether the replacement will actually be done is implementation dependent and is subject to performance heuristics.

2. Adding a #pragma sind to the C/C++ for loop will ensure that the short vector version of the function is called. Instances of the function will be invoked iteratively in a single execution strand until all elements of the array arguments have been processed.

3. If the function is invoked from a _Cilk_for loop, then the short vector version is used, and the execution of the _Cilk_for loop is as described in the _Cilk_for section. This may result in multiple instances of the function executing concurrently.

4. If the array notation syntax is used, then the execution is the same as in case #2 above. However, Intel is considering augmenting the implementation of this syntax and allowing concurrent execution of instances.

TIP: **Implementation note.** The compiler will generate three translations of the source code into machine code. One is for the compliant version, which operates on a single element per invocation. The second the short vector version, that operates on multiple elements at a time. The number of elements to be used in by the short vector version of the function is determined by the ration of the types of arguments and return value of the function, and the width of the vector registers in the target CPU. The programmer can use the vector length clause described below to override the compiler's choice. The third version is designed to receive an additional argument which is a mask argument. This argument is implicit and the application programmer does not explicitly provide it. The mask argument is required for the cases in which the function is called from a loop under a conditional statement. The application programmer can use the mask clause to suppress the generation of either the second or the 3'rd version.

5.3 declspec (vector) definition

The declspec has the following optional clauses. For each SIMD declspec, one instance of vectorized function is created. The declspec vector and its associated clauses are considered part of the function signature. If the function declaration is inconsistent with the function prototype with respect to these clauses then the behavior is undefined.

processor(cpuid)
directs the compiler to create a vector version of the function for the given target processor. The default processor is taken from the implicit or explicit processor- or architecture- specific flag in the compiler command line. The target processor defines both the vector ISA that the compiler is allowed to use, and the width of the vectors. The following token are accepted for the processor clause: pentium_4, pentium_4_sse3, core2_duo_ssse3, core_2_duo_sse_4_1, core_i7_sse4_2

- **vectorlength(num)**
  directs the compiler to use the vector length (VL) equal to num. The default vector length is computed from the processor's vector register and the size of the return value (or the first vector parameter if return type is void). For example:

    - If the target has XMM vector architecture (i.e., no YMM support) and the return type of the function is int, the default VL is 4.
    - If the target has AVX1.0 (Sandybridge) vector architecture,
        - the default VL is 4 if the return type of the function is int. (Integer vector operations in AVX1.0 are performed on XMM.)
        - the default VL is 8 if the return type of the function is float.
        - the default VL is 4 if the return type of the function is double
- **linear(param1:step1, param2:step2, …, paramN:stepN)** directs the compiler that the consecutive invocation of the function in the serial execution, the values of param1, param2, …, paramN are incremented by step1, step2, …, stepN, respectively.
- **scalar(param1, param2, …, paramN)** directs the compiler that the values of these parameters can be broadcasted to all iterations as a performance optimization.
- **mask / nomask** directs the compiler to generate only a masked / only an unmasked vector version of the function. If none is specified then both versions will be generated.

5.4 Rules

The following language construct shall not appear within a function that is marked with __declspec(vector):

- Loops, including the keywords for, while, do and overloaded operators
- Switch statements
- Goto, setjmp, longjmp
- Calls to functions other than other elemental functions and the intrinsic short vector math libraries provided with the Intel compilers
- Operations on structs, other than member selection
- _Cilk_spawn
- Array notations
- C++ exceptions
- Elemental functions cannot be virtual, and can only be called directly, not through a function pointer

Note that Intel is considering removal of some of these restrictions in the future.

6 Pragma SIMD

6.1 Description

This section describes the pragma simd portion of the Cilk Plus language. The pragma can be applied to a loop, to indicate the intention that the loop needs to be implemented using vector instructions. Unlike the other components of the Intel® Cilk™ Plus language extension, this extension applies to Fortran as well as to C/C++. It is expressed as a pragma for C/C++ and as a dir for Fortran. Otherwise it works the same way for C/C++ and Fortran.

6.2 SIMD pragma/directive definition

SIMD pragma has the following five optional clauses to guide the compiler. It is the programmer's responsibility to correctly use these clauses so that the compiler can obtain enough information to generate correct vector code.

- **vectorlength(num1, num2, …, numN)** directs the compiler that it must choose from one of the specified vector lengths (VL). Each iteration in the vector loop will execute the computation equivalent to the VL iterations of scalar loop execution. Multiple vector length clauses are merged as a union. The values of num1 through numN have to be compile- time constants.
- **private(var1, var2, …, varN)** directs the compiler to make these variables private to each iteration of the loop. Multiple private clauses are merged as a union. Unless the compiler can prove initial/last values are unused, initial value is broadcast to all private instances, and the last value is copied out from the last iteration instance. The names var1 throguh varN have to be names of variables that are available in the scope of the loop.
- **linear(var1:step1, var2:step2, …, varN:stepN)** directs the compiler that, for every iteration of scalar loop, var1 is incremented by step1, var2 is incremented by step2, and so on. Every iteration of the vector loop, therefore increments these vars by VL*step1, VL*step2, …, VL*stepN, respectively. Multiple linear clauses are merged as a union. If a var is given two or more steps, the program is ill formed. The names var1 through varN have to be variable names available in the scope of the loop. The expressions step1 through stepN can be either compile- time constants or expressions that evaluate in run time.
- **reduction(operator:var1, var2,…, varN)** directs the compiler to perform vector reduction of operator kind to var1, var2, …, varN. A SIMD pragma may have multiple reduction clauses with the same or different

operators. If the same variable var appears in more than one reduction clauses the program is ill formed.

- **[no]assert** directs the compiler to assert or not to assert if it cannot generate vector code. The default is assert. If noassert is specified, then if the compiler fails to generate vector code then it will be allowed to generate scalar code instead. If multiple instances of this clause appear on the same pragma the program is ill formed.

6.3 Additional rules

- A variable may belong to at most one of private, linear, or reduction (or none of them).
- Within the vector loop, an expression is evaluated as a vector value if it is private, linear, reduction, or it has a subexpression that is evaluated to a vector value. Otherwise, it is evaluated as a scalar value (= broadcast the same value to all iters). Scalar value does not necessarily mean loop invariant, although that is the most frequently seen usage pattern of scalar value.
- A vector value may not be assigned to a scalar L-value.

6.4 Restrictions

- The countable loop for the SIMD pragma has to conform to the for-loop (C/C++) or DO-loop (FORTRAN) style usable for OpenMP worksharing loop construct. See the OpenMP 3.0 specification http://www.openmp.org/mp-documents/spec30.pdf (Section 2.5.1). Additionally, the loop control variable has to be signed or unsigned integer type, or a pointer type.
- SIMD loop body must be free from C++ exception handling code and Windows SEH, setjmp and longjmp code.
- Fast FP compilation model is used for the SIMD loop.
- The vector values must be of “plain old data” type: signed/unsigned 8/16/32/64-bit integers, single/double-precision floating point or single/double-precision complex. Later implementations will extend this to other types.
- The SIMD loop may not contain another loop (i.e., innermost loops only). Note that C++ inlining can create such an inner loop and result in an error, which may not be obvious at the source level. This restriction is intended to be removed in the future.
- The SIMD loop performs memory references unconditionally. Therefore, all address computations have to result in valid memory addresses, even though such locations may not be accessed if the loop is executed sequentially.
- A linear variable p has to have an assignment of the form p+=step within the loop body. If there is no assignment, or there are multiple assignments to same variable which was declared as linear using the liner clause, then the program is ill formed. If the value of step as used

within the loop is different from the value provided in the linear clause then the program behavior is undefined.