Doc No: N3419=12-0109
Date: 2012-09-21
Author: Robert Geva
Intel Corp. robert.geva@intel.com

Vector loops and Parallel Loops

Contents

1 Introduction

This document presents proposals for two language constructs: vector loops and parallel loops. It provides their motivations and semantics. Although this paper offers many specifics describing the capabilities and semantics of the proposed constructs, it does not yet attempt to present formal wording for WP changes. Those details will be forthcoming if and when the committee agrees with the direction of this proposal. The syntax used in the examples is intended as a straw-man proposal; actual keyword names, attributes, and/or operators can be determined later, when discussion has progressed to the point that a bicycle-shed discussion is in order.

1.1 Motivation

The need for adding language support for parallel programming to C++ was presented in February 2012, in Kona. The presentation described the growth of multicore and vector hardware and the need to support programming this new hardware cleanly, portably, and efficiently in C++. The Evolution Working Group (EWG) in Kona agreed that parallelism is an important thing to support and created a study group to research it further. The study group met in Bellevue, WA in May, 2012. There appeared to be enthusiasm for targeting some level of both multicore and vector parallelism support for the next standard (also known as *C++1y*, tentatively targeted for 2017). The two proposals in this document provide more specifics to what was presented in Kona and Redmond, in addition to the proposal in N3409, which also provides additional motivation. Also, specific background and motivation for parallel loops and vector loops are presented below, as part of their respective sections.

1.2 Document structure

Both parallel loops and vector loops are countable loops. The document therefore provides a language specification for countable loops, which is a part of both proposals. (There is no proposal to introduce countable loops per se as a language construct). Then, the proposal describes parallel loops and vector loops separately, and concludes with some alternatives and a discussion on the semantic differences between parallel and vector loops.

2 Countable loops

A countable loop is a loop whose trip count can be determined before its execution begins. The advantage of that knowledge is efficient implementation, especially for parallel loops.

Every countable loop has a single loop control variable (LCV). The LCV is initialized before the execution of the loop. It is used to determine the termination of the loop, by comparing its value to another expression, and it is incremented as part of the increment clause of the loop. The amount of increment, or *stride,* is loop invariant. The LCV is the only variable that is used both in the condition clause and incremented in the increment clause of the loop.

2.1 Grammar

Iteration-statement:

> `modified_for` `(` *for-init-decl*$_{opt}$ `;` *condition* `;` *incr-expression-list*`)` *statement*

Here, "`modified_for`" is a placeholder for an actual keyword to be used in a proposal that relies on countable loops. Below, this document will use `cilk_for` for the proposed parallel loops and `simd_for` for the proposed vector loops.

2.2 Syntactic constraints

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

The initialization portion of the countable loop has the same rules as a `for` loop in the current C++ language specification.

The condition and the *incr-expression-list* shall not be empty. The condition shall have one of the following two forms:

> *identifier OP expression*
> *expression OP identifier*

where *OP* is one of: == != < > <= >=.

The loop increment can have a comma separated list of expressions, where exactly one of them involves the same *identifier* that appears in the condition section. That *identifier* is called the loop control variable (LCV). Any other variables modified by these expressions are additional induction variables. Each expression within the *incr-expression-list* shall have one of the following forms:

> `++` *identifier*
> *identifier* `++`
> `--` *identifier*

identifier --
identifier += *incr*
identifier -= *incr*
identifier = *identifier* + *incr*
identifier = *incr* + *identifier*
identifier = *identifier* - *incr*

For ++ operators, the *stride* is defined to have the value of 1; for -- operators, the *stride* is defined to have the value of -1; for the += operator, the *stride* is *incr*; and for -= the stride is *–incr*.

Each induction variable, including the LCV, shall have integral, pointer or class type. No storage class may be specified within the declaration of an induction variable. It may not be declared as `const` or `volatile`. Because modification of an induction variable in a parallel or vector loop causes undefined behavior (see dynamic constraints, below), each induction variable is treated as if it were `const` within the loop body, including for the purposes of overload resolution.

Condition syntax	Requirements	Loop count
identifier < *limit* *limit* > *identifier*	(*limit*) − (*first*) shall be well-formed and shall yield an integral *difference_type*; *stride* shall be > 0	((*limit*) − (*first*)) / *stride*
identifier > *limit* *limit* < *identifier*	(*first*) − (*limit*) shall be well-formed and shall yield an integral *difference_type*; *stride* shall be < 0	((*first*) − (*limit*)) / −*stride*
identifier <= *limit* *limit* >= *identifier*	(*limit*) − (*first*) shall be well-formed and shall yield an integral *difference_type*; *stride* shall be > 0	((*limit*) − (*first*) + 1) / *stride*
identifier >= *limit* *limit* <= *identifier*	(*first*) − (*limit*) shall be well-formed and shall yield an integral *difference_type*; *stride* shall be < 0	((*first*) − (*limit*) + 1) / −*stride*

identifier != *limit*
limit != *identifier*
	(*limit*) − (*first*) and (*first*) − (*limit*) shall be well-formed and yield the same integral *difference_type*;
	stride shall be != 0
		if *stride* is positive then ((*limit*) − (*first*)) / *stride*
		else ((*first*) − (*limit*)) / −*stride*

The *incr* expression shall have integral or enumeration type. If the loop increment uses operator ++ or +=, the expression:

identifier += (*difference_type*)(*incr*) shall be well-formed. If the loop increment uses operator `--` or `-=`, the expression

$$identifier\ {-=}\ (\ difference\_type\ )\,(incr)$$

shall be well-formed. The loop is a use of the required operator function.

2.2.1 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 any induction 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 the *LCV* 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.

[*Note:* Unsigned wraparound is not allowed – *end note*].

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.

In either parallel or vector execution of a countable loop, the copy constructor for an induction 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.

3 Parallel Loops

3.1 Background

This portion presents the proposal to add a language construct to the C++ language to express parallel loops. Obviously, parallel loops are loops where the iterations are unordered and can execute in parallel with one another. The specifics of this proposal, cilk_for parallel loops, include additional capabilities:

- cilk_for parallel loops are composable with the tasking model elaborated in a separate document (N3409);
- cilk_for parallel loops support hyperobjects, which are briefly introduced in N3409 and will be more fully described in a future proposal, as a vehicle for reductions; and
- the *serialization* of cilk_for parallel loops, which is obtained by replacing the `cilk_for` with serial `for`, is syntactically well defined and semantically equivalent to serial execution.

A parallel loop in which the amount of parallelism is not mandated by the language construct enables the runtime scheduler to realize significant efficiencies, including automatic load balancing among iterations of the loop. Please refer to the background provided in Intel's related proposal, N3409; it introduces some of the concepts and terminology used within this paper. The strict fork-join parallelism documentis a pre-requisite to this proposal. That document provides overall motivation for adding language constructs for structured fork-join parallelism. In the current document we describe the parallel loop construct specifically in relation to strict fork-join parallelism.

3.2 A language based construct

The language-based `cilk_for` allows for a more efficient implementation than, for example, the following `cilk_spawn`-based alternative. This alternative uses a sequence of `cilk_spawn` invocations to parallelize the iterations of a serial `for` loop:

```
for (k = 0; k < K; ++k) { cilk_spawn body(k); }
```

While this loop and many other possible parallel loop constructs share the obvious semantics that the iterations can execute in in parallel, there are a few specific and important additional properties held by the proposed `cilk_for` construct:

1. The `cilk_for` construct is synchronous with respect to the statement following the `cilk_for` loop, whereas in a program with a serial for loop with a cilk_spawn in each iteration, as above, the spawns continue in parallel execution until they encounter a `cilk_sync`.
2. In the `cilk_for` construct, the loop is countable, i.e. the trip count is known at run time before the execution starts. The compiler can enforce that the loop is countable via static analysis.

3. The cilk_for construct allows for composable implementation by not providing semantics that enforce parallel execution and / or tying execution to any resources.
4. The countable loop allows an efficient implementation in which the compiler and the scheduler use a divide and conquer algorithm where the loop iterations are divided into varying batches of trip count, allowing maximal parallelism with minimal movement of work between cores, while still providing composability across the whole program.
5. The loop interoperates with the same hyperobjects as the structured fork join parallelism proposal. The same serial equivalence guarantees are provided.

The syntactic advantage of `cilk_for` being a language construct is the similarity to serial for loops. This, combined with the fact that most candidates for parallelization in existing serial programs are `for` loops, makes it a productive way to parallelize existing programs. Use of a syntactically similar construct also makes it possible to toggle parallelism on / off for development tasks such as debugging and performance engineering.

3.3 Grammar

iteration-statement:

`cilk_for (` *for-init-decl*$_{\texttt{opt}}$ `;` *condition* `;` *incr-expression-list*`)` *statement*

The serialization of “cilk_for” is “for”

A cilk_for loop shall be a countable loop.

The loop control variable shall be declared either in the loop statement or in the innermost task block (see N3409) containing the loop.

The syntax above describes the construct available in the current Intel compiler and the Cilk Plus branch of the gcc compiler. A future construct may be proposed for parallel loops that look more like the current range-based `for` loops introduced in C++11.

3.3.1 Dynamic constraints

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. However, once any loop iteration begins, it is executed either until it terminates normally or throws an exception, even if another iteration has thrown an exception. If multiple loop iterations throw exceptions that are not caught in the loop body, the `cilk_for` statement re-throws the exception that would have occurred first in the serialization of the program once all pending iterations have completed.

It should be noted that if an exception occurs, some code may be executed that would not have been executed in the serialization of the parallel loop. However, all objects constructed within the loop will be destructed either by the normal termination of that iteration or by the handling of an exception.

3.3.2 Semantics

The iterations of a cilk_for loop can execute in any order.
The body of a cilk_for loop is a task block (See N3409).

4 Vector loops

4.1 Introduction

This portion of the document presents a proposal to add a language construct to the C++ language to express vector execution of loops. The fundamental capabilities of this construct are:

- the designation of a loop as a vector loop,
- the designation of a portion of the loop as an ordered block that is sequenced as if it appeared in a sequential loop,
- the ability to call elemental (vectorized) functions from the loop, which execute as if they were part of the loop body,
- the ability to use multiple linear induction variables and
- the ability to perform efficient vectorized reductions.

While the language proposal is new, execution of vector loops, e.g. via compiler auto vectorization, has been in practice for some time. Therefore, the goal of this proposal is not to invent new semantics but rather to capture the expectations of programmers who have been using vector execution without the benefit of being able to express them explicitly via language constructs.

4.2 Problem statement

Processors have been providing single-instruction-multiple-data (SIMD) level parallelism at least since the early 1990's. The hardware trend has been to grow such SIMD capabilities steadily, both by providing wider vector registers and by improving the capabilities of the instruction sets that operate on the vectors. While the core counts vary at a given point in time between various product configurations, such as processors in servers vs. phone form factors, the vectors are typically part of the instruction set architecture (ISA) and are fixed across all form factors. Therefore, the performance improvement from SIMD level parallelism is more predictable than the performance improvement gained with core / thread level parallelism. In smaller form factors, the performance potential from vector level parallelism is larger than the performance potential from core and thread level parallelism.

Vector / SIMD execution has been in practice for many years, in fact predating the availability of SIMD in microprocessors, and is well understood. SIMD execution on microprocessors is currently achieved mostly by one of two ways: writing scalar loops and relying on compiler optimizations (auto vectorization) to convert the code within such scalar loops into vector code, or by writing intrinsics, which are very low level programming constructs that map onto a particular SIMD ISA.

This proposal intends to allow a developer to directly express the intent that operations can execute in an order consistent with a single-instruction-multiple-data order of execution, allowing the compiler to use vector level parallelism. The expectation is that the language does not have to add new data types, and therefore expressions can be written using operations that operate on existing, scalar data items. The problem of vectorizing a scalar loop then becomes a problem of order of evaluation, and this proposal is focused on loops with semantics that allow an order of evaluation which facilitates vector execution, while maintaining consistency with existing serial loop constructs.

4.3 Vector execution

The iterations of a vector loop execute on a single thread, and in chunks, where informally, the size of the chunk should ideally correspond to as many iterations of a scalar loop as can fit within the vector resources of the target machine. Vector execution requires reordering expressions from different iterations so that multiple evaluation of the same expression from different iterations can be grouped together. The semantics of vector loops are expressed in terms of these allowed reorderings, rather than in terms of vector instructions.

Vector loops are unlike parallel loops. The semantics of parallel loops are that the iterations are un-sequenced. The sequencing rules of vector loops in contrast allow interleaving of operations from multiple iterations but provide more strict guarantees on ordering. Another interesting distinction is chunked execution. Loop iterations that execute together cannot make progress independent of each other, i.e. a subset of the vector lanes cannot block while others make progress. Therefore, using existing constructs for critical sections will not work – they will likely deadlock (in the case of locks and mutexes) or livelock (in the case of spin locks). Seen from the other direction, since parallel loops are expected to have well-defined behavior when critical sections are used within them, the implication is that a parallel loop cannot also be used as a vector loop. Other programming constructs have well-defined behavior in parallel loops but not in vector loops.

4.4 Syntax:

This section does not yet constitute formal wording. Formal wording for inclusion in a future working paper will be submitted as a proposal to the CWG if and when the EWG agrees to the concept and general principles of vector loops.

iteration-statement:

    `simd_for` *chunk-clause*$_{opt}$ `(` *for-init-decl*; *condition* ; *expression* `)` *statement*

chunk-clause:

    `< chunk =` *constant-expression* `>`

inorder-statement:

    `inorder` *statement*

The *scalar elision* of a vector loop is a C++11 loop obtained from the `simd_for` loop by replacing the keyword `simd_for` by the keyword `for`, and deleting the optional *chunk-clause*. The scalar elision of *inorder-statement* is *statement*.

The scalar elision of a vector loop is defined syntactically and is a well formed loop in C++11, and produces a result equivalent to the vector loop.

4.5 Language Rules

A vector loop shall be a countable loop.

The loop control variable shall be declared in the same function that contains the loop and:

- If the vector loop is nested either within a vector loop or within a parallel loop, then the LCV shall be declared within the enclosing vector loop.
- If the vector loop is nested within a task block then the LCV shall be declared within the task block.

The expression `<chunk=`*N*`>` is optional. *N* shall be a positive integral compile-time constant.

The following constructs shall not appear within the body of a vector loop:

1. Any parallelism construct, such as creation of a thread, the locking of a mutex, or a parallel loop;
2. Throwing or catching an exception.

4.6 Semantics

A vector loop executes in chunks, where the chunk size is determined by the implementation. If the optional chunk expression is present, then the actual chunk size used by the implementation can be the same or smaller but not greater than the specified size. However, note that reducing the chunk size does not change the dependencies allowed by a larger chunk size, according to the following definitions.

4.6.1 Notation

For an expression X, X_i is the evaluation of the expression X in the i^{th} iteration of the loop.

4.6.2 Evaluation order to allow reordering

0. If expression X_i is sequenced before Y_i in the scalar elision of the loop, then X_i is also sequenced before Y_i in the vector loop.
1. For every X_i and X_{i+c} evaluated as part of a vector loop with chunk size *c*, X_i is sequenced before X_{i+c}
2. For any *X* and *Y* evaluated as part of a vector loop, if X_i is sequenced before Y_i and $i < j$, then X_i is sequenced before Y_j.

4.6.3 Restrictions on Variables:

1. Variables declared in the loop are private per iteration of the loop. The implication is that each chunk of the vector loops sees a vector of size 'chunk' of these variables.
2. Variables declared outside of the loop are uniform; they are shared across all iterations of the loop. Assignment to these variables in more than one unsequenced expression will produce undefined behavior.

4.7 Ordered blocks

The keyword `inorder` applies to a statement block. The expressions in the ordered block are evaluated in a more strict order, unlike those in the rest of the vector loop: for any two sub-expressions X and Y within an ordered block of a loop, X_i is sequenced before Y_{i+1}.

This allows certain constructs to be used in the body of a vector loop that would not otherwise be legal. In particular, it allows the use of scoped locks.

4.8 Elemental functions

Elemental functions add modularity to vector loops, and allow separate compilation of functions to be called from vector loops. When an elemental function is called from a vector loop, multiple consecutive instances of the elemental functions execute in a chunk, as if they were compiled as a part of the body of the vector loop. The ability to write vector code outside of the scope of the vector loops allows modular programming and independent deployment, such as in libraries.

The details of the elemental functions construct are not presented at this time for brevity. A more detailed description is expected towards the next meeting.

5 Appendix 1: Alternative to guarantee / enforce reordering:

The proposed semantics allow the compiler to reorder expressions in an order that facilitates vectorization, but do not require it. They also allow implementations to use the same order as executing the scalar elision of the loop. The following alternative describes semantics that would *require* an execution order that is achievable with vector execution but is inconsistent with serial execution, so it would not be possible to support scalar elision.

In this alternative, for any expressions X and Y evaluated as part of a vector loop, if X_i is sequenced before Y_i and iterations i and j are evaluated in the same chunk, then X_i is sequenced Y_j, regardless of whether i < j.

Example:

Consider the following code illustration:

```
void foo( int *a, int n )
{
 int itmp[4] = {3,2,1,0};
    for (int i = 0; i < n; i += 4) {
        simd_for <chunk=4> (int j = 0; j < 4; j++) {
            int t = a[ i + itmp[j]];
            a[i + j] = t;
       }
    }
 }
```

Without the alternative rule, the vector loop in this code illustration has unsequenced value computations and side effects of non-atomic objects, and thus its behavior is undefined. With the alternative rule, the behavior is well-defined and should result in a reversal of the values in the array a. However, the alternative rule does not give the implementation latitude to choose an optimal chunk size that matches the hardware capabilities, possibly resulting in performance degradation. This alternative is not being proposed at this time.

6 Alternatives

The focus of this document is on the capabilities and on the semantics of the parallel and vector loop constructs. Syntaxes for these constructs are presented to make the proposal concrete but they are not an inherent part of the proposal. The language constructs proposed here can be as powerful and as useful with alternative syntaxes.

One example of an alternative syntax would be to replace the proposed reserved words, `cilk_for` and `simd_for`, by contextual keywords that would appear between the existing keyword `for` and the open parenthesis, such as

`for simd (` *init* ; *compare* ; *expression* `)`

Another potential alternative is to use the attribute syntax, for example

`[[simd]] for (` *init* ; *compare* ; *expression* `)`

7 Distinctions between the two loop constructs

This document proposes two constructs, one for parallel loops and one for vector loops. While both would be new language constructs in C++, they are not new for practitioners. Programmers have significant amount of experience with parallel loops and with vector loops, accomplished with alternative means such as OpenMP and automatic vectorization. The goal of describing the semantics therefore is not an invention of a new execution of a loop, but rather, an attempt to capture existing practices.

7.1 Commonality

Parallel loops and vector loops have a few common characteristics. They both require the loop to be a countable loop, as defined in this document. They both relax the ordering constraints that would be required if the loop was a serial loop.

7.2 Recap: the difference in the specifications

The root of the difference between parallel loops and vector loops is that they relax ordering constraints differently from each other. The parallel loop can execute all iterations in any order. The order of execution of a vector loop is more constrained, as specified here. There are two sets of implications. One is semantic, and one is performance.

7.3 Semantic differences

The semantic specification provided here for parallel loops is consistent with existing practices and expectations of programmers, in particular, that these loops allow use of critical sections. A potential implementation of a critical section is to lock an object, enter the critical section, evaluate it and release the lock.

The semantics specification provided here for vector loops allows the compiler to implement the loop using vector instructions, with the implication that iterations of the loop that are executing in a concurrent but lockstep fashion, and cannot make forward progress independent of each other. The result is that while critical sections have well defined and expected behavior in a parallel loop, they would cause a deadlock in a vector loop.

Conversely, the specification provided here for vector loops captures existing practices and expectations of forward data dependence across the iterations of a vector loop. Namely, a value created in an iteration *j* of the loop can be used in any iteration *k* where $k > j$.

Parallelizing a vector loop will break code that relies on these dependences and will produce different results.

7.4 Difference in performance model

Performance requirements to keep a parallel loop scalar and avoid vectorizing it are unlikely. Existing practices do welcome automatic compiler vectorization of parallel loops on a best effort basis. The converse is often not the case.

Consider for example divide and conquer algorithms, a well-known design pattern. A divide and conquer algorithm can be used to break a large problem size to smaller problem sizes and operate on the small problems concurrently. The divide can be applied recursively, and the resulting tasks can execute in parallel, by parallelizing the recursion. Once the problem size is small, the algorithm executes a base case. By design, the considerations for parallel execution were expressed by parallelizing the recursion, and therefore programmer's expectation is that the base case is not parallel. On the other hand, whenever the base case is implemented as a loop that can be vectorized, using a vector loop would be appropriate and productive, while using a parallel loop would be counter-productive.

8 Summary

All current platforms provide hardware resources for parallel execution, and many of them have multiple levels of parallelism, including cores and vectors. The two proposals in this document, alongside additional proposals and in particular N3409, intend to add parallelism to the C++ language and allow C++ to be used for parallel programming and not fall behind other languages.

The proposals are based on well understood programming practices done both in other languages and within C++ via auxiliary constructs such as OpenMP. The integration into the C++ language is expected to provide a safer solution for the programmer as well as make C++ a leading choice for parallel programming.