Graduate from MIT to GCC Mainline with Intel® Cilk™ Plus

By Barry Tannenbaum, *Software Development Engineer, Intel*

Overview

Intel® Cilk™ Plus is a set of extensions to C and C++ used to express task and data parallelism. These extensions are easy to apply, yet powerful, and used in a wide variety of applications. Like the very popular Intel® Threading Building Blocks (Intel® TBB), Intel Cilk Plus was inspired by the MIT Cilk project. Unlike Intel TBB, Intel Cilk Plus is compiler-based—allowing it to expand beyond parallel tasking with features for vectorization and reductions—and can be used in C as well as C++. Support for Intel Cilk Plus is now available in Intel® compilers, the GNU Compiler Collection (GCC), and a branch of LLVM/Clang.

For decades, software developers have depended on Moore's Law to make applications run faster and do more. In 1965, Intel co-founder Gordon Moore noted that the number of transistors on integrated circuits doubled every two years. This trend has allowed computers to become faster and more powerful, from the original Intel® 4004 to the Intel® Xeon® processors that power today's datacenters. But in the mid-2000s, conventional, general-purpose CPUs hit a wall:

increasing clock rates offered more problems (e.g., power consumption) than benefits. The solution adopted by Intel and other CPU designers was to use the additional transistors predicted by Moore's Law to create multiple cores and to add vector instruction units that can perform the same operation on multiple values simultaneously. The number of cores on a CPU and vector unit width has steadily increased. Today, Intel produces the Intel® Xeon Phi™ product family of coprocessors with up to 61 cores, each having vector units capable of operating on as many as sixteen single-precision floating point or eight double-precision floating point computations at a time.

Unfortunately, programming a multithreaded, vectorized application that can take advantage of the resources available in modern CPUs is complex and error-prone. Even the most experienced developers can't keep track of all of the things that may happen simultaneously. The result is buggy programs with problems that are difficult to reproduce and fix, and that don't scale well when the core count is increased. Intel Cilk Plus is designed to make it easier for developers to build, develop, and debug robust applications that take full advantage of modern processor architectures.

A Small Taste of Intel Cilk Plus

The classic example of an Intel Cilk Plus application is the following calculation of a Fibonacci number. There are certainly much better ways to calculate Fibonacci numbers, but this implementation provides a good example of a recursive function.

```
int fib(int n)
{
    if (n < 2)
        return n;
    int x = cilk_spawn fib(n-1);
    int y = fib(n-2);
    cilk_sync;
    return x + y;
}
```

This example demonstrates task parallelism using Intel Cilk Plus. The first recursive call to fib() can execute in parallel with the continuation (the second recursive call to fib()). Both recursive calls must complete before the results are combined and returned at the end of the function. Intel Cilk Plus is especially well suited to recursive algorithms, though an easy-to-use looping construct is also available.

Matrix multiplication provides a small example of data parallelism in Cilk Plus. Here's a simple serial implementation:

```
template<typename T>
void matmul(int ii, int jj, int kk, T *a, T *b, T *product)
{
    for (int i = 0; i < ii; ++i)
        for (int j = 0; j < jj; ++j)
            for (int k = 0; k < kk; ++k)
                product[i * jj + j] += a[i * kk + k] * b[k * jj + j];
}
```

And, here's the same code using array notations to vectorize the inner loop:

```
template<typename T>
void matmul_vec(int ii, int jj, int kk, T a[], T b[], T product[])
{
    for (int i = 0; i < ii; ++i)
        for (int j = 0; j < jj; ++j)
            product[i * jj + j] =
                __sec_reduce_add(a[i*kk:kk:1] * b[j:kk:jj]);
}
```

The statement

```
a[i*kk:kk:1] * b[j:kk:jj]
```

multiplies each element of a row of matrix a against a column of matrix b, producing a temporary vector of length `kk`. The call to `__sec_reduce_add()` sums the elements of the temporary vector.

The Origins of Intel Cilk Plus

The Intel Cilk Plus programming language grew out of three separate research projects at the MIT Laboratory for Computer Science. When these projects were combined in 1994, the resulting project was christened "Cilk," a play on threads and the C language. MIT Cilk is an extension of C and implemented as a source-to-source compiler. The Cilk-1 system was first released in September 1994. The current implementation, Cilk-5.3, is available from the MIT Computer Science and Artificial Intelligence Laboratory (CSAIL), though it is no longer supported.

In 2006, Cilk Arts, Inc. licensed the Cilk technology from MIT with the goal of developing a commercial implementation extending C++. Cilk++ v1.0 was released in December 2008 with support for Windows Visual Studio* and GCC/C++ on Linux*. While interesting, Cilk++ used non-standard linkages, making it difficult to call Cilk++ code from C or C++, or to use standard debuggers with Cilk++ code.

On July 31, 2009, Cilk Arts announced that its products and engineering team had been acquired by Intel Corporation. Intel and Cilk Arts integrated and advanced the technology further, resulting in the release of Intel Cilk Plus as part of Intel® C++ Composer 12.0 in September, 2010. Intel Cilk Plus provides all the power of previous Cilk implementations for both C and C++. It uses standard calling conventions, is compatible with existing debuggers, and adds syntax for support of data parallelism.

Intel has stated its desire to refine Intel Cilk Plus and gain industry-wide adoption. It published the Cilk Plus Language Specification, ABI Specification, and Zero-Cost Annotation Specification in November 2010 to allow other vendors to implement Intel Cilk Plus and, optionally, to adopt the Intel Cilk Plus runtime library.

In 2011, Intel announced that it was helping implement Intel Cilk Plus in the "cilkplus" branch of the GCC C and C++ compilers. The initial implementation was completed in 2012, and presented at the 2012 GNU Tools Cauldron conference. As part of the branch, Intel released the Intel Cilk Plus runtime as open source. Since then, Intel has worked with the GCC community to merge Intel Cilk Plus into the GCC mainline, culminating in the acceptance of Intel Cilk Plus into the GCC trunk for inclusion in the GCC 4.9 release.

What does the availability of Intel Cilk Plus in the GCC C and C++ compilers mean for developers?

- It means that Intel Cilk Plus is now available from a second source. Developers can code to Intel Cilk Plus confident that their code isn't dependent on features available only in Intel® Composer XE. Regardless of what Intel does to its compiler, Intel Cilk Plus will still be available in GCC.
- The implementation of Intel Cilk Plus, including the full source of the Intel Cilk Plus runtime, is available for users to inspect, comment upon, and improve. Intel has provided a community website for Intel Cilk Plus and has stated its willingness to accept contributions from the open source community to port Intel Cilk Plus to other architectures, other operating systems, and extend and improve the language.

Why Use Intel Cilk Plus?

Intel Cilk Plus provides a higher level of abstraction than other threading frameworks such as Intel TBB or OpenMP*. Instead of focusing on what needs to be done to execute an application efficiently in parallel, Intel Cilk Plus encourages programmers to focus on expressing the potential parallelism in an application.

The Intel Cilk Plus runtime uses a *work stealing scheduler* to efficiently run applications on all of the available cores. If a core stalls for some reason, or if the workload is unbalanced, the scheduler will steal tasks from that core and run them on idle cores. The separation of the expression of parallelism from the scheduling of the parallel execution means that when sufficient parallelism has been expressed, Intel Cilk Plus applications can scale to more cores without modification.

Intel Cilk Plus allows developers to write *composable* code. Composability means that you can write a library using Intel Cilk Plus and call it from applications using Intel Cilk Plus, without worrying about oversubscribing your system.

Intel Cilk Plus reducers allow developers to resolve race conditions without using locks.

The Intel Cilk Plus runtime and Intel Cilk Plus reducers work together to provide *serial semantics*. A properly written deterministic Intel Cilk Plus application will give the same results regardless of the number of cores it runs on. Serial semantics makes it easier to test your application, since you can directly compare a serial result with a parallel result. It can also make it easier to debug your application, since you can run it on a single core and use standard debuggers to examine the program state.

The vectorization extensions in Intel Cilk Plus allow programmers to express the data parallelism in their program. Failure to fully utilize the vector units in modern CPUs leaves a majority of the computational capabilities idle.

"The Intel® Cilk™ Plus work-stealing scheduler will automatically schedule the tasks on as many cores as possible and balance the load among the processors in a near-optimal fashion."

Intel® Cilk™ Plus Features

- Tasking keywords: Simple, yet powerful, expressions of the task parallelism of your application.
- Reducers: Eliminate contention for shared variables among tasks by automatically creating "views" of them as needed and "reducing" them in a lock-free manner.
- #pragma simd: Specify that a loop should be vectorized.
- Array notation: Allow users to express data parallelism for arrays and sections of arrays.
- SIMD-enabled functions: Generate vectorized variants of functions that can be implicitly called from an array notation expression or a #pragma simd loop.

Tasking Keywords

Intel Cilk Plus adds three keywords: `cilk_spawn, cilk_sync` and `cilk_for`. These describe the parallel regions of your application. The Intel Cilk Plus work-stealing scheduler will automatically schedule the tasks on as many cores as possible and balance the load among the processors in a near-optimal fashion. .

cilk_for

A `cilk_for` loop is like a standard for loop, with the proviso that any iteration of the loop can run in parallel with any other iteration. All iterations of the loop are guaranteed to have completed before the execution continues after the loop. The Intel Cilk Plus runtime will automatically break the loop into portions to make optimal use of the available cores on the system.[1]

cilk_spawn and cilk_sync

`cilk_spawn` specifies that a function can be executed in parallel with the continuation of the calling function.

`cilk_sync` specifies that all function calls spawned in a function must complete before execution can continue.

`cilk_spawn` and `cilk_sync` make it easy to implement recursive algorithms in Intel Cilk Plus. The developer uses these two keywords to annotate a program, and the Intel Cilk Plus runtime will select which processors should run each section.

It is important to note that the Intel Cilk Plus keywords describe the parallelism of the program; they do not command it. The Intel Cilk Plus runtime will schedule the parallel regions described by `cilk_for`, `cilk_spawn`, and `cilk_sync` based on the available CPU resources. A properly written Intel Cilk Plus program will run correctly on a single core.

Unlike some other threading packages, Intel Cilk Plus programmers should not try to tailor their programs to the number of cores the program is running on. Tuning your application to run optimally on a specific number of cores is fragile. All of the tuning needs to be redone when you or your customer gets a new system with more cores or a different amount of cache memory, or another program is run on the system simultaneously with your program. As a general rule, Intel Cilk Plus programs should expose at least ten times as many parallel tasks as there are CPUs in order to allow the Intel Cilk Plus runtime to adapt to the conditions on the system, stealing work from busy cores and executing it on cores that have available cycles.

The Intel Cilk Plus keywords implement *fully-strict fork/join* parallelism. That means that any spawns that occur within a cilk_for loop or within a function will complete before the cilk_for or function exits. This strictness makes it easier to reason about the parallelism in your program, since it limits the code that needs to be considered.

Reducers

In addition to all the errors that serial programs are prone to, adding parallelism introduces *race conditions*. A race condition occurs when two parallel regions of code access the same memory location and at least one of them writes to the memory location. The result is undefined behavior. The traditional solution to fix race conditions is to use locks to protect shared variables, but locks introduce their own problems:

- Incorrect lock usage can result in deadlocks.
- Contention for locked regions of code can slow down a parallel program.
- Even when properly used, locks do nothing to enforce ordering, possibly resulting in non-deterministic results.

Reducers solve all of these problems by providing each parallel region with its own view of a variable which it can update without locking. When the parallel regions sync, the views are merged in the same order as they would have been if the program were run serially.

#pragma simd

#pragma simd tells the compiler that the following for loop is intended to be vectorized. If the compiler is not able to vectorize the loop for some reason, the compiler can be told to issue a warning message. The use of #pragma simd tells the compiler to assume that the loop is safe for vectorization, in contrast to vectorization "hints" which require the compiler to be conservative and prove that the loop is safe for vectorization—leaving many vectorization opportunities unexploited.

Array Notation

Intel Cilk Plus includes a set of notations that allow users to express high-level operations on entire arrays or sections of arrays. These notations help the compiler to effectively vectorize the application. Array notation can be used for both fixed-length and variable-length arrays. Array notation uses the following syntax:

```
pointer [ lower-bound : length : stride ]
```

Each of the three colon-separated section parameters can be any integer expression. The user is responsible for guaranteeing that the section comprises elements within the bounds of the array object. Each section parameter can be omitted if the default value is sufficient.

- The default lower-bound is 0. If the lower-bound is defaulted, then length and stride must also be defaulted.
- The default length is the length of the array. If the compiler cannot determine the length of the array from its context, length must be specified. If length is defaulted, then stride must also be defaulted.
- The default stride is 1. If the stride is defaulted, the second ":" must be omitted.

It is important to note that length is the number of steps that will be made in the array section. It is the programmer's responsibility to guarantee that `(length * stride) + lower-bound` is within the bounds of the array.

Here are some samples of array notation, with the corresponding serial C or C++ equivalents:

Array Notation

```
int a[20], b[20], c[20];

// Initialize array
a[:] = 5;


// Copy array, adding 5
b[:] = a[:] + 5;


// Set even elements
b[0:10:2] = 15;


// Sum two arrays
c[:] = a[:] + b[:];


// Call a function for
// each element of array
func(a[:]);
```

Scalar C/C++ Equivalent

```
int a[20], b[20], c[20];

// Initialize array
for (int i = 0; i < 20; i++)
    a[i] = 5;

// Copy array, adding 5
for (int i = 0; i < 20; i++)
    b[i] = a[i] + 5;

// Set even elements
for (int i = 0; i < 20; i += 2)
    b[i] = 15;

// Sum two arrays
for (int i = 0; i < 20; i++)
    c[i] = a[i] + b[i];

// Call a function for
// each element of array
for (int i = 0; i < 20; i++)
    func(a[i]);
```

Array sections can be combined with if statements, the C/C++ conditional operator, or case statements. For example,

```
if (a[0:n] > b[0:n])
    c[0:n] = a[0:n] - b[0:n];
else
    c[0:n] = 0;
```

will set each element `c[i]` to either `a[i] – b[i]`, or `0`, depending on whether `a[i] > b[i]`.

Intel Cilk Plus includes reduction functions that operate on an array section and return a scalar result. Built-in reduction functions return the sum, product, max and min of the elements, the index of the min or max element, whether all or none of the elements are zero, and whether any elements are zero or are not zero.

SIMD-Enabled Functions

A *SIMD-enabled function* is a function which can be invoked either on scalar arguments, on array sections, or on array elements processed in a vectorized loop. SIMD-enabled functions are specified by using the annotation `__declspec(vector)` on Windows* or `__attribute__((vector))` on Linux* or Mac* OS to indicate that the compiler should generate both scalar and SIMD versions of the function. When called with an array section, a SIMD-enabled function will be passed a vector-width of elements to operate on simultaneously. For example:

```
__declspec(vector) func(double a);

func(a[:]);
```

Putting It All Together

The **Karatsuba algorithm (http://en.wikipedia.org/wiki/Karatsuba_algorithm)** for polynomial multiplication uses a divide and conquer approach, recursively breaking two polynomials into halves and multiplying the halves until it reaches a small enough pair of polynomials which it just multiplies. The sample code provides four implementations of the Karatsuba algorithm; a serial implementation, an implementation vectorized using array notation, an implementation parallelized using the Intel Cilk Plus keywords, and an implementation which is both parallelized with the Intel Cilk Plus keywords and vectorized using array notation. The combined implementation shows the synergy between task and data parallelism. It's also worth mentioning that the speedup of the vectorized versions is so slight because the Intel compiler's auto-vectorization is already doing a good job on this application.

```
Testing Karatsuba implementations... Validated

Starting speed tests. Parallel runs will use 8 threads...

Timing 2048 multiplications of 10000-degree polynomials

              Version  Time  Speedup
               Serial 18.267  1.00 x
           Vectorized 16.334  1.12 x
             Parallel  2.792  6.54 x
 Parallel/Vectorized  2.590  7.05 x
```

Compiler (Intel64)	Compiler Options	System Specifications
Intel® C++ Composer XE 2013 SP1 for Windows, Update 2	Release Configuration /O3 /Qipo /fp:fast /QxHost	Windows* 7 2 Intel® Xeon® E5520 CPUs @ 2.27GHz 6GB memory Hyper-threading disabled in BIOS

The source for this implementation of the Karatsuba algorithm is available at the download page of the Intel Cilk Plus website: **http://www.cilkplus.org/download**.

Beyond GCC

In addition to the Intel Cilk Plus implementation in GCC, Intel has announced that it is working on an implementation of Intel Cilk Plus in LLVM/Clang. The implementation has been posted on GitHub and is available at **http://cilkplus.github.io/**. As I write this, Intel Cilk Plus/LLVM is still in development.

The Intel Cilk Plus team at Intel has been working on an implementation of software pipelines in Intel Cilk Plus with the MIT group that originally developed Cilk. A prototype is posted at the Intel Cilk Plus website in the **Experimental Software** section.

Getting Started

Users of Intel Composer XE can simply add Intel Cilk Plus statements to their programs. The compiler will automatically link against the Intel Cilk Plus runtime if needed.

An article on how to download and build GCC 4.9 is available at the Intel Cilk Plus website: **http://www.cilkplus.org/build-gcc-cilkplus**. GCC users need to use the following options to enable Intel Cilk Plus features and link against the Intel Cilk Plus runtime:

```
g++ -fcilkplus –lcilkrts
```

Information on how to download and build the Intel Cilk Plus branch of LLVM/Clang is available at: **http://cilkplus.github.io/**.

Further Reading

Cilk Plus community website (**http://www.cilkplus.org/**): Cilk Plus information, sample applications, public libraries, the latest sources, and more.

Intel® Cilk™ Plus Tutorial (**http://www.cilkplus.org/cilk-plus-tutorial**): An introduction to using Intel Cilk Plus.

C/C++ Extensions for Array Notations Programming Model (**https://software.intel.com/sites/products/documentation/studio/composer/en-us/2011Update/compiler_c/optaps/common/optaps_par_cean_prog.htm**): An in-depth exploration of using array notations, SIMD-enabled functions, and #pragma simd.

Intel® Cilk™ Plus for LLVM/Clang (**http://cilkplus.github.io/**): The Intel Cilk Plus/LLVM development page. Shows the status of the implementation and has links and information on how to build Intel Cilk Plus/LLVM.

Intel® C++ Compiler Code Samples (**http://software.intel.com/en-us/code-samples/intel-c-compiler**): Examples of C++ using explicit vectorization and parallelization.

1. As this is being written, cilk_for has not yet been accepted into GCC 4.9. Intel is working with the GCC community to complete the implementation of this feature of Cilk Plus in GCC