Flow Graphs, Speculative Locks, and Task Arenas in Intel® Threading Building Blocks

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

Letter from the Editor
Speaking in Code
By James Reinders

Graduate from MIT to GCC Mainline with Intel® Cilk™ Plus
Intel® Cilk™ Plus provides a higher level of abstraction than other threading frameworks. This article explores its capabilities for expressing task and data parallelism.

Flow Graphs, Speculative Locks, and Task Arenas in Intel® Threading Building Blocks
A look at Intel® TBB features, including the flow graph interface, speculative locks that take advantage of the Intel® Transactional Synchronization Extensions (Intel® TSX) technology, and user-managed task arenas that provide enhanced concurrency control and work isolation.

20 Years of the MPI Standard: Now with a Common Application Binary Interface
Examines MPI compatibility issues and resolution, as well as the potential of the upcoming common MPI Application Binary Interface (ABI).

Mastering Performance Challenges with the New MPI-3 Standard
Find out how to measure the overlap of communication and computation, and how an MPI application can benefit from nonblocking collective communication.

An OpenMP® Timeline
An infographic time-capsule of OpenMP®.

Leverage Your OpenCL™ Investment on Intel® Architectures
Get more out of OpenCL™—from cross-device portability to the Intel® Xeon Phi™ coprocessor.
LETTER FROM THE EDITOR

James Reinders, Director of Parallel Programming Evangelism at Intel Corporation.


Speaking in Code

Mathematics is no longer the only universal language. Developers and engineers worldwide are sharing programming languages and models that prove the maxim that “necessity drives invention.” In this issue, we look at the rapid pace of change in our common languages and some of the newest possibilities for application innovation.

We open with two co-feature articles. Graduate from MIT to GCC Mainline with Intel® Cilk™ Plus looks at the evolution and current state-of-the-art of Intel® Cilk™ Plus—a set of extensions to C and C++ used to express task and data parallelism. Flow Graphs, Speculative Locks, and Task Arenas in Intel® Threading Building Blocks examines new features and the enhanced cross-platform support that allows applications to port to many different OSes or architectures without rewriting and re-debugging core algorithms.

20 Years of the MPI Standard: Now with a Common Application Binary Interface looks at the potential of the MPI Application Binary Interface (ABI), a common standard slated for adoption in the summer of 2014, which will allow different MPI implementations to compete based on performance alone.

Mastering Performance Challenges with the New MPI-3 Standard demonstrates the performance benefits of the MPI-3 nonblocking collective operations supported by Intel® MPI Library 5.0 and Intel® MPI Benchmarks (IMB) 4.0.

An OpenMP* Timeline encapsulates a brief history of OpenMP in an infographic.

Finally, Leverage Your OpenCL™ Investment on Intel® Architectures outlines some of the performance portability considerations for the Intel® Xeon Phi™ coprocessor.

We hope you find the techniques presented in this issue valuable, as you contribute to the shared language of software. Whether your applications power the software-defined datacenter, responsive cross-platform experiences, or the next breakthrough in APIs, we salute your pioneering efforts to move software forward.

James Reinders
June 2014
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.

```c
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:

```cpp
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:

```cpp
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 \( \text{kk} \). The call to \texttt{__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.

**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 \((\text{length} \times \text{stride}) + \text{lower-bound}\) is within the bounds of the array.

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

<table>
<thead>
<tr>
<th>Array Notation</th>
<th>Scalar C/C++ Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>int a[20], b[20], c[20];</td>
<td>int a[20], b[20], c[20];</td>
</tr>
<tr>
<td>// Initialize array</td>
<td></td>
</tr>
<tr>
<td>a[: ] = 5;</td>
<td>// Initialize array</td>
</tr>
<tr>
<td>for (int i = 0; i &lt; 20; i++)</td>
<td></td>
</tr>
<tr>
<td>a[i] = 5;</td>
<td></td>
</tr>
<tr>
<td>// Copy array, adding 5</td>
<td></td>
</tr>
<tr>
<td>b[: ] = a[: ] + 5;</td>
<td>// Copy array, adding 5</td>
</tr>
<tr>
<td>for (int i = 0; i &lt; 20; i++)</td>
<td></td>
</tr>
<tr>
<td>b[i] = a[i] + 5;</td>
<td></td>
</tr>
<tr>
<td>// Set even elements</td>
<td></td>
</tr>
<tr>
<td>b[0:10:2] = 15;</td>
<td>// Set even elements</td>
</tr>
<tr>
<td>for (int i = 0; i &lt; 20; i += 2)</td>
<td></td>
</tr>
<tr>
<td>b[i] = 15;</td>
<td></td>
</tr>
<tr>
<td>// Sum two arrays</td>
<td></td>
</tr>
<tr>
<td>c[: ] = a[: ] + b[: ];</td>
<td>// Sum two arrays</td>
</tr>
<tr>
<td>for (int i = 0; i &lt; 20; i++)</td>
<td></td>
</tr>
<tr>
<td>c[i] = a[i] + b[i];</td>
<td></td>
</tr>
<tr>
<td>// Call a function for</td>
<td></td>
</tr>
<tr>
<td>// each element of array</td>
<td></td>
</tr>
<tr>
<td>func(a[: ]);</td>
<td>// Call a function for</td>
</tr>
<tr>
<td>// each element of array</td>
<td></td>
</tr>
<tr>
<td>for (int i = 0; i &lt; 20; i++)</td>
<td></td>
</tr>
<tr>
<td>func(a[i]);</td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Version</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>18.267</td>
<td>1.00 x</td>
</tr>
<tr>
<td>Vectorized</td>
<td>16.334</td>
<td>1.12 x</td>
</tr>
<tr>
<td>Parallel</td>
<td>2.792</td>
<td>6.54 x</td>
</tr>
<tr>
<td>Parallel/Vectorized</td>
<td>2.590</td>
<td>7.05 x</td>
</tr>
</tbody>
</table>

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](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/](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](http://www.cilkplus.org) 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 -lcilkrt
```

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 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.


---

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.
Flow Graphs, Speculative Locks, and Task Arenas in Intel® Threading Building Blocks

By Alexey Kukanov, Software Architect, Vladimir Polin, Software Engineering Manager, and Michael J. Voss, Software Architect, Threading Runtimes, Developer Products Division, Intel Software and Services Group

Introduction

The Intel® Threading Building Blocks (Intel® TBB) library provides software developers with a solution for enabling parallelism in C++ applications and libraries. The well-known advantage of the Intel TBB library is that it makes parallel performance and scalability easily accessible to software developers writing loop- and task-based applications. The library includes (Figure 1) a number of generic parallel algorithms, flow graph interface, a work-stealing-based task scheduler, concurrent containers, scalable memory allocation, thread local storage, and synchronization primitives. By mapping essential parallel patterns to these building blocks, developers build robust parallel applications that abstract platform details and threading mechanisms, while achieving performance that scales with increasing core count.
Generic Parallel Algorithms
Efficient scalable way to exploit the power of multi-core without having to start from scratch

Concurrent Containers
Concurrent access, and a scalable alternative to containers that are externally locked for thread-safety

Flow Graph
A set of classes to express parallelism via a dependency graph or a data flow graph

Thread Local Storage
Supports infinite number of thread local data

Task Scheduler
Sophisticated engine with a variety of work scheduling techniques that empowers parallel algorithms and the flow graph

Synchronization Primitives
Atomic operations, several flavors of mutexes, condition variables

Thread-safe timers
Threads
OS API wrappers

Memory Allocation
Scalable memory manager and false-sharing free allocators

1 Intel® Threading Building Blocks features.

Structured Parallel Programming1 describes a number of useful parallel patterns. Intel TBB algorithms map to those patterns as follows:

parallel_for: map
parallel_reduce, parallel_scan: reduce, scan
parallel_do: workpile
parallel_pipeline: pipeline
parallel_invoke, task_group: fork-join
flow_graph: plumbing for reactive and streaming apps
The Intel TBB library can be used with any C++ compliant compiler. It also supports C++11 features, such as lambda expressions, that reduce coding efforts by removing the need to implement special functor classes. Different Intel TBB library components can be used independently, and even mixed with other threading technologies.

Intel TBB version 4.2 introduces new features and improvements that include support for the latest Intel® architecture features, such as the Intel® Transactional Synchronization Extensions technology (Intel® TSX) and Intel® Xeon Phi™ coprocessors for Windows* OS. It also provides support for Windows Store* and Android® applications2,3 and introduces some new functionality, as well as adds a number of improvements to its popular features (for more details, see the library documentation). Intel TBB 4.2 is available as a standalone product and is also a component of software bundles such as Intel® INDE, Intel® Cluster Studio XE, Intel® Parallel Studio XE, Intel® C++ Studio XE, Intel® Composer XE, and Intel® C++ Composer XE.

The open source version of the Intel TBB library supports more architectures, compilers, and operating systems, thanks to contributions from the community. It can work with the Solaris* OS with the Oracle* C++ compiler for Intel architecture and SPARC*-compatible processors, IBM Blue Gene* supercomputers and PowerPC*-compatible architectures, Android OS, Windows Phone* 8 OS and the Windows RT* OS for ARM*-compatible architectures, FreeBSD* OS, Robot* Operating System, and many other platforms where GCC* built-in atomics are supported. This broad cross-platform support means that once an application is written using the Intel TBB library, it can be ported to many different OSes or architectures without rewriting and re-debugging the core algorithms.

In the remainder of this article, we focus on a few specific features of the library. First, we provide an overview of the flow graph interface available since Intel TBB 4.0. We then present two new features introduced in Intel TBB 4.2: speculative locks that take advantage of the Intel Transactional Synchronization Extensions (Intel TSX) technology and user-managed task arenas that provide enhanced concurrency control and work isolation.

Expressing Dependency and Data Flow Graphs

While the Intel TBB library is well known for its loop-based parallel algorithms, the flow graph interface4 extends its capabilities to allow fast, efficient implementations of dependency graph and data flow algorithms, enabling developers to exploit parallelism at higher levels in their application.
For example, consider a simple application that invokes four functions sequentially, as shown in Figure 2(a). A loop-parallel approach would involve examining each of these functions, looking for parallelism that can be exploited with algorithms such as `parallel_for` and `parallel_reduce`. In some cases this is sufficient for good performance; but in others, there may not be enough parallelism to exploit at that level. For example, Figure 2(b) shows the application after loop-parallelism is found and exploited in functions B and D. The execution time will improve, but what if a greater level of performance is still desired?

Sometimes, invoking functions one after the other can be overly constrained: a total order must be chosen for invocations, even if only a partial order is required. In Figure 2(a), the functions were ordered by the developer to ensure that each function executes after all of the values it needs have been computed, but what if functions B and C both require the output generated by A, but C does not depend on the output of B? Figure 2(c) shows the graph- and loop-parallel implementation of the example. In this implementation, the loop-level parallelism is exposed and the total ordering is replaced with a partial ordering that would allow B and C to execute concurrently.

The Intel TBB flow graph interface allows developers to easily express graph parallelism. It provides features to support the dependency, streaming, and data flow graphs that can be found in many domains, such as media, gaming, finance, high performance computing, and healthcare. This interface is a fully supported feature of the library, and has been available since Intel TBB 4.0.
When using a flow graph, computations are represented by nodes and the communication channels between these computations are represented by edges. The user is responsible for using edges to express all dependencies that must be respected when nodes are scheduled for execution, giving the Intel TBB scheduler the flexibility to exploit the\textit{parallelism} that is explicit in the graph topology. When a node in the graph receives a message, an Intel TBB task is spawned to execute its body object on the incoming message.

The flow graph interface supports several different types of nodes (Figure 3), including functional nodes that execute user-provided body objects, buffering nodes that can be used to order and buffer messages as they flow through the graph, aggregation and deaggregation nodes that join and split messages, and other special purpose nodes. Users connect instances of these node types together with edges to specify the dependencies between them and provide body objects to perform their work.

Below is the source code for a simple “Hello World” flow graph application. This example is very simple and does not contain any \textit{parallelism}, but it does demonstrate the syntax of the interface. In this code, two nodes are created, hello and world, which are constructed with lambda expressions that print “Hello” and “World” respectively. Each node is of type \texttt{continue\_node}, a functional node type provided by the interface. The \texttt{make\_edge} call creates an edge between the \texttt{hello} node and the \texttt{world} node. Whenever a task spawned by the \texttt{hello} node completes, it will send a message to the \texttt{world} node, causing it to spawn a task to execute its lambda expression.
In the code above, the call to `hello.try_put(continue_msg())` sends a message to the `hello` node, causing it to spawn a task to execute its body object. When that task completes, it sends a message to the `world` node. Only when all of the tasks spawned by the nodes are complete, does the call to `g.wait_for_all()` return.

The Intel TBB flow graph interface enables the expression of very complex graphs that may include thousands of nodes and edges, cycles, buffering, and more. Figure 4 shows a visual representation of two flow graph implementations of Cholesky Factorization that use an algorithm similar to that described in Performance Evaluation of Concurrent Collections on High-Performance Multicore Computing Systems. In Figure 4(a), each invocation of the Intel® Math Kernel Library (Intel® MKL) functions `dpotf2m, dtrsm, dgemm` and `dsyrk` is a distinct node in the flow graph. In this case, the graph is large with many nodes per tile of the matrix, but is easily created by modifying the original sequential Cholesky Factorization loop nest, replacing function invocations with node and edge creations. In this graph, the edges are used to communicate before-after relationships; each node must wait until all of its predecessors are complete. It is easy to see the parallelism available in this graph.

Figure 4(b) shows an alternative implementation that is also possible to express with the flow graph interface. This version is a small, compact flow graph, where there is only a single node for all invocations of each Intel MKL function type. In this implementation, the tiles are passed as messages between nodes in the graph. When a node receives a set of matching tiles, it spawns a task to apply its body object to this set of tiles and then forwards the new tile it generates on to other nodes. The parallelism in this graph comes from the library’s ability to execute multiple instances of each flow graph node concurrently.
While the details of the implementations described in Figure 4 are beyond the scope of this article, this example demonstrates that the Intel TBB flow graph interface is a powerful and flexible abstraction. It can be used to create large, directed-acyclic graphs such as that shown in Figure 4(a), where the developer chose to create a separate node for each Intel MKL call. And, it can also be used to create compact data flow graphs that include cycles, joins, and conditional executions as shown in Figure 4(b).

More information on the flow graph interface is available in the Intel TBB Reference Manual.

Speculative Locks

Intel TBB 4.2 features speculative locks: new synchronization classes based on Intel TSX technology. A speculative lock may allow protected critical sections to proceed simultaneously, under an assumption that data accesses and modifications do not conflict with each other. If in practice a conflict occurs, one or more of the speculative executions should be invalidated, leaving the protected data intact and thus remaining invisible to other threads. Threads involved in a conflict will then repeat their critical sections, possibly taking a real lock for data protection (as Intel TSX technology does not guarantee that speculative execution will eventually succeed).

With the Intel TBB implementation of speculative locks, all these steps happen under the hood and the programmer can use the customary mutex API. Moreover, on a processor that does not support Intel TSX technology, the implementation will automatically fall back to a regular lock, thus allowing developers to write portable programs that may benefit from transactional synchronization.

The Intel TBB library currently provides two mutex classes that support Intel TSX technology: speculative_spin_mutex and speculative_spin_rw_mutex. The latter was added as a preview feature in Intel TBB 4.2 update 2.

The class speculative_spin_mutex is API-compatible and very similar in nature to the class spin_mutex; both reside in the same header file tbb/spin_mutex.h. The main difference of speculative_spin_mutex, besides the ability to utilize Intel TSX, is its size. In order to avoid sharing of a cache line with any other data, which would likely result in a high degree of execution conflicts and related performance loss, an instance of speculative_spin_mutex occupies two cache lines.
Here is a code sample that uses an Intel TBB speculative lock:

```c++
#include <tbb/spin_mutex.h>
#include <set>

tbb::speculative_spin_mutex tsx_mtx;
std::set<int> g_Set;

void thread_safe_add_to_set( int value ) {
    tbb::speculative_spin_mutex::scoped_lock lock(tsx_mtx);
    g_Set.insert(value);
}
```

The class `speculative_spin_rwlock`, as you may guess from its name, implements a read-write spin lock with speculative execution. One may note that any speculative lock is by definition non-exclusive, allowing not only reads, but even non-conflicting writes being done simultaneously. So “speculative RW lock” may sound like a tautology. Remember, however, that a thread may acquire the lock “for real.” In this case, `speculative_spin_mutex` has to provide exclusive access for the thread, no matter whether it reads or modifies the data; therefore, it is not really a read-write lock. On the other hand, `speculative_spin_rwlock` will always allow multiple readers to proceed. Moreover, both real and speculative readers may run simultaneously. But this comes with a cost: internal data fields of the class need to be kept at separate cache lines. Together with false sharing avoidance, these lead to an instance of `speculative_spin_rwlock` occupying three cache lines.

Though `speculative_spin_rwlock` currently resides in `tbb/spin_rwlock.h` header file, and even uses `spin_rwlock` as part of the implementation, the two classes are not fully compatible. There are no `lock()` and `unlock()` methods in `speculative_spin_rwlock`. Rather it enforces a scoped locking pattern, that is, it has to be accessed via class `speculative_spin_rwlock::scoped_lock`. Since it is a preview feature, a TBB_PREVIEW_SPECULATIVE_SPIN_RW_MUTEX macro should be defined to a non-zero value prior to including the header file.

Unfortunately, the applicability and benefit of speculative locks are rather problem-specific. You should not think of these new classes as “better spin locks”—careful performance measurement is necessary to decide if speculative locks are the right instrument for your use cases.
User-Managed Task Arenas

Another significant piece of new functionality recently added to the Intel TBB library is user-managed task arenas.

In our terminology, an arena is a place for threads to share and steal tasks. Initially, the library supported just one global arena for an application. Later, we changed it to maintain separate arenas for each application thread, motivated by feedback that work submitted by different threads should be better isolated. Then, we received requests for concurrency control and work isolation to not be tied to application threads. To address this, we introduced user-managed task arenas. Currently, it is still a preview feature (and requires the TBB_PREVIEW_TASK_ARENA macro to be non-zero), but we are working toward making full support available later in 2014.

The API to user-managed arenas is provided by class task_arena. When creating a task_arena, a user may specify its desired concurrency and how much of this concurrency should be reserved for application threads.

```c
#define TBB_PREVIEW_TASK_ARENA 1
#include <tbb/task_arena.h>
tbb::task_arena my_arena(4, 1);
```

In the given example, an arena is created for four threads—where one place is always reserved for an application thread. It means that up to three worker threads managed by the Intel TBB library can join this arena and work on the tasks shared there. There is no limit on the number of application threads submitting jobs to the task_arena, but the overall concurrency will still be limited by four. All “extra” threads will be unable to join the arena and execute tasks there.

In order to submit some work to an arena, one should call its `execute()` or `enqueue()` methods:

```c
my_arena.enqueue( a_job_functor );
my_arena.execute( a_job_functor2 );
```

A job for any of these methods can be represented by a C++11 lambda expression or an instance of a functor class. The two methods differ in the way the job is submitted to the arena. `task_arena::enqueue()` is an asynchronous call for submitting a fire-and-forget job; the caller thread does not join the arena and returns immediately. In contrast, `task_arena::execute()` does not return until the submitted job is completed. If possible, the caller thread joins the arena and executes tasks there, otherwise it is blocked for the time necessary to complete its job.
While it is possible to submit a lot of serial job pieces into a task_arena, it is not an intended use case. Typically, one submits a job that creates sufficient **parallelism**, for example, with a parallel_for call:

```cpp
my_arena.execute( [&] {
    tbb::parallel_for(0,N,iteration_functor());
});
```

or with a flow graph:

```cpp
tbb::flow::graph g;
... // create the graph here
my_arena.enqueue( [&] {
    ... // start graph computations
});
... // do something else
my_arena.execute( [&] {
    g.wait_for_all();
}); // does not return until the flow graph finishes
```

More information on task arenas is available in the Intel TBB Reference Manual.

**Summary**

The Intel TBB C++ template library offers a rich set of components to efficiently exploit higher-level, task-based **parallelism** and implement portable future-proof applications to tap the power of multicore and many-core. The library lets developers focus on expressing the parallelism in their applications without focusing on the low-level details of managing that parallelism. In addition to efficient, high-performing implementations of the most commonly used high-level parallel algorithms and concurrent containers, the library provides efficient low-level building blocks such as a scalable memory allocator, locks, and atomic operations.

While the Intel TBB library is a mature technology, we continue to improve its performance and expand its capabilities. In the Intel TBB 4.0 release, we introduced the flow graph to allow developers to more easily express dependency and data flow graphs. In the Intel TBB 4.2 release, we responded to new architectural features by introducing new synchronization classes based on Intel TSX technology and responded to user requests for better control of concurrency and isolation by introducing user-managed task arenas.
You can find recent Intel TBB versions and information on the commercial and open source sites.


Intel TBB open source site: http://threadingbuildingblocks.org


References


The Parallel Universe

Try Intel® Threading Building Blocks (Intel® TBB) >

BLOG HIGHLIGHTS

Resource Guide for Intel® Xeon Phi™ Coprocessor Developers

BY TAYLOR KIDD »

This article identifies resources for software developers as they begin working with the Intel® Xeon Phi™ coprocessor, which is based on the Intel® Many Integrated Core (Intel® MIC) architecture. It is one of three such guides designed for people in one of the following roles:

- **Administrator** refers to a person responsible for administration of one or more servers equipped with the Intel Xeon Phi coprocessor (including clusters of such servers).
- **Developer** refers to a person programming for systems equipped with the Intel Xeon Phi coprocessor.
- **Investigator** refers to anyone else who needs to learn more about the Intel Xeon Phi coprocessor, particularly those deciding whether an organization should adopt the technology.

Each guide focuses on the resources most likely to be of primary interest to people in that role. For example, documentation on maintaining clusters is potentially of interest to an administrator, but far less likely to be valuable to a developer. Likewise, programming syntax and semantics are important to a developer, but typically not to an administrator. The content in each guide is tailored accordingly.

More
20 Years of the MPI Standard
Now With a Common Application Binary Interface

By Alexander Supalov Intel Cluster Tools Architect, and Artem Yalozo, Software Engineer, Intel Corporation

Introduction

Message Passing Interface (MPI)—the de facto industry standard for distributed memory computing—celebrates its 20th anniversary this year. There are few interfaces that can compete with MPI on the critical performance required for high performance computing (HPC).

MPI has come a long way since its first introduction by the MPI Forum in May 1994. In September 2012, a major new version, the MPI-3 standard, was voted into existence. The latest version of the standard adds fast one-sided communication, non-blocking collective operations, and quite a few other features.

Based on this new standard, Intel Corporation, in cooperation with Argonne National Laboratory, Cray Corporation, and IBM Corporation, formed an initiative that introduced the long-desired common MPI Application Binary Interface (ABI) at SuperComputing 2013. This ABI will be available in Intel® MPI Library 5.0, shipping in June 2014, as in a number of compatible third-party products, including MPICH*, Cray MPI*, and IBM POE*.
Motivation

Compatibility has long been a key concern for the Intel MPI Library, a multi-fabric message passing library that is used on high performance computing clusters. Our customers have always expected it to preserve compatibility, even between major product releases. This allowed for a binary-compatible upgrade path to a subsequent version of the product library, making the transitions to new versions easier and more successful. Without ABI compatibility, any released software building on top of an existing version needs to be re-released for every new product version.

Reasons for Potential Incompatibility

The root cause for prior MPI library incompatibilities stemmed from the use of different source bases. Different versions were based on different MPICH source codes and implemented different versions of the MPI standard. For example, the basis for Intel MPI 4.1 was MPICH2, and hence it implemented the MPI-2.2 standard. The basis for Intel MPI 5.0 is MPICH3 and hence the new MPI-3 standard.

Application binary interface is the low-level interface between two program modules. It describes the binary interface to a library, and should not be confused with the Application Programming Interface (API). The ABI determines such details as how functions are called and the size, layout, and alignment of data types. With ABI, compatible programs conform to the same set of runtime conventions. A library is binary compatible if a program linked to a previous version of the library continues to work with a newer version without being recompiled.

The second kind of compatibility implies that all library functions do the same thing in the newer version that they did in older versions. Of course, there is some flexibility here. The old behavior could be at some point incorrect, so it could be changed in the new release of the software. And this could be a root cause of broken behavior compatibility. This could also be caused by the changed MPI standard.

Compatibility Scenarios, Issues, and Solutions

Ideally, both backward and forward compatibility of the software could be ensured. In practice, products provide a reasonable balance between backward compatibility and the need to implement newer standards.

The main compatibility scenario for the Intel MPI Library is the backward compatibility scenario. In this case, an application built with a new Intel MPI version runs with previous library runtime. Forward compatibility is considered, but not claimed.
ABI compatibility problems may cause crashes in the client code. They may also cause data corruption. They may even go unnoticed. Because of this, a careful investigation of potential incompatibilities should be performed before trying to catch the incompatibility crashes during testing.

There are several language bindings provided by the Intel MPI Library: C, C++, and Fortran 77/90. Changes in each interface are the potential cause for binary incompatibility. **Table 1** summarizes reasons and possible solutions for binary incompatibility cases discovered between Intel MPI Library 5.0 (based on MPICH3) and 4.1 (MPICH2).

<table>
<thead>
<tr>
<th>Reason (changes between Intel MPI 4.1 and MPICH3)</th>
<th>Language binding</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predefined constant changed: MPI_MAX_ERROR_STRING</td>
<td>All</td>
<td>Return to 4.1 ABI</td>
</tr>
<tr>
<td>Constants/new definitions added: MPI_ATTR_FAILED PROCESSES, MPI_UNWEIGHTED, MPI_ERR_LAST_CLASS</td>
<td>C</td>
<td>Include constant, take new definition</td>
</tr>
<tr>
<td>Predefined constant added/changed: MPI_COMBINER_HINDEXED_BLOCK, MPI_COMBINER_*</td>
<td>C/Fortran</td>
<td>Assign next value to HINDEXED_BLOCK, leave others unchanged</td>
</tr>
<tr>
<td>MPI_Status items order changed (count and cancelled moved down). Type of count changed from int to 64-bit MPI_Count. typedef struct MPI_Status { int MPI_SOURCE; int MPI_TAG; int MPI_ERROR; MPI_Count count; int cancelled; } MPI_Status;</td>
<td>C</td>
<td>Move count and cancelled up. Interpret 32-bit count and cancelled fields as a 63-bit count and a 1-bit cancelled values</td>
</tr>
<tr>
<td>Predefined constant added/removed: MPI_2COMPLEX, MPI_2DOUBLE_COMPLEX</td>
<td>Fortran</td>
<td>Include constant</td>
</tr>
<tr>
<td>Common blocks removed/added: MPIPRIV1, MPIPRIV2, MPIPRIVC MPICMB5, MPICMB9</td>
<td>Fortran</td>
<td>Return to 4.1 ABI, add new common block(s) for all MPI-3 things</td>
</tr>
<tr>
<td>Virtual functions table changed (order changed): virtual void Shift() Cartcomm Dup() virtual int Get_cart_rank() virtual void Get_topo() virtual int Get_dim() virtual int Map() virtual Cartcomm Sub()</td>
<td>C++</td>
<td>Return to 4.1 ABI</td>
</tr>
</tbody>
</table>

1 Binary compatibility issues and their resolution.
The Intel **MPI Library** 5.0 Beta provided ABI compatibility with Intel **MPI Library** 4.x by introducing the necessary changes in the source codes with regards to discovered compatibility issues. Originally, Intel MPI Library 5.0 Beta was not compatible with MPICH3. However, the ABI changes could be controlled during the library build stage (optionally), and an MPICH3 compatible library could still be built. However, Intel MPI 5.0 is compatible with MPICH3.

**Table 2** summarizes reasons and possible solutions for discovered behavior incompatibility between Intel MPI Library 4.1 (based on MPICH2 1.4) and 4.0 (MPICH2 1.1).

<table>
<thead>
<tr>
<th>Reason (changes between Intel MPI 4.1 and MPICH2 1.4)</th>
<th>Language binding</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Behavior for some functions changed by MPI standard:</td>
<td>C</td>
<td>Change function behavior based on I_MPI_COMPATIBILITY environment variable value</td>
</tr>
<tr>
<td>MPI_Cart_create</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Cart_map</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Graph_create</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Win_get_attr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Behavior for some collective operations changed by MPI standard:</td>
<td>C</td>
<td>Change function behavior based on I_MPI_COMPATIBILITY environment variable value</td>
</tr>
<tr>
<td>Gather</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Allgather et al.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Behavior compatibility issues and their resolution.

Compatibility in behavior for the Intel MPI Library is not provided by default because of the need to implement the newer standard that changed the behavior. However, backward compatibility in behavior can be achieved at runtime by setting up a special environment variable **I_MPI_COMPATIBILITY** to the value of 4.

As a result, within a single library the Intel MPI introduces support for the newer standard and still remains binary compatible with the majority of the applications built using older versions of the Intel MPI Library.
Conclusion

The Intel MPI Library provides compatibility with all previous versions. This is one of its significant advantages. Meanwhile, the new MPI ABI will be adopted by all partners by the summer of 2014. (For more details on the new ABI, visit the MPICH ABI Compatibility Initiative at: http://www.mpich.org/abi/.) Once this occurs, Intel MPI Library, MPICH, Cray MPI, and IBM POE will become fully interchangeable at runtime. Moreover, this ABI will allow both MPI-2.x and MPI-3 standards to be supported by one MPI library. This will preserve customer investment into the earlier MPI standards and related applications, and simplify the life of the MPI implementers by allowing them to support only one package in the field. This will also allow our binary compatibility to extend across our collaborators’ MPI implementations, which is important both to our customers and to our partners.

The ABI and the resulting ability to switch between libraries can help every developer by letting different MPI implementations compete based on performance alone. This will also open up the MPI space to the associated tools currently built for a specific MPI distribution and unable to reach outside of this box.

References

3. The Intel® MPI Library: www.intel.com/go/mpi
4. MPICH: http://www.mpICH.org
5. MPICH ABI Compatibility Initiative: http://www.mpich.org/abi/
Mastering Performance Challenges with the New MPI-3 Standard

By Mikhail Brinskiy, Software Development Engineer, Alexander Supalov, Intel Cluster Tools Architect, Michael Chuvelev, Software Manager, and Evgeny Leksikov, Intel Corporation

Introduction

This article demonstrates the performance benefits of the MPI-3 nonblocking collective operations supported by the Intel® MPI Library 5.0 and the Intel® MPI Benchmarks (IMB) 4.0 products. We'll show how to measure the overlap of communication and computation, and demonstrate how an MPI application can benefit from the nonblocking collective communication.

The Message Passing Interface (MPI) standard is a widely used programming interface for distributed memory systems. The latest MPI-3 standard contains major new features such as nonblocking and neighbor collective operations, extensions to the Remote Memory Access (RMA) interface, large count support, and new tool interfaces. The use of large counts lets developers seamlessly operate large amounts of data, the fast, one-sided operations strive to speed up Remote Memory Access (RMA)-based applications, and the nonblocking collectives enable developers to better overlap the computation and communication parts of their applications and exploit potential performance gains.
Here, we concentrate on two major MPI-3 features: nonblocking collective operations (NBC) and the new RMA interface. We evaluate the effect of communication and computation overlap with NBC, and describe potential benefits of the new RMA functionality.

For our measurements, we use Intel MPI Library 5.0 Beta³ and Intel MPI Benchmarks 4.0 Beta (IMB), both of which support the MPI-3 standard. Starting from version 4.0 Beta, IMB contains two new binaries: IMB–NBC, intended to measure the effectiveness of nonblocking collective operations, and IMB–RMA, covering the most important MPI-3 updates to the RMA interface. The complete IMB download, as well as a detailed description of all IMB benchmark suites, are available at: http://software.intel.com/en-us/articles/intel-mpi-benchmarks.

**Nonblocking Collective Operations (NBC)**

NBC semantics allow an MPI user to overlap communication with computation, like nonblocking point-to-point operations already do. Also, NBC mitigate the effects of pseudo-synchronization that is inherent in many collective algorithms due to data dependencies. A detailed analysis of the nonblocking collective operations potential is available at: http://htor.inf.ethz.ch/publications/img/hoefler-nbc-standard.pdf.

A common assumption about the nonblocking operations (collective, as well as point-to-point) is that the progress happens in the background without user intervention.¹ However, the MPI standard does not define an explicit progression rule and leaves it up to a so-called “high quality” implementation to offer true asynchronous progress. Thus, many early MPI libraries did not offer asynchronous progression and just performed all communication in the intervening MPI calls. With the advent of NBC, a user of MPI is free to use different approaches to overlap a collective operation with computation: either periodically calling a `MPI_Test()` routine to ensure explicit message progression, or using a single `MPI_Wait()` call to complete the operation.

The IMB methodology to measure the effectiveness of communication and computation overlap is designed to match the most popular use of nonblocking communication, specifically in cases when some activity is done between the operation start and its completion by the test or wait calls. This methodology assumes the communication and computation times to be approximately equal, in order to minimize the inaccuracy induced by the network or operating system (OS) noise. IMB makes the CPU busy for the desired amount of time by calculating a fully vectorized 100x100 matrix by vector product. Thus, in a nutshell, the benchmark flow looks as follows:

1. Measure the time needed for a pure communication call (e.g., `MPI_Ibcast()` followed by `MPI_Wait()`)
2. Start communication (e.g., call `MPI_Ibcast()`)
3. Start computation with duration equal to the time measured in step 1. Thus we ensure that communication and computation parts consume approximately the same amount of time.
4. Wait for communication to finish (i.e., call `MPI_Wait()`)
Given the description above, the IMB-NBC benchmarks output four timings:

- **time_pure** is the time for nonblocking operation, which is executed without any concurrent CPU activity;
- **time_CPU** is the time of the test CPU activity (which is supposed to be close to the time_pure value);
- **time_ovrlp** is the time for nonblocking operations concurrent with CPU activity;
- **overlap** – the estimated overlap percentage obtained by the following formula:
  \[ \text{overlap} = 100 \times \max(0, \min(1, \frac{(\text{time_pure} + \text{time_CPU} - \text{time_ovrlp})}{\max(\text{time_pure}, \text{time_CPU})})) \]

The Intel MPI Library supports asynchronous message progressing in the multithreaded library only, so we use it in our experiments to leverage the performance potential of the nonblocking collective operations. To enable asynchronous progress in the Intel MPI Library, the environment variable MPICH_ASYNC_PROGRESS should be set to 1.

In our experiment, we compare the results of the IMB-NBC benchmarks measuring overlap potential of nonblocking collectives with the regular blocking collective operations results obtained using the IMB-MPI1. The aim is to check which is more efficient: to perform a blocking collective operation and run computation afterward, or to use its nonblocking counterpart and overlap it with the same amount of computation. Thus, the charts reflect two timings: the time taken by the nonblocking operation concurrent with the CPU activity (which is time_ovrlp value in the IMB-NBC output), and the aggregated communication and computation time. The latter is the sum of the blocking collective operation time (measured by the IMB-MPI1 benchmark) and the time of the test CPU activity (which is the time_CPU value in the IMB-NBC output).

The results shown in these charts have been collected using 48 processes on 4 IVT nodes (12 processes per each node). Each node is equipped with two Intel® Xeon® processors E5-2697 and one Mellanox Connectx-3 InfiniBand* adapter.

The overlap obtained by the ialltoall and iallgather benchmarks is close to being ideal. **Figure 1** and **Figure 2** reflect only big message sizes as the most representative ones. Even though the nonblocking operations with asynchronous progress enabled cause significant overhead, especially noticeable at small message sizes, such an overlap is more efficient than the regular blocking collective followed by the computation. However, this is not always the case. **Figure 3** depicts the results for the broadcast benchmark and small messages sizes. While the obtained overlap is quite good (about 60–70%), the regular blocking broadcast performs much faster, which makes it more performance efficient on message sizes of up to 2KB.
1. MPI_Ialltoall overlapped with computation vs. MPI_Alltoall followed by computation.

2. MPI_Iallgather overlapped with computation vs. MPI_Allgather followed by computation.

3. MPI_Ibcast overlapped with computation vs. MPI_Bcast followed by computation.
Combining these results, we reach the following conclusions:

- Nonblocking collective operation may provide significant performance benefits in comparison with its blocking counterparts when overlapped with computation, mostly on medium and large message sizes.
- In some cases, the overhead introduced by the nonblocking operation itself and asynchronous message progressing may negate the benefit of the overlap. This is mostly relevant for small message sizes, where latency is usually an important factor.

**New RMA interface**

MPI-3 standard significantly extends the RMA interface by introducing several major features, including different memory models, several new communication and synchronization calls, and new ways of creating RMA windows. Support for different memory models is one of the most important novelties intended to distinguish situations when cache coherency is supported (unified model) and when it is not supported (separate model, typical of the MPI-2 standard). This distinction enables high-performance MPI implementations and easier programming. The major update of the synchronization semantics is intended to align the passive target communication mode with the current HPC needs. The main problems of the previous RMA interface that MPI-3 standard tries to address are described at: [http://upc.lbl.gov/publications/bonachea-duell-mpi.pdf](http://upc.lbl.gov/publications/bonachea-duell-mpi.pdf).

Prior to version 4.0 Beta, IMB provided a set of RMA benchmarks that only utilize active communication mode. These benchmarks are available in the IMB-EXT module. IMB 4.0 Beta concentrates on the new MPI-3 RMA functionality and passive target communication mode. The new IMB-RMA module contains 19 benchmarks measuring new atomic functions, halo exchanges using the `MPI_Put()` and `MPI_Get()` operations, and concurrent RMA calls to different targets. All of these benchmarks utilize the passive target communication mode.

One of the new RMA benchmarks demonstrates whether MPI implementation supports the natural passive mode of the communication, in which the target process should not necessarily call MPI for the origin process to complete its access epoch. This may be supported via the asynchronous message progressing described above, or by utilizing underlying hardware capabilities that support communication offload in some way. In Intel MPI Library 5.0 Beta, RMA operations are implemented using regular point-to-point messages; therefore the operation needs to be progressed on the target, as well as on the origin, to ensure its completion. But as stated above, asynchronous message progressing is supported by the multithreaded version of the Intel MPI Library. The single-threaded version of the library does not support the real passive target mode. Therefore, the target must call MPI to ensure operation progressing. In our test, we compare the results obtained with the enabled asynchronous progress and the results collected with the single-threaded library to evaluate the benefits of the truly passive mode support in the Intel MPI Library.
The benchmark described above measures two timings:

1. The time consumed by the origin process to complete the `MPI_Put()` operation, while the target process is sitting in `MPI_Barrier()` call (therefore ensuring progress).

2. The time consumed by the origin process to complete the `MPI_Put()` operation, while the target process performs some computations outside the MPI stack, and then invokes `MPI_Barrier()`. The computation lasts for about the same time as is needed for the origin process to complete step 1. To achieve this, the origin process sends the timing value needed for the `MPI_Put()` completion to the target process.

The target and the origin processes are synchronized prior to each `MPI_Put()` operation. Thus, in case of absence of direct communication offload capabilities or a separate thread for message progressing, the time obtained in step 2 should be noticeably greater than the time obtained in step 1 (about 2x). But if the MPI implementation provides truly passive mode for RMA, these timings should be about the same.

4. Influence of asynchronous progress on the results of the truly passive put benchmark (small to medium message sizes).

5. Influence of asynchronous progress to the results of the truly passive put benchmark (large message sizes).
The charts in Figure 4 and Figure 5 show that enabling asynchronous message progressing brings very significant overhead, which is clearly seen on message sizes of up to 128 KB. However, on big message sizes the overhead is not really noticeable, while the computation performed on the target does not delay the completion of the MPI_Put() operation on the origin.

We analyzed the behavior of the passive mode communication model with the Intel MPI Library 5.0 Beta and observed that while the truly passive mode is supported by the library with asynchronous progress enabled, there are still significant improvement opportunities. These include native support for RDMA progress and various performance optimizations, such as improvement of asynchronous message progressing that currently introduces significant overhead and is available only in the multithreaded version of the library. The updated RMA interface in MPI-3 provides MPI users with a very flexible interface for one-sided operations. The potential benefits of this new interface are described at: http://htor.inf.ethz.ch/publications/img/mpi_mpi_hybrid_programming.pdf. Now, it's time for MPI implementations to be performance-efficient and competitive with various PGAS languages.
Conclusion

MPI-3 standard provides MPI users with a more flexible interface for writing performance-efficient programs. We have examined two major novelties—nonblocking collective operations and the new RMA interface with the help of the Intel MPI Library 5.0 Beta. We conclude that each of these new features may contribute to performance improvement of MPI applications. Though for existing MPI programs the use of new MPI-3 features may lead to application modification, in some cases, as we have demonstrated, these benefits far outweigh the efforts.

References

An OpenMP* Timeline

By Jim Cownie, OpenMP Architect, Alejandro Duran, Application Engineer, Michael Klemm, Senior Application Engineer, and Luke Lin, OpenMP Software Product Manager

1996 Vendors provide similar but different solutions for loop parallelism, causing portability and maintenance problems.

Kuck and Associates, Inc. (KAI) | SGI | Cray | IBM | High Performance Fortran (HPF) | Parallel Computing Forum (PCF)

In spring 7 vendors, Intel, and DOE agree on the spelling of parallel loop and form the OpenMP ARB. By October, version 1.0 of the OpenMP specification for Fortran is released.

The OpenMP ARB reaches 15 members of which 5 are supercomputing centers. This mixture of vendors and users is a trademark of OpenMP's cooperative style of operation.

OpenMP releases its first Technical Report that outlines how accelerator and coprocessor devices will be handled.

OpenMP gears toward version 4.1 and 5.0. Topics under discussion include more support for heterogeneous systems, improvements to the tasking model, support for transactional memory, data affinity, and interoperability with other programming models.


OpenMP ARB Membership Evolution

<table>
<thead>
<tr>
<th>Year</th>
<th>Permanent ARB</th>
<th>Auxiliary ARB Members</th>
<th>OpenMP Google Scholar Hits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1997</td>
<td>8</td>
<td>8</td>
<td>36</td>
</tr>
<tr>
<td>1998</td>
<td>11</td>
<td>11</td>
<td>110</td>
</tr>
<tr>
<td>1999</td>
<td>13</td>
<td>13</td>
<td>293</td>
</tr>
<tr>
<td>2000</td>
<td>15</td>
<td>15</td>
<td>477</td>
</tr>
<tr>
<td>2001</td>
<td>17</td>
<td>17</td>
<td>698</td>
</tr>
<tr>
<td>2002</td>
<td>19</td>
<td>19</td>
<td>1020</td>
</tr>
<tr>
<td>2003</td>
<td>22</td>
<td>22</td>
<td>1350</td>
</tr>
<tr>
<td>2004</td>
<td>26</td>
<td>26</td>
<td>1330</td>
</tr>
<tr>
<td>2005</td>
<td>25</td>
<td>25</td>
<td>1370</td>
</tr>
<tr>
<td>2006</td>
<td>24</td>
<td>24</td>
<td>1600</td>
</tr>
<tr>
<td>2007</td>
<td>23</td>
<td>23</td>
<td>1880</td>
</tr>
<tr>
<td>2008</td>
<td>22</td>
<td>22</td>
<td>2320</td>
</tr>
<tr>
<td>2009</td>
<td>21</td>
<td>21</td>
<td>3100</td>
</tr>
<tr>
<td>2010</td>
<td>20</td>
<td>20</td>
<td>4100</td>
</tr>
<tr>
<td>2011</td>
<td>19</td>
<td>19</td>
<td>5370</td>
</tr>
<tr>
<td>2012</td>
<td>18</td>
<td>18</td>
<td>6010</td>
</tr>
<tr>
<td>2013</td>
<td>17</td>
<td>17</td>
<td>6470</td>
</tr>
<tr>
<td>2014</td>
<td>16</td>
<td>16</td>
<td>7777</td>
</tr>
</tbody>
</table>

Open Computing Language (OpenCL™) is a royalty-free standard for cross-platform parallel programming. The OpenCL programming model supports heterogeneous platforms with various types of devices including general-purpose CPUs, graphics processors, and coprocessors. The OpenCL specification includes several low-level features that specifically address GPUs, such as local memory, barriers, images, samplers, and workgroups. Such features may not exist in other devices. Intel introduced OpenCL support for its general-purpose CPUs in 2010, and added support for the Intel® Xeon Phi™ coprocessor, a highly parallel x86 coprocessor, early in 2013.

OpenCL mainly attracts developers because of its promise of cross-device portability and its broad support by hardware vendors. OpenCL code designed for a specific target GPU device will usually run on x86 seamlessly, or with only minor changes. Nevertheless, performance portability
across devices is not guaranteed. Specifically, OpenCL code tuned for a GPU would typically perform sub-optimally on an Intel Xeon Phi coprocessor, and vice versa. This is normally caused by the use of GPU hardware-specific features such as local memory and barriers; the use of too few workgroups (which exposes insufficient coarse-grain parallelism); or workgroups that are too small (leading to excessive fine-grain parallelism).

A typical Intel Xeon Phi coprocessor includes 60 Intel® architecture cores, each capable of running up to four hardware threads. The OpenCL runtime schedules these 240 hardware threads at the granularity of whole workgroups. Therefore, any OpenCL task (NDRange) which includes fewer than 240 workgroups will underutilize the available execution resources. To achieve high resource utilization and efficient load balancing, one should aim for at least one thousand workgroups in each NDRange.

With OpenCL on the Intel Xeon Phi coprocessor, fine-grained NDRanges may cause some runtime scheduling overhead. This overhead becomes noticeable when workgroups take less than 200 microseconds each. Therefore, coarser-grained workgroups are preferred. One way to achieve this is to increase the workgroup duration by fusing multiple kernels into a single larger kernel. Another option is to configure the NDRange to include fewer but larger workgroups, while maintaining the minimal number of workgroups as described above.

Intel's OpenCL compiler implicitly vectorizes each workgroup routine based on dimension zero of the NDRange (even though it cannot know its value). In effect, the dimension zero loop is unrolled by the target vector size. For example, a generated 3-D kernel code might look like this:

```c
__kernel kernel_name(/* kernel arguments */)
{
    for (int dim2=0; dim2 < get_local_size(2); dim2++)
        for (int dim1=0; dim1 < get_local_size(1); dim1++)
            {
                for (int dim0=0; dim0 + VECTOR_SIZE < get_local_size(0);
                     dim0 += VECTOR_SIZE)
                    /* Vectorized_Kernel_Body */
                    for (; dim0 < get_local_size(0); dim0++)
                        /* Original_Kernel_Body, for remainder iterations */
            }
}
```

A kernel being vectorized across dimension zero of the workgroup.
On an Intel Xeon Phi coprocessor, the vectorization size is set to 16 regardless of the data types used. It is therefore good practice to make the local size at dimension zero a multiple of 16. Moreover, unit stride accesses to buffers along dimension zero should result in more efficient vector code.

Although existing OpenCL code that has been tuned for GPUs may perform sub-optimally on the Intel Xeon Phi coprocessor, with modest tuning effort the same OpenCL code can perform well on both Intel Xeon Phi and GPUs. One application that exhibits this behavior is the BUDE molecular docking code developed by Simon McIntosh-Smith and colleagues at the University of Bristol. BUDE’s OpenCL implementation sustains over 32% of peak floating point performance on a wide range of many-core architectures, including the Intel Xeon Phi coprocessor. BUDE’s approach to delivering performance portability with OpenCL is described in the recent paper *High Performance in silico Virtual Drug Screening on Many-Core Processors*.¹

Bristol joined the Intel Parallel Computing Center Program in February 2014, with a particular focus on exploiting OpenCL and OpenMP* to leverage Intel Xeon Phi.² Codes being optimized in Bristol for these platforms include the BUDE molecular docking code, the ROTORSIM CFD code,³ and the CloverLeaf mini-app from the Mantevo benchmark suite.⁴

**Summary**

OpenCL has evolved considerably since the release of its initial (1.0) specification at the end of 2008. The range of implementations currently available provides a de facto portable programming standard across disparate heterogeneous platforms, enabling customers to leverage their development investments. Both the specification and implementations are still actively advancing. Performance portability is one major challenge that is being pursued by many OpenCL teams in both research and industry. In this short article, we have outlined some of the performance portability considerations related specifically to the Intel Xeon Phi coprocessor.

References


Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
In a previous post I discussed the Intel® Transactional Synchronization Extensions (Intel® TSX) technology released in the new generation of processors. I described the Intel® Threading Building Blocks (Intel® TBB) implementation of the HLE interface (speculative_spin_mutex). Now we can talk about the implementation of speculative_spin_rtw_mutex, a Preview Feature of TBB 4.2 Update 2.

speculative_spin_rtw_mutex uses RTM for mutual exclusion, and allows both concurrent reads and concurrent writes. It also contains a spin_rw_mutex because it may be necessary to perform the operation without speculation.

If concurrent executions of code protected by the mutex do not conflict, all reads complete and writes are atomically committed without explicitly taking the lock.

If there is a conflict or another problem that prevents speculative execution, and the transaction is aborted, speculative_spin_rtw_mutex may retry the transaction or it may take the lock for real. If a writer takes the lock for real, all speculative readers and writers will abort the transaction and wait for the writer to complete, at which time the transactions may be retried. If a reader takes the lock for real all speculative writers will abort the transaction and wait for the reader to release the lock, at which time the transactions may be retried. Real readers and speculative readers may proceed in parallel. All this happens "under the covers," as part of the TBB implementation.