Electronic Design®

By TAMMY CARTER, M. S., Sr. Product Manager
Software & GPGPUs, Curtiss-Wright Defense Solutions,
www.curtisswrightds.com

Choosing a Parallelization Technique: What's the Best Path?

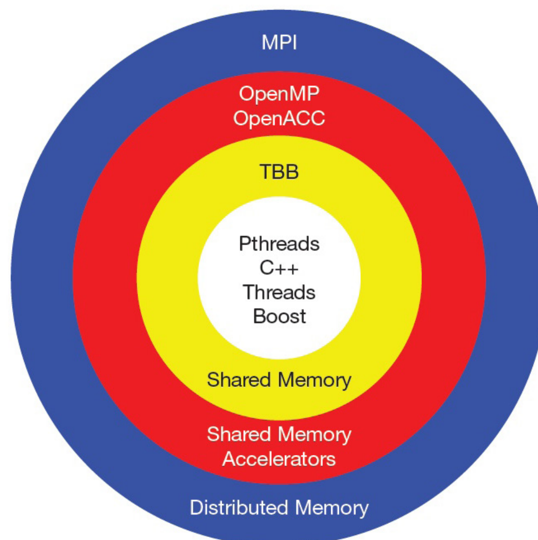
High-performance computing means the use of multiple cores and parallel processing, but there are many ways to address the software implementation.

Most of the legacy code that needs porting to newer systems is serial code, meaning that the code runs on a single processor with only one instruction executing at a time. Modern OpenVPX boards incorporate powerful, multicore processors such as the Intel Xeon-D. The inefficiency of running serial code on these high-performance processors increases the number of boards required in your system, negatively impacting your SWaP-C. General-purpose GPUs (GPGPUs) are also becoming more common in OpenVPX systems due to their massively parallel architecture that consists of thousands of cores designed to process multiple tasks simultaneously.

To modernize your serial code for parallel execution, you must first identify the individual sections that can be executed concurrently, and then optimize those sections for simultaneous execution on different cores and/or processors. Parallel programs must also employ some type of control mechanism for coordination, synchronization, and data realignment. To aid in parallelization, numerous open standard tools are available in the form of language extensions, compiler extensions, and libraries.

So, which method or path is right for you? Let's explore a few options.

Most of the options have evolved from pthreads, a "C" library defined by the IEEE as POSIX 1003.1 standard, which started circa 1995. POSIX is short for Portable Operating System Interface for UniX. Being a mature technology, pthreads is available on most CPU-based platforms. Remember, a thread is a procedure that runs independently from its main program. Scheduling procedures to run simultaneously and/or independently is a "multi-threaded" program.



Parallel-programming options.

WHY WOULD YOU WANT TO USE THREADS?

Creating threads requires less system overhead than creating processes, and managing threads requires fewer system resources than managing processes. Pthreads share the same memory space within a single process, eliminating the need for data transfers.

Consisting of approximately 100 functions, pthreads' calls are divided into four distinct groups: thread management, mutexes, conditional control, and synchronization. As the name implies, thread-management functions handle the creating, joining, and attaching of the threads. An abbreviation for "mutual exclusion," mutexes function as locks on a common resource such as a variable, or hardware device, to ensure that only one thread has access at a given time. The synchronization functions manage the read/write locks and the barriers. Think of a barrier as a stop sign; the threads must wait at this point

until all of the threads in their group arrive and then they can proceed to the next operation.

C++/BOOST THREADS

While pthreads is a C library, C++ 11 std:threads, as the name implies, is a C++ class library. C++ threads include features such as scope locks, recursive mutexes, object lifelines, and exceptions. While pthreads provides functions to cancel threads, signal handling of primitives, and control of stack size, C++ threads does not. In some cases, C++ threads are built on top of pthreads libraries, resulting in a slight performance penalty. By definition, C++ sports a higher abstraction level, a good interface, and easier integration with other C++ classes than pthreads.

Comprising more than eight individual libraries, Boost C++ supports a wide range of application domains including C++ threads. Members of the C++ standards committee created Boost and it serves as an extension to the STL (Standard Template Library). In addition to being open source and designed to be platform-neutral, Boost is well-documented and peer-reviewed. Since most of Boost libraries consist of templates, you can simply add the correct header file to get started.

Inherent in Boost and C++ threads is the RAII (Resource Acquisition Is Initialization), which binds the lifecycle of the resource to the lifetime of the object with automatic storage duration. Next, let's visit the compiler extension camp; home to OpenMP and OpenACC.

OpenMP

Supporting C/C++ and FORTRAN, OpenMP (Open Multi-Processing) provides a simple interface for shared-memory parallel programming on multiple platforms. A consortium of major computer hardware and software vendors created OpenMP in 1998. The OpenMP specification consists of APIs, pragmas, and settings for OpenMP-specific environment variables.

By definition, pragmas must be machine- or operating-system-specific. If the compiler doesn't understand the pragma, it will ignore it—this is an important feature. For example, you can place OpenMP directives in your serial code and still run the code in either serial mode or parallel mode depending on your compiler setting. In C/ C++ code, a pragma will look like "#pragma token-string." For instance, "#pragma omp parallel for" might be all that's needed to parallel a simple "for" loop.

OpenMP performs parallelization by using a master thread to fork a specified number of worker threads, and then divides the task between the threads. Allocated by the runtime environment, the threads will execute on different cores. Upon completion, the threads join back to the calling thread and then resumes sequential execution.

Since the resource management is hidden, you're probably wondering how OpenMP determines the number of threads to use in a block. By default, it will be the number of available execution pipelines; i.e., the number of cores in the processor, or double that if hyper-threading is enabled. Of course, there are methods to override the default. OpenMP supports both task parallelism (different tasks running on the same data) and data parallelism (the same task running in parallel on different data). By default, each OpenMP thread executes the parallelized section of the code independently.

In version 4.0 released in 2013, OpenMP added support for heterogeneous systems, including multiple attached NVIDIA GPUs. The host (a CPU) can create and destroy the data environment on the devices, as well as map the data to the devices. In addition to offloading the target code regions to the target devices, the host updates the data between the host and the device.

OpenACC

In 2011, a group of hardware and software companies created OpenACC (Open Accelerators) to fill the missing gap of accelerator support in OpenMP. The lofty goal back then was to support a host system with a wide variety of targets including GPUs, digital signal processors (DSPs), Xeon Phi, and cell processors.

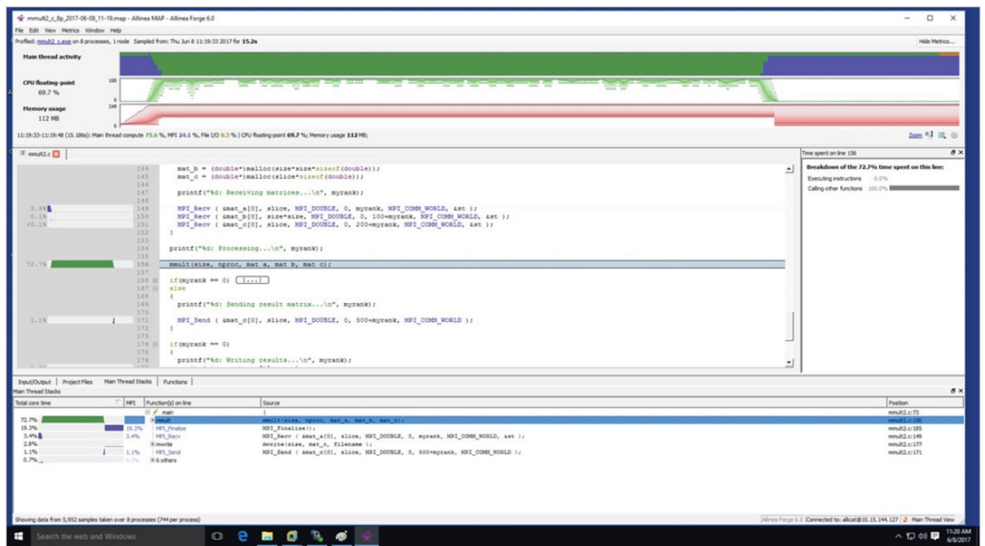
Like OpenMP, OpenACC uses data directives, clauses, and parallel directives that don't require modification of the underlying code. OpenACC also supports C/C++ and FORTRAN applications. Unlike OpenMP, OpenACC works with AMD GPUs as well as NVIDIA GPUs. Currently, more compilers support OpenMP than OpenACC. For example, the GNU and Intel compilers have supported OpenMP for many years, while initial support for OpenACC is just appearing in version 7 of the GNU compiler; it's currently not supported by the Intel compiler. The main compiler for OpenACC is PGI.

To achieve their goals of supporting a variety of accelerators, OpenACC took a descriptive approach, while the OpenMP design is prescriptive. OpenACC uses the directives to describe the properties of the parallel region to the compiler, which then choses the optimal values. In OpenMP, the user must explicitly specify the parallel execution strategy, and then map it to the underlying architecture. Since the user is in control, OpenMP will allow directives that will run very fast, but may not give the right answer! This makes a good debugger essential.

The addition of the "SIMD" (single instruction, multiple data) directive to OpenMP allows multiple iterations of the loop to be executed concurrently using vector instructions. Note that OpenMP uses "SIMD" in the pragma to refer to a type of implementation, while OpenACC uses "Vector" as a synonym for "SIMD" but without implying an implementation.

Another difference, OpenACC requires the parallel loop be data-race-free across loop iterations. OpenACC also uses

the concept of “gangs of workers” instead of the threads employed in OpenMP. Depending on the target, a simple parallel directive in OpenACC will run with multiple gangs, or a combination of gangs and vector lanes. Translating this to OpenMP would require multiple directives. There have been talks of merging OpenMP and OpenACC, and each new release of either standard brings them closer together in functionality.



Profiling an MPI program.

TBB LIBRARIES

Now, let’s switch gears and look at how the Intel TBB (Thread Building Blocks) library can help parallel your code.

Striving to avoid the issues with lower-level APIs like pthreads, Intel TBB (Thread Building Blocks) is a C++ template library designed to take advantage of multicore processors from Intel, Arm, and Power Architecture. It provides natural nested/recursive parallelism as well as supports data parallel programming for better scalability.

TBB’s approach eliminates the need to create, synchronize, or terminate threads manually. Treating operations as “tasks” to abstract, while accessing multiple processors, TBB will dynamically allocate these “tasks” to individual cores.

TBB consists of generic parallel algorithms, concurrent containers, low-level synchronization primitives, a scalable memory allocator, and a work-stealing task scheduler. The task scheduler will determine the task sizes, the number of resources and how they’re allocated to the tasks, and finally, schedule the resources accordingly. Since cache use is one of the most important factors, the scheduler will favor tasks that were most recently in that core because the memory will most likely still be holding that task’s data.

To decouple the programming from the underlying hardware, Intel executes the task based on graph dependencies. Intel provides the Flow Graph Analyzer to graphically construct the graphs and analyze your application. The flow graphs represent computational tasks as nodes, and inter-node communications as edges. Different types of nodes will execute user code, order and buffer messages, and split/join messages, as well as other functions.

So, what are the strengths and weaknesses of TBB? It gives the programmer direct control of the parallel algorithm, but hides the resource management, plus it doesn’t work on GPUs.

MPI

Message Passage Interface (MPI) is the highest-level framework with its own built-in multi-machine distributed infrastructure. It simplifies the development of portable and scalable parallel applications in C, C++, FORTRAN, Python, and R for both distributed-memory and shared-memory architectures.

MPI’s goal is to hide the underlying communications mechanism without sacrificing performance. In 1991, a small group of researchers from academia and industry created MPI. After two years of work, the MPI working group presented the first draft of the MPI standard at the 1993 Supercomputing Conference. Major vendors of supercomputers, universities, government laboratories, and industry embraced it.

If deciding to use MPI, the most important consideration will be that all parallelism is explicit. The user must identify the parallelism, and then implement the algorithm using MPI constructs.

In its simplest form, MPI employs point-to-point messages based on send/receive operations achieved by synchronous, asynchronous, or buffered communications. Collective communications are used to transmit data among all processes in a specified group, and the barrier function will synchronize the processes without passing data. Broadcast, scattering, gathering, and all-to-all transfers between the nodes are examples of data-movement options.

Another valuable function is “MPI_Reduce(),” which takes data from a group of nodes and performs a function such as sum, product, or a user-defined function, and then stores the results on one node. The latest version of MPI added advanced features such as non-blocking collective communications,

remote memory access (RMA), and procedures for creating virtual topologies, such as indexing the processors as a two-dimensional grid instead of the standard linear array.

COMPARISONS

With your head spinning from this whirlwind tour, you're probably thinking "which one should I choose?" That answer depends on your requirements, schedule, and a host of other considerations:

- Do you have an easily parallelizable algorithm?
- Do you have numerous arbitrary tasks that you would like to execute simultaneously?
- How much communication is required between the tasks?
- What scalability is a requirement for growth?

Pthreads and C++ threads are very low level and provide extremely fine-grained control of your thread management. Pthreads and C++ threads are also very flexible, but require a steeper learning curve and a greater programming effort to achieve the desired performance.

Code using pthreads and C++ threads is optimized for the current number of cores, resulting in additional effort to scale up to more cores, or move to different processors. In addition, neither translate to GPUs. I like to describe pthreads and C++ threads as the assembly language of the parallel world. Unlike pthreads and C++ threads, OpenMP allocates the number of threads based on the number of available cores, leading to a more scalable solution.

OpenMP and OpenACC have a unified code base for both serial and parallel application. Because the original serial code is unmodified, the probability of inadvertently introducing a bug is reduced. This single code base also helps when debugging complex code, where it's unknown if the code is broken because of the parallelization, or due to a bug in the original code. However, pragmas can make it more difficult to debug synchronization, race conditions, and even simple errors due to the lack of visibility into the pragma. A good OpenMP/OpenACC debugger can alleviate these and other debugging heartaches. By supporting work on one section of the code at a time, pragma-based tools enable incremental parallelism.

Often considered siblings, OpenMP and OpenACC share a long list of similarities, but with a few notable differences driven by philosophy. While OpenMP believes compilers are dumb and users are smart, the OpenACC mantra in compilers can be smart, and even smarter with the user's help. For example, OpenMP isn't dependency-aware and will attempt to parallel whatever the user requests, while OpenACC will refuse to parallel if the compiler detects an error.

TBB can be harder to plug into existing code and has a steeper learning curve than either OpenMP or OpenACC, but it could be argued that TBB gives you more control.

Finally, there's MPI, which some would point out is not directly a parallelization tool. MPI can solve a wider range of problems than either OpenMP or OpenACC, but it requires more program changes to port from serial code. Remember, MPI's performance relates directly to the communication mechanism between nodes. OpenMP and OpenACC tend to be better choices for multicore processors and GPUs, while MPI performance would be better on a distributed network.

HYBRID PROGRAMMING

You're probably now thinking "I like the features of this model, but I could also use the benefits of the other tool!" Never fear, the hybrid-programming model rides to the rescue. A common example is calling a lower-level library based on pthreads from a user application using TBB or one of the pragma-based frameworks. All of the programming models discussed will work in conjunction with MPI. Typically, MPI will handle the board-to-board communications, while one of the other paradigms tackles internal processor parallelism.

FINDING A SOLUTION

Once you've decided which parallel model is correct for your application, the next step is to have a versatile toolset that enhances your capability to produce quality software with less effort. Curtiss-Wright's OpenHPEC Accelerator Suites can help your software team build complex, efficient, multi-threaded, multiprocessor code fast. OpenHPEC includes several versions of MPI (OpenMPI and Mvapich2), and, both OpenHPEC and OpenHPEC LX (for smaller systems) include the Arm (formerly Allinea) debugger and profiler—DDT and MAP. The Arm tools are used on more than 75% of the world's supercomputers, and are taught at most major universities.

The Arm DDT is the Swiss Army knife of debuggers, and, along with the MAP profiler, you can debug and optimize single and multi-threaded C and C++ programs. Unlike most tool suites, it debugs and optimizes pthreads, OpenMP, OpenACC, and MPI, as well as the mixed hybrid-programming models. Using the MAP profiler is easy—there's no need to instrument your code and take a chance of introducing bugs, nor is there a need to remember arcane compilation settings.

Tammy Carter is the Senior Product Manager for GPGPUs and software products, featuring OpenHPEC, for Curtiss-Wright Defense Solutions. In addition to a M.S. in Computer Science, she has over 20 years of experience in designing, developing, and integrating real-time embedded systems in the defense, communications and medical arenas.