Tune OpenMP Applications

by Alexei Alexandrov, Senior Software Developer, Intel
High performance computing (HPC) has a long history and today is critical to business, research, and science. Clusters consisting of thousands of machines help enable many advances of modern science with both theoretical and practical implications, working 24/7 to enrich the lives of every person on earth.

HPC parallelism is exploited in three levels: process, thread, and SIMD vectorization (including auto-vectorization, for example in the Intel® compiler). For process-level parallelism, the Message Passing Interface (MPI) is the de facto standard in the HPC industry. For thread-level parallelism, the OpenMP® programming model is prevalent in the HPC community, and other models—such as Intel® Threading Building Blocks and Intel® Cilk™ Plus—are gaining traction. While Intel® VTune™ Amplifier XE supports all of these paradigms, this article will focus on useful techniques for profiling HPC programs that use OpenMP.

The examples shown are collected from an OpenMP application running on an Intel® Xeon Phi™ coprocessor, but the techniques used are valid for tuning all OpenMP programs. The examples use Intel VTune Amplifier's command line to collect data, as this is the most common way to gather data on systems with Intel Xeon Phi processors. But you don’t need to worry about learning all the switches. The graphic user interface (GUI) is commonly used to set up the analysis and generate a command line that you can cut and paste.

Profiling OpenMP programs with Intel VTune Amplifier XE

We’ll begin by walking through a simple scenario of collecting and analyzing performance data for an OpenMP program. This will include three steps:

1. Collect the data using a command-line interface.
2. Analyze the data using an interactive GUI, with examples of capabilities such as filtering and loop analysis.
3. Generate the hotspot profile using an Intel VTune Amplifier command line interface. This is often useful for automating performance regression testing or feeding the data into another program.

In step 1, to profile a program that uses OpenMP on an Intel® Xeon® processor-based computer, you can use any of the supported analysis types, just launching the data collection as usual:

```
$ amplxe-cl -collect hotspots -- ~/sp.A.x
```

or using hardware event-based sampling analysis types to profile a program executed on an Intel Xeon Phi coprocessor:

```
$ amplxe-cl -c knc-lightweight-hotspots -search-dir all:p=/lib/firmware/mic -- ssh mic0 ~/sp.A.x
```

The search directory is specified here to make sure the Intel Xeon Phi runtime binaries can be found on the host during result post-processing.

In step 2, having collected the result and opened it in the Intel VTune Amplifier XE GUI, you can begin your analysis. For an OpenMP program, it is a good idea to start looking into data by grouping the data by threads in the Bottom-up view. To do this, select a thread-based grouping in the Grouping combo-box above the grid (Figure 1). Selecting several rows in the grid allows you to easily see the number of selected threads and the summary statistics for them—this is useful for understanding how a given team of threads behaved. By right-clicking the selected items, you can also filter in or filter out the selected data. Combined usage of grouping and filtering allows you to dice and slice the data as needed. The filter feature is particularly useful for filtering out the time spent in the OpenMP runtime library to see the pure contribution of the user code.
on overall performance. For example, Figure 1 shows that the user module only contributed 14.5 percent of CPU cycles; the rest of the cycles were spent in other modules, mostly spinning because of non-optimal CPU affinity and the number of OpenMP threads running.

In the case above, using the thread grouping revealed that only 64 software threads were effectively executing the user code—although there were 244 hardware threads available since this is an Intel Xeon Phi coprocessor with 61 cores. From the program source code, it became clear that the available parallelism is limited by one dimension of the input problem size, so that the program needs to be changed to adopt the higher available parallelism. Or, at the very least, the affinity and number of OpenMP threads to use should be set to match the workload properties. Setting KMP_AFFINITY to "balanced" and OMP_NUM_THREADS to 64 indeed provided better execution time and better balance between the threads. Since the Intel® OpenMP Library actively uses spinning instead of waiting, the CPU time spent outside of the user module (and usually in the OpenMP module) is a common indication of imbalance or excessive serial execution. You can also choose grouping by Core to understand the program balance in terms of hardware cores. This is often useful to understand the performance of Simultaneous Multithreading (SMT) parallelism (Figure 2).

Having identified and fixed high-level balance and overhead issues, you can narrow down to function-level analysis (Figure 3). When using the Intel compiler together with the Intel OpenMP implementation, the OpenMP region bodies are conveniently aggregated into pseudo functions with names like compute_rhs_omp$parallel@17, so that it’s easier to distinguish the time spent inside and outside of the region body. For instance, in this case the name reads as “the OpenMP Parallel Region at line 17 in function compute_rhs.”

Knowing a hot function, you can dive into its C, C++, or Fortran source and assembly to identify which source lines were taking most of the time and which assembly code was generated by the compiler for them. Understanding the latter often provides guidance for how you should direct the compiler to vectorize the inner loops. Starting with version 3.0, Intel VTune Amplifier XE also helps you understand the structure of the program in terms of loops (useful for loopy HPC codes). To enable that mode, switch the “Loop Mode” to “Loops and functions” in the GUI filter bar (or use "-loop-mode=loop-and-function" in the amplxe-cl command line interface). The top-down view will show the looping structure of the program (Figure 4). In this example, we can easily see that the OpenMP region function has a loop at line 295, which nests to loop at line 296, and then to loop at line 297—where the latter loop being peeled by the compiler as part of the vectorization process. This is identified by observing two loop instances belonging to the same source line. Note that the first of the peeled instances takes the larger fraction of the time, since the second instance is a remainder loop with a small iteration count.

“Applications targeting highly parallel architectures must be able to leverage extensive hardware resources. Once your application is optimized for Intel® Xeon® processors, you can maximize performance on Intel® Xeon Phi™ coprocessors by following a straightforward code optimization methodology.”
Figure 3: Function-level hotspot view.

Figure 4: Using loop analysis to understand the looping structure of the program.
The timeline view can be used to narrow down the scope of the data to a specific time region. For example, to skip the program startup "cold" phase or to magnify the data into the execution of specific region. Using the ITT API, supported by Intel VTune Amplifier, to mark up the program execution may be useful for the latter. When viewing results that ran many threads, use the "Tiny" mode of timeline bands to easily fit more data onto your display:

![Figure 5: Using tiny mode for timeline bands to view many-thread data over time.](image)

Finally, in step 3 of our simplified optimization flow, it may be useful to automate the hotspot extraction using the command line interface (e.g., for automatic performance regression testing). Most of the capabilities mentioned above can be used from the command line. For example, here you can see how to output the ten top hotspots grouped by thread and function, filtered by a specific module and time region, and with loop analysis on:

```
$ amplxe-cl.exe -R hotspots -loop-mode=loop-and-function -filter module=sp.A.x -limit=10 -group-by thread,function -r <result directory path>
```

<table>
<thead>
<tr>
<th>Thread</th>
<th>Function</th>
<th>Module</th>
<th>CPU Time:Self</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread (0x2cd3)</td>
<td>[Loop@0x4136a1 at line 297 in compute_rhs_Somp$parallel@17]</td>
<td>sp.A.x</td>
<td>0.394</td>
</tr>
<tr>
<td>Thread (0x2cd5)</td>
<td>[Loop@0x4136a1 at line 297 in compute_rhs_Somp$parallel@17]</td>
<td>sp.A.x</td>
<td>0.385</td>
</tr>
<tr>
<td>Thread (0x2cb5)</td>
<td>[Loop@0x4136a1 at line 297 in compute_rhs_Somp$parallel@17]</td>
<td>sp.A.x</td>
<td>0.376</td>
</tr>
<tr>
<td>Thread (0x2cd7)</td>
<td>[Loop@0x4136a1 at line 297 in compute_rhs_Somp$parallel@17]</td>
<td>sp.A.x</td>
<td>0.376</td>
</tr>
<tr>
<td>Thread (0x2cc1)</td>
<td>[Loop@0x4136a1 at line 297 in compute_rhs_Somp$parallel@17]</td>
<td>sp.A.x</td>
<td>0.358</td>
</tr>
<tr>
<td>Thread (0x2ccf)</td>
<td>[Loop@0x4200d8 at line 294 in x_solve_Somp$parallel_for@27]</td>
<td>sp.A.x</td>
<td>0.358</td>
</tr>
<tr>
<td>Thread (0x2ca2)</td>
<td>[Loop@0x4136a1 at line 297 in compute_rhs_Somp$parallel@17]</td>
<td>sp.A.x</td>
<td>0.349</td>
</tr>
<tr>
<td>Thread (0x2ccf)</td>
<td>[Loop@0x4200d8 at line 294 in x_solve_Somp$parallel_for@27]</td>
<td>sp.A.x</td>
<td>0.339</td>
</tr>
<tr>
<td>Thread (0x2cd2)</td>
<td>[Loop@0x4200d8 at line 294 in x_solve_Somp$parallel_for@27]</td>
<td>sp.A.x</td>
<td>0.339</td>
</tr>
<tr>
<td>Thread (0x2cab)</td>
<td>[Loop@0x4136a1 at line 297 in compute_rhs_Somp$parallel@17]</td>
<td>sp.A.x</td>
<td>0.330</td>
</tr>
</tbody>
</table>

**Figure 6**

**Conclusion**

The Intel VTune Amplifier XE techniques described here are useful for diving into performance analysis of an HPC program, but there is still more to learn. The product documentation and online resources (see the Intel Knowledge Base) can provide further information on the features of the product, including MPI program analysis, loop and inline function analysis, ITT API usage, performance analysis automation using command-line reporting, and many more. The techniques were illustrated using an Intel Xeon Phi coprocessor, but apply equally well to an Intel® Xeon® system. One nice benefit is that tuning to improve the parallelism in your application usually yields performance benefits when running on both Intel Xeon processors and Intel Xeon Phi coprocessors: a double win! 

[ ]
Expand Your Debugging Options

by Nicolas Blanc, Software Engineer, Intel
and Georg Zitzlsberger, Technical Consulting Engineer, Intel

Overview
Execution of typical applications for the Intel® Xeon Phi™ coprocessor is distributed among
the host and one or more coprocessors. In general, there are two basic domains for Intel Xeon Phi coprocessor-based applications:

- Heterogeneous applications that execute seamlessly across the host and selected
coprocessors using either the explicit or implicit offload model
- Applications solely executed on each of the available coprocessors, using the so-called
  native model

Developing and debugging applications in both native and offload models requires
dedicated support. Intel® Parallel Studio XE 2013 for Linux® provides a range of solutions,
along with ease of use and full awareness of the Intel Xeon Phi architecture. Vendors, such as
Allinea or Rogue Wave, enrich the Intel Xeon Phi environment with further debugging
options. Thus, developers for Intel Xeon Phi can select among many debugging options,
depending on their field of application. Next, we’ll look at a selection of Intel debugging
solutions, grouped by the two typical domains.

Offload Model
There are two different offload models: explicit and implicit. Both provide a
simple, yet flexible programming
environment to develop applications
running on the Intel Xeon Phi copro-
cessors and host systems. An offload
application comes as a single executable
file containing both host and target
code. Detection of available coprocessors,
including data transfers, is automatically
handled at runtime. Therefore, work
packages are scheduled transparently
among any coprocessors and the host.
In case no coprocessor is available, the
execution remains entirely native on
the host system.

The programming flexibility makes
debugging with standard tools more
complex. Therefore, Intel Parallel Studio
XE 2013 for Linux® provides an Eclipse*
debugger plugin with full awareness
of both offload models. This allows
instant debugging without further
configuration. Using Eclipse as an inte-
grated development environment (IDE)
provides an easy-to-use, well-known
graphical interface for this debugger.
The integration also offers scalability
up to hundreds of threads on multiple
coprocessors for C, C++, and Fortran—a
necessity for Intel Xeon Phi because of
its manycore architecture.
Native Model

For the native model—where applications run exclusively on the coprocessor—different debug solutions are provided by the Intel Parallel Studio XE 2013 suite for Linux. Here, development takes place on one system while the created applications are running on a remote coprocessor. Such a coprocessor may not necessarily be installed on the same development system, but can be installed on another host, reachable via network.

Debug support for this use case requires remote capabilities, which are also available via the aforementioned Eclipse plugin. A second solution is the Intel® Debugger (IDB) for command-line debugging. A typical IDB debug session starts with the following simple steps:

```sh
$ idbc_mic -tco -rconnect=tcpip:<coprocessor>
(idb) idb file-remote <path_to_executable_on_coprocessor>
(idb) file <path_to_executable_on_dev_system>
...
```

The debugger is started via `idbc_mic` by providing the name or IP address of the coprocessor. When the debugger is started, the path of the executable to be run on the coprocessor is specified first, followed by the path of the same executable on the development system that launched the debugger. Afterwards, IDB can be used as usual.

Alternatively, IDB can also attach to an application already running on the coprocessor via its process ID `<pid>`.

```sh
$ idbc_mic -tco -rconnect=tcpip:<coprocessor>
(idb) attach <pid> <path_to_executable_on_dev_system>
...
```

The advantages of using IDB are its speed, compatibility with C, C++, and Fortran, and GNU GDB* syntax for a flat learning curve. With its focus on the command line, it can easily be used for automated script-based testing as well. More information on how to use IDB can be found in the Intel® Debugger User’s and Reference Guide.

A third solution for the native mode is using GNU GDB for Intel Xeon Phi. Like IDB, it provides remote debugging capabilities; in addition, it can be hosted directly on the coprocessor. Developers preferring GDB on the host can continue using it for debugging on the coprocessor. Intel added support to GDB for the Intel® Many Integrated Core (Intel® MIC) architecture of Intel Xeon Phi. This version is not part of Intel Parallel Studio XE 2013 for Linux, but can be downloaded from our Intel® Many Integrated Core Architecture Forum.

Summary

The wide variety of debugging tools supports the different, versatile use cases of the Intel Xeon Phi coprocessor. Developers can choose between comfortable GUI based or fast, low overhead command line debuggers. There are a number of different vendors offering such solutions, including Intel.

Learn More

Intel® Parallel Studio XE 2013

Intel® Xeon Phi™ coprocessor

and Intel® Many Integrated Core Architecture (Intel® MIC)

Intel® Debugger User’s and Reference Guide

Intel® Many Integrated Core Architecture resources

(including GNU GDB sources)

Allinea releases tools for Intel® Xeon Phi™ coprocessor developers

Rogue Wave announces support for the Intel® Xeon Phi™ coprocessor in key products
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