OpenMP* Region Analysis with Intel® VTune™ Amplifier XE

By Kirill Rogozhin, Technical Consulting Engineer, Software and Services Group, Developer Products Division, Intel and Dmitry Prohorov, Intel® VTune™ Amplifier XE Engineering Manager

OpenMP* is a commonly used parallel programming model, especially in high performance computing (HPC). OpenMP constructs are intended to boost performance by distributing work among multiple threads. However, adding OpenMP directives into the code doesn't always result in the desired performance improvements—a programmer often needs to tune it. The tuning method is closely tied with performance analysis—you need to know the reason behind the inefficiency to fix the bottleneck.

The majority of well-known performance tools show performance-related information associated with functions/loops in a bottom-up or top-down style, rather than showing particular OpenMP parallel regions. As a result, a programmer loses the parallel region/parallel loop context. Without that context, understanding inefficiencies like imbalance or overhead becomes difficult.
Intel® VTune™ Amplifier XE 2015 can produce profiling results in terms of OpenMP constructs that the programmer operates with. It can show the parallel and serial time of an application, the difference between measured and ideal execution of a parallel region, statistics broken down by parallel loops, and per-region CPU utilization histograms. Users can more easily understand where to invest their tuning efforts by knowing how much gain in application wall-clock time they can theoretically achieve by fixing each problem. Additionally, overhead and spin time classification helps users understand the reasons of the inefficiency (e.g., waiting on a barrier due to load imbalance or spinning on a lock due to synchronization).

This article will describe what types of OpenMP inefficiencies the Intel VTune Amplifier can detect and how users could treat and address them.

**BLOG HIGHLIGHTS**

Improving MPI Communication between the Intel® Xeon® Host and Intel® Xeon Phi™

BY LOC-NGUYEN »

MPI Symmetric Mode is widely used in systems equipped with Intel® Xeon Phi™ coprocessors. In a system where one or more coprocessors are installed on an Intel® Xeon host, Transmission Control Protocol (TCP) is used for MPI messages sent between the host and coprocessors or between coprocessors on that same host. For some critical applications this MPI communication may not be fast enough.

In this blog, I show how we can improve the MPI Intra-Node communication (between the Intel Xeon host and Intel Xeon Phi Coprocessor) by installing the OFED stack in order to use the Direct Access Programming Library (DAPL) as a fabric instead. Even when the host does not have an InfiniBand* Host Channel Adapter (HCA) installed, the DAPL fabric can still be used to transfer MPI messages via scif0, to a virtual InfiniBand* interface.

On an Intel® Xeon® E5-2670 system running the Linux* kernel version 2.6.32-279 and equipped with two Intel® Xeon Phi™ C0 stepping 7120 coprocessors (named mic0 and mic1), I installed MPSS* 3.3.2 and Intel® MPI Library 5.0 on the host. Included in the Intel MPI Library is the benchmark tool IMB-MPI1. For illustration purposes, I ran the Intel MPI Benchmark Sendrecv before and after installing the OFED stack obtained results for comparison. In this test used with two processes, each process sends a message and receives a message from the other process. The tool reports the bidirectional bandwidth.

More
Simple Configuration

Good news: You almost don't need any special configuration except the most recent versions of the Intel® Compiler and VTune Amplifier XE. To get all the features described in this article, use the following versions:

- Intel® VTune™ Amplifier XE 2015 Update 2
- Intel® Parallel Studio XE 2015 Composer Edition Update 2

To profile an OpenMP application, just run any typical analysis, such as Advanced Hotspots. The only additional setting you need is to set the following environment variable before running performance collection. For example (for Linux*):

```
export KMP_FORKJOIN.FRAMES_MODE=3
```

The reason is that some instrumentation of Intel® OpenMP runtime (related to per-barrier statistics) was still in the experimental state when this article was written. So you need to enable the functionality explicitly using the environment variable.

Please note that the experimental feature is a beta-quality feature that may or may not appear in a future production release. Data collected with the experimental feature enabled is not guaranteed to be backward-compatible with future releases.

Exploring Inefficiencies: Serial Code and CPU Utilization

We recommend you first look at the CPU utilization histogram on the Summary pane of your application results. It displays the elapsed time of your application, broken down by CPU utilization levels (Figure 1). It shows only useful utilization so the CPU cycles that were spent by the application burning CPU in spin loops (active wait) are not counted on the histogram.
Ideally in parallel applications, most of the elapsed time should be concentrated in the “green” area, so the majority of the CPU cores are utilized. **Figure 1** shows the result of our test running on an Intel® Xeon Phi™ coprocessor. It is clear that the majority of available computing resources are underutilized. There may be two main reasons:

- Big serial portion: Most of the code is running without parallelism at all.
- Poor efficiency of parallel regions: The code is parallel, but some bottlenecks limit scalability.

Now look at the “OpenMP Analysis” section on the same Summary pane (**Figure 2**).
This section contains the elapsed time of the application split into the elapsed time of the serial portion (outside of any parallel region) and the parallel part of the program. If the serial portion is significant, that would be a good place to start looking. Find ways to minimize serial execution by either introducing more parallelism or by doing algorithm or microarchitecture tuning for sections that seem unavoidably serial. For high thread-count machines, serial sections have a severe and negative impact on potential scaling (following the infamous Amdahl’s Law) and should be minimized as much as possible. In our test, 93.4 percent of wall clock time is spent in a serial region, so this is definitely the bottleneck.

Serially executed code can be explored going to the Bottom-up tab, choosing the “/OpenMP Region/Thread/Function..” grouping and filtering by the Master Thread of the “[Serial - outside any region]” row (Figure 3). Here in the Bottom-up grid, we see that a serial region takes 21.571s, which is the majority of the elapsed time. But if we sort by CPU time, the first line is occupied by an OpenMP parallel region at line 179. It takes more than 164 seconds of CPU time compared to ~101 seconds in the serial region. Also, when we look at the serial region, it is essential to filter by Master Thread only. There are multiple OpenMP worker threads waiting or spinning on barriers when a serial region executes. We should exclude this time if we want to explore pure serial code. In our test, it takes only 20.713s of CPU time, which is several times less than the top-most parallel region. But on the highly parallel Intel Xeon Phi coprocessor, this kills performance; Amdahl’s Law is strict here.

General efficiency of parallel regions can be checked with the “OpenMP Region CPU Usage Histogram” on the Summary pane. This is essentially the same as the usual CPU usage histogram, but the CPU usage is reported not globally but rather for the selected parallel region. Figure 4 shows the CPU usage histogram from the same Intel Xeon Phi coprocessor result as Figure 1, but only for the parallel region at line 153. Here we see a different picture—sometimes the CPU concurrency is close to the ideal of 224, but more time is still spent with 32-40 concurrent hardware threads.
For more information regarding performance and optimization choices in Intel® software products, visit software.intel.com/en-us/articles/optimization-notice.

The Parallel Universe

**Exploring Performance Opportunities: Potential Gain**

Let’s take a look at another test case. We have analyzed NAS Parallel Benchmarks (NPB) to explore possible problems in OpenMP parallel region execution.

**Test setup:**

- **CPU:** Intel® Xeon® processor E5-2697 v2 @ 2.70GHz, 24 cores/48 threads.
- **OS:** RHEL* 7.0 x64
- **Compiler:** Intel® Parallel Studio XE 2015 Composer Edition Update 2
- **Workload:** NPB 3.3.1, “CG - Conjugate Gradient, irregular memory access and communication” module, class B

The number of OpenMP threads is configured to 24 to match the physical core count. **Figure 5** shows quite good CPU utilization for the benchmark run—the opposite of what we’ve seen in the previous example with a long serial region on an Intel Xeon Phi coprocessor. This is still not
ideal—significant time is distributed between 2 and 6 concurrently running cores. This means some parallel code is executed there, but it doesn't always load all 24 available cores. Serial time for NPB is negligible, so this is not a problem here (Figure 6). But look at the “Potential Gain” metric, highlighted in pink.

Potential gain estimates the difference in elapsed time between the actual measurement and an idealized execution of parallel regions, assuming perfectly balanced threads and zero overhead of the OpenMP runtime on work scheduling. So it is essentially the potential benefit of tuning, showing the maximum time that you might save by improving parallel execution. The potential gain metric can be more important than CPU or elapsed time, because it doesn't focus you on the top time-consuming region; it focuses you on the region where you most likely get maximum results from tuning.

In our test, Potential Gain shows that optimization of all parallel regions to an ideal state would save us 3.975s, or 34.9 percent of total application runtime—a feasible opportunity for optimization.

Those were metrics of the whole application. Now, let's go deeper, to the parallel region level. The five OpenMP top parallel regions are listed on the Summary tab, sorted by potential gain. In our test, the parallel region at line 514 is the source of almost all the application's potential gain—3.958s (Figure 6). This is good for us—we have narrowed down the problem to a single parallel region.
Finding the Reason Behind Inefficiency

Once we’ve focused on a particular parallel region, click the link on the region name to navigate to a Bottom-up view grouped by OpenMP Region and the region of interest selected (Figure 7).

The Bottom-up grid view has different statistics about the parallel regions—elapsed time with potential gain, the number of OpenMP worker threads employed for the given region, and the number of instances of the region (how many times it was called; e.g., from an outer loop or calling function). CPU time is broken down into effective time (user code execution), spin time, and overhead time. We can see that spinning time is significant: 92.159s. Before investigating where it comes from, let’s take a quick look at the source code (Fortran)—see Figure 8. The parallel region at lines 514:695 contains multiple parallel loops designated by ‘!$omp do’ constructs. This is bad news for us—metrics for the whole parallel region don’t tell which parallel loops are problematic.
Fortunately, VTune Amplifier XE can break down information not only by parallel region, but by OpenMP barrier as well. All 

```
#pragma omp for
```

or

```
!$omp do
```

constructs have synchronization barriers, unless you include a “nowait” clause. Since VTune Amplifier can distinguish those barriers, we should be able to see CPU and wall clock time for each of the parallel loops inside the same parallel region. You might need to create a custom grouping such as “/OpenMP Region/ OpenMP Barrier Type/ OpenMP Barrier/..” to see the per-barrier data.

After grouping by OpenMP barriers, things become clearer. First, most of the potential gain, elapsed, and CPU time come from the parallel loop at line 572 that is highlighted in Figure 9.

![Per-barrier performance metrics](image)

Second, expanding the Spin Time column breaks it into categories, revealing that all our spinning is due to imbalance. Third, an “OpenMP Loop Schedule Type” column appears saying that the given loop is statically scheduled.

### Fixing Imbalance

Source code of the loop at line 572 is shown in Figure 10. There is no “Schedule” OpenMP clause so scheduling is static by default. The parallel loop suffers from imbalance, so it would be natural to try dynamic scheduling in the hopes that the OpenMP runtime will automatically redistribute the workload. Changed code is shown in Figure 11.

![OpenMP parallel loop at line 572, original version](image)

![Parallel loop at line 572 with dynamic scheduling](image)
Analyzing and Fixing Scheduling Overhead

After changing scheduling to dynamic, the parallel loop performance has become even worse. Elapsed time increased from the original 10.445s to 11.102s (Figure 12). However, details in the grid show a different picture now—the imbalance disappeared, so it was really fixed. But another column is highlighted in pink—74.99s of CPU time goes to scheduling overhead. It means that the OpenMP runtime library performs some internal processing too heavily.

![Scheduling overhead for loop at line 572](image)

Look at the column “OpenMP Loop Chunk.” It has changed from the original 3125 to 1. This means that each iteration is scheduled individually—work items for worker threads are too small, and the OpenMP library has to schedule them too often. This parallelism is too fine-grained.

Grain size 1 is the default for dynamic scheduling, the minimal value. To prove our theory about fine granularity, we changed the grain size to 20 (Figure 13).

![Parallel loop at line 572 with dynamic scheduling and grain size=20](image)
Now see the performance results in Figure 14. Both imbalance and scheduling overhead take only about 1s. The barrier of the loop at line 572 has gone down in the hotspot list, because its potential gain has dropped to 0.077s from the original 3.133s. Elapsed time of the given loop is 8.928s versus the original 10.445s. Overall parallel region CPU time has decreased from the original ~250s to ~213s. So we have won some portion of the potential gain, though only about half of the potential 3 seconds.
A More Effective Way to Analyze Performance

Performance analysis of OpenMP applications is more natural with VTune Amplifier XE. You can investigate performance efficiency and bottlenecks in a “top-down” way: Start from general CPU utilization analysis and check what code is purely serial. You can see CPU utilization of the whole application or a particular parallel region. The potential gain metric helps you focus on the most interesting parallel regions in terms of potential benefit. Per-barrier data breakdown allows you to easily work with multiple parallel loops in the same parallel region.

The Bottom-up grid provides clear OpenMP statistics: scheduling type, chunk size, number of OpenMP threads, and parallel region instances. Performance metrics, such as detailed categorization of spin and overhead time, are intended to guide programmers in root-cause performance problems. This helps you understand what limits performance: imbalanced load, non-optimal granularity, waiting on locks, or something else.

These all are “synergy” features of the Intel OpenMP runtime library and VTune Amplifier XE. Both are available for free download as part of the Intel® Parallel Studio XE 2015 Professional Edition. You can download VTune Amplifier XE individually and additional online materials at the product site. Check out what performance issues the tools can find in your OpenMP application.

Try Intel® VTune™ Amplifier XE

Available in these software tools:
- Intel® Parallel Studio XE 2015 Professional and Cluster Editions >
- Intel® System Studio 2015 >
- Intel® Media Server Studio Professional Edition >