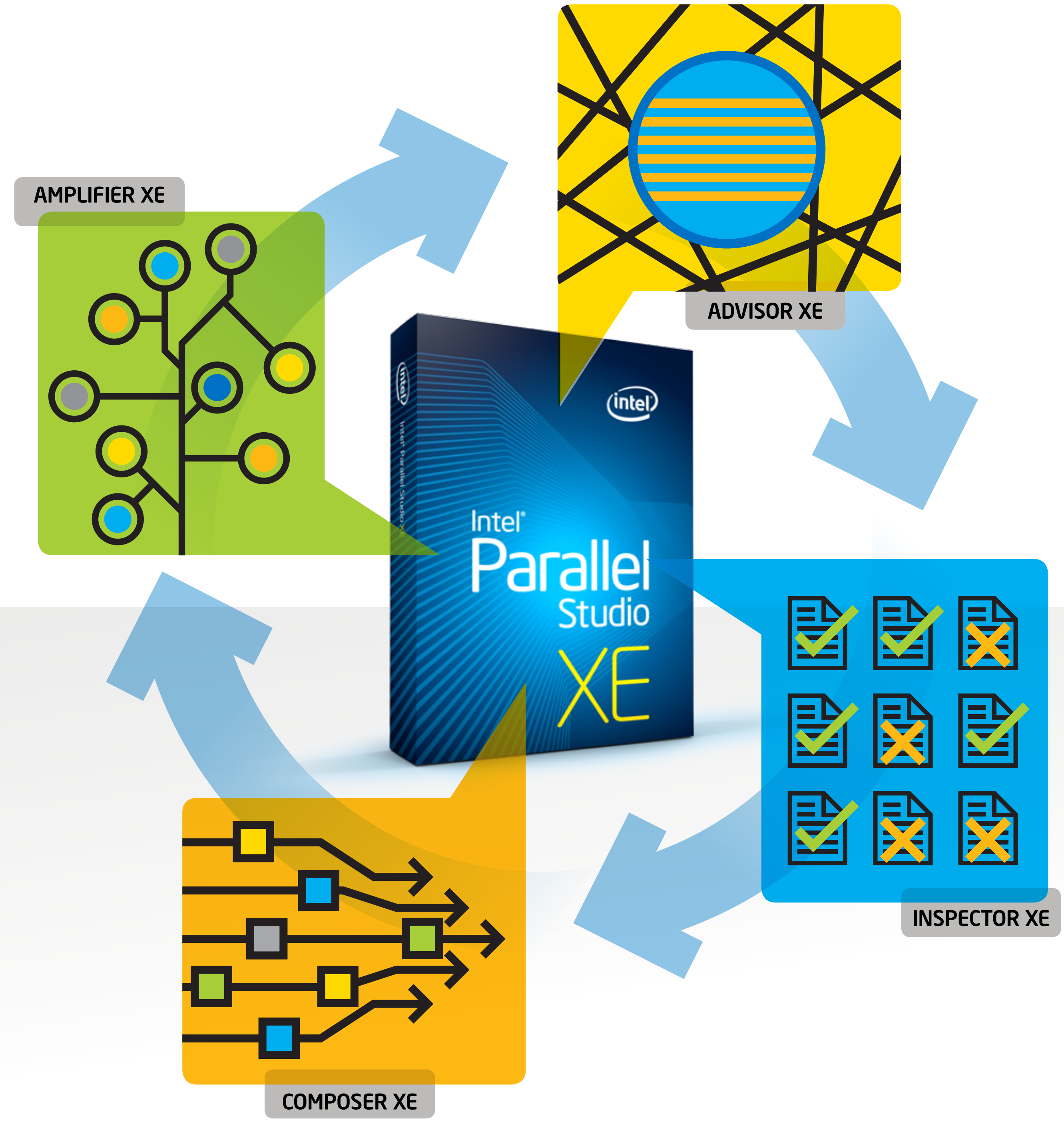
The Parallel Universe



Issue 15 | August 2013

CONTENTS

© 2013, Intel Corporation. All rights reserved. Intel, the Intel logo, Intel Core, Intel Inside, Cilk Plus, Pentium, VTune, VPro, Xeon and Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries. *Other names and brands may be claimed as the property of others.

LETTER FROM THE EDITOR

James Reinders, Director of Parallel Programming Evangelism at Intel Corporation.
James is a coauthor of two new books from Morgan Kaufmann, Intel® Xeon Phi™ Coprocessor High Performance Programming (2013), and Structured Parallel Programming (2012). His other books include Intel® Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism (O'Reilly Media, 2007, available in English, Japanese, Chinese, and Korean), and VTune™ Performance Analyzer Essentials (Intel Press, 2005).

Results Matter

It seems that we constantly find ourselves needing to push our software to deliver more than the status quo. The software that we, as innovators, create is critical in enabling everything from enterprise datacenters to mobile devices to run faster, create new experiences, connect users to services, and secure personal and business data. In this issue, we explore some of the new tools and capabilities that are helping software innovators to raise the bar.

Efficient Software Development: What's New in Intel® Parallel Studio XE 2013 Service Pack 1 takes a close look at the enhancements in the latest release of Intel Parallel Studio. The new features cover a wide spectrum—from predicting scalability to accelerating discovery of memory leaks to analyzing both CPUs and GPUs. The common denominator is an increase in programming efficiency and the quality of the end product.

Accelerated development can also be a result of faster and more accurate debugging. *Coprocessor Debugging Support in Intel® Parallel Studio XE* walks through Linux* and Windows* debug scenarios for both native applications and offload programs running completely or partially on the Intel® Xeon Phi™ coprocessor.

In *Full Scale Ahead: The Weather Research and Forecast (WRF) Model and Intel® Cluster Studio XE 2013,* scalability is a critical factor in next-generation, mesoscale, predictive analysis. A collaborative team of engineers and researchers tested some of the most advanced, supercomputing simulations against challenges such as Hurricane Katrina.

Whether we are striving to meet ever-increasing performance expectations, connect the Internet of Things, or bring our vision to life, the tools are available to support our development processes, increase insight, and help our applications get quantitative and qualitative results.

James Reinders
September 2013

Efficient Software Development: What's New in Intel® Parallel Studio XE 2013 Service Pack 1

by Kirill Rogozhin, *Technical Consulting Engineer, Developer Products Division Technical Computing, Analyzers and Runtimes, Intel*

Intel® Parallel Studio XE is a well-known suite for professional software developers, which helps them to create highly optimized and stable applications running on multicore systems and to scale these applications to future architectures. The latest update of this suite, called **Intel® Parallel Studio XE 2013 Service Pack 1** (SP1), contains a number of new features and optimization worth highlighting. For example, Intel® Composer XE now supports key features of the OpenMP* 4.0 (**http://openmp.org**) standard. The threading prototyping tool, Intel® Advisor XE, adds the ability to predict scalability over a range of core counts, enabling software architects to estimate scalability for Intel® Xeon Phi™ systems. Memory and thread debugger, Intel® Inspector XE, improves the transition from other verification tools by easily importing Rational Purify* and Valgrind* suppression files. With the latest version of Intel Inspector XE, you can find memory leaks even before the

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

analysis completes and the application exits. Performance profiler Intel® VTune™ Amplifier XE details overhead time and has added to the command line interface (CLI) to support drill down to the source. Exploring the function call tree is now easier with the Caller/Callee view. In addition to the CPU, now you can analyze the GPU (Intel® Graphics Technology) as well. And developers will get full support of the newest hardware, such as 4th generation Intel® Core™ processors, using the enhanced General Exploration profile within Intel VTune Amplifier XE.

Here, we will focus on the most interesting new features available in the Parallel Studio XE 2013 SP1 release for each suite component: Intel Composer XE, Intel Advisor XE, Intel Inspector XE, and Intel VTune Amplifier XE. (For information on the basic functionality of Intel Parallel Studio XE 2013, **visit: http://software.intel.com/en-us/intel-parallel-studio-xe/**.)

Intel Composer XE

Intel Composer XE combines C++ and Fortran compilers, performance libraries, parallel programming models, and a debugger extension (Linux*). Intel Composer XE 2013 SP1 has many improvements. We'll focus on vectorization enhancements and compiling for coprocessors and accelerators, introduced with **OpenMP 4.0** key features support.

OpenMP SIMD Constructs

Using single instruction multiple data (SIMD) instructions—also called vector or packed instructions—is one of the efficient ways to implement data parallelism. The Intel® compiler does its best to vectorize the code automatically; however, sometimes the programming style makes it difficult for the compiler. Therefore, user-hinted or SIMD vectorization is implemented in the Intel Compiler using "#pragma omp simd." With OpenMP 4.0, you can use these SIMD capabilities in standardized codes that are portable for both C, C++, and Fortran. When the "omp simd" construct is declared prior to a "for loop", it forces the compiler to generate vector code for that loop so that multiple iterations will be executed concurrently by means of SIMD instructions. You can use additional OpenMP clauses such as "reduction." **(Figure 1)**

Here, we will focus on the most interesting new features available in the Intel® Parallel Studio XE 2013 SP1 release for each suite component: Intel® Composer XE, Intel® Advisor XE, Intel® Inspector XE, and Intel® VTune™ Amplifier XE.

1

```
double pi()
{
    double pi = 0.0;
    double t;
#pragma omp simd private(t) reduction(+:pi)
    for (i=0; i<count; i++) {
        t = (double)((i+0.5)/count);
        pi += 4.0/(1.0+t*t);
    }
    pi /= count
    return pi;
}
```

2

```
#pragma omp declare simd notinbranch
float min(float a, float b) {
    return a < b ? a : b;
}

#pragma omp declare simd notinbrach
float distsq(float x, float y) {
    return (x - y) * (x - y);
}
```

3

```
#pragma omp parallel for simd
    for (i=0; i<N; i++)
        d[i] = min(distsq(a[i], b[i]), c[i]);
```

It's possible to define SIMD ("elemental") functions, where the developer specifies operations to perform on each element. Based on this, the compiler generates vector code for the function and knows that the function is available for usage from a SIMD loop. The "omp declare" construct is used to define a SIMD function. **(Figure 2)**

This allows you to combine SIMD-level parallelism (inside one core) and higher-level thread parallelism (across several cores). The following loop is initially vectorized, and then the resulting iterations are distributed among the threads. **(Figure 3)**

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

4

```
#pragma omp target map(to(b:count)) map(to(c,d)) map(from(a:count))
{
  #pragma omp parallel for
    for (i=0; i<count; i++)
      a[i] = b[i] * c + d;
}
```

With these tools, you can more efficiently program for coprocessors, multiply your efforts in powerful parallel frameworks, and find and fix complex performance issues on the latest hardware.

Utilizing the Coprocessor

Accelerators and coprocessors, such as the Intel Xeon Phi coprocessor, are becoming more and more popular. With OpenMP 4.0, you can offload computations (ideally, highly parallel computations) using the "omp target" construct. It runs the subsequent basic block of code on an accelerator/coprocessor. If a target device does not exist or is not supported by the implementation, the target region is executed by the host device (CPU). You can control the data environment with the "map" clause, as in the example (see **Figure 4**).

Intel Advisor XE

Intel Advisor XE enables modeling of parallel execution of serial code, so it can be used to prototype your parallel implementation before you actually introduce parallel instructions to the code. It gives software architects the flexibility to experiment with different threading designs before investing much effort in implementation. The tool can also predict data races that may appear after parallelizing the code, so you can resolve them as early as possible.

Extended Suitability Analysis

The scalability estimate initially was limited to 32 cores. This was sufficient for CPU threading, but not for the higher number of cores available on the Intel Xeon Phi coprocessor. Intel Advisor XE can now predict scalability for a custom number of cores. **Figure 5** shows an estimation for 512 cores, and even more are possible. The developer can quickly estimate the theoretical peak of his algorithm: is the algorithm ready to scale to make it beneficial to run on Intel Xeon Phi coprocessors?

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

Experiment Snapshot

Users of Intel Inspector XE and Intel VTune Amplifier XE have the ability to conduct several tests and to keep all historical results, comparing performance changes and tracking problem states. Earlier versions of Intel Advisor XE used to keep only the most recent result. This limitation came from the complex nature of the Intel Advisor XE profile, which was intricately tied with the current code version.

The latest release in Parallel Studio XE 2013 SP1 allows you to create a snapshot of an Intel Advisor XE experiment. This copy is read-only and allows the developer to refer to previous estimations. You can evolve your project, change algorithms, etc., and can always compare performance and scalability estimations of older approaches to the current version (see **Figure 6**).

Pause/Resume Annotations

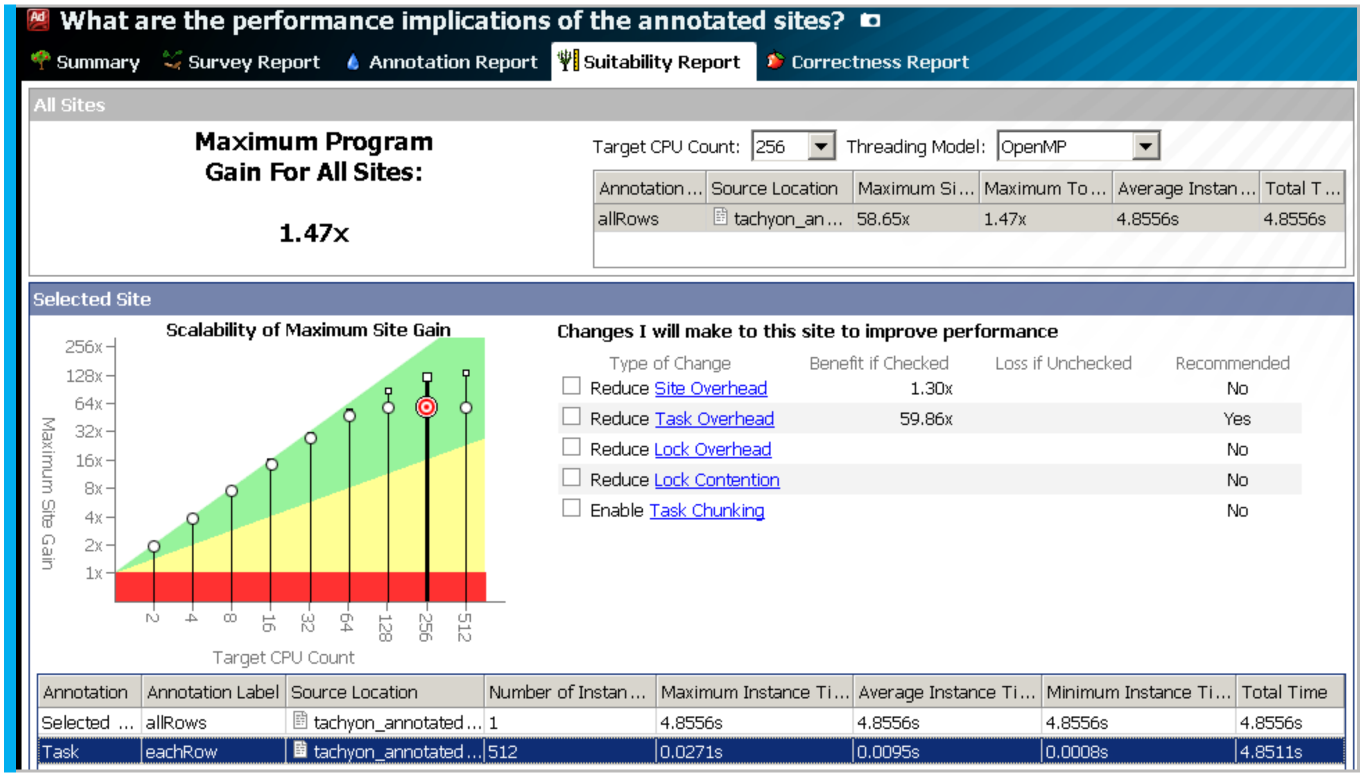
Intel Advisor XE analysis can have significant overhead, as it performs complex modeling of parallel execution and attempts to detect threading errors in the model. To reduce collection time on a large application, a user can bypass analysis of particular regions of code so the tool skips them, saving time. Analyzing only particular code regions may also improve the results precision—reducing the distortion from churning large amounts of unrelated data. The developer implements this using new annotation types: `ANNOTATE_DISABLE_COLLECTION_PUSH` and `ANNOTATE_DISABLE_COLLECTION_POP` (graphical-user-interface (GUI) buttons are also available) (see **Figure 7**).

Intel Inspector XE

Intel Inspector XE is a memory and thread debugger that performs dynamic analysis in runtime and integrates with standard debuggers. It combines a memory debugger that can find memory leaks, uninitialized and invalid memory accesses, and more, with a thread debugger that can detect deadlocks, data races, and other issues with thread communication. These kinds of problems may be missed by traditional regression testing and static analysis.

Import Valgrind and Rational Purify Suppression Files

Intel Inspector XE has the ability to suppress selected problems discovered during its analysis that are not relevant. Similar tools from other vendors have this ability as well. There may be many reasons for suppressing a problem: false positive reports, problems in third-party modules that cannot be fixed and should be eliminated from Inspector reports, etc. Suppression rules for large software projects are stored in separate files that contain definitions of the suppressed errors: code locations, module, problem type, and other information.



Maximum Program Gain For All Sites:

1.47x

Target CPU Count: 256 Threading Model: OpenMP

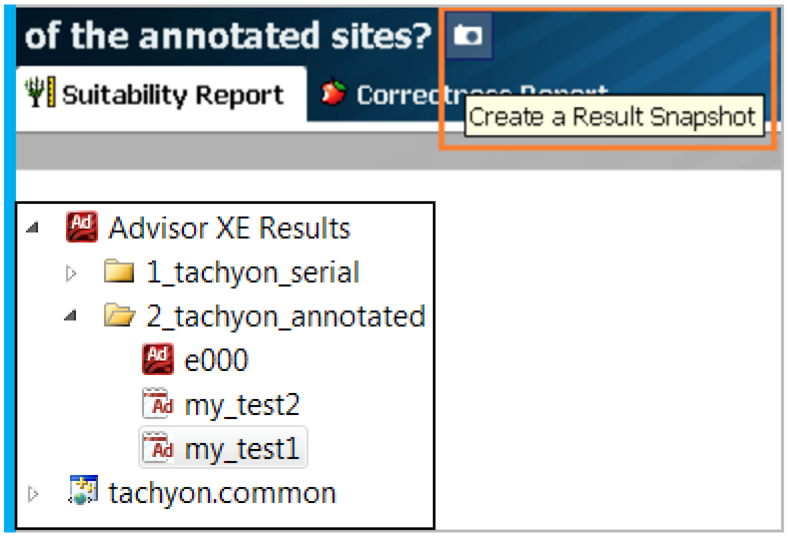
Annotation ...	Source Location	Maximum Si...	Maximum To...	Average Instan...	Total T...
allRows	tachyon_an...	58.65x	1.47x	4.8556s	4.8556s

Changes I will make to this site to improve performance

Type of Change	Benefit if Checked	Loss if Unchecked	Recommended
☐ Reduce Site Overhead	1.30x		No
☐ Reduce Task Overhead	59.86x		Yes
☐ Reduce Lock Overhead			No
☐ Reduce Lock Contention			No
☐ Enable Task Chunking			No

Annotation	Annotation Label	Source Location	Number of Instan...	Maximum Instance Ti...	Average Instance Ti...	Minimum Instance Ti...	Total Time
Selected ...	allRows	tachyon_annotated...	1	4.8556s	4.8556s	4.8556s	4.8556s
Task	eachRow	tachyon_annotated...	512	0.0271s	0.0095s	0.0008s	4.8511s

5 Scalability estimate for up to 512 cores



6 Experiment snapshot in Intel® Advisor XE

The updated version of Intel Inspector XE is able to import suppression files from other tools, such as Valgrind and Rational Purify. These files are converted into the Intel Inspector XE suppression file format. This feature simplifies the transition to Intel Inspector XE for those who have used other correctness-checking tools and might have such files in their projects. Inheritance of suppression rules saves time and preserves previous investment in investigating problems that have been suppressed in the past.

The updated version of Intel Inspector XE saves suppression files in a new text format that enables editing suppression rules manually in any text editor.

Detecting Memory Leaks before a Program Exits

Memory leak detection is one of the most common functions of memory analysis in Intel Inspector XE. In previous versions this analysis required a full program run, from start to exit, for the tool to be able to catch all heap allocations and deallocations that occurred in the program. If the target application runs too long, or if it never stops (like a daemon), or it crashes before exit, it was impossible for Intel Inspector XE to analyze memory leaks. The latest version of Intel Inspector XE solves these issues by introducing the "on-demand" leak report feature. Now, you can define a region to look for leaking memory. You can do it from the GUI controls, from the command line, or by calling APIs from the source code. Therefore, you can now check a particular

BLOG HIGHLIGHTS

AVX-512 Instructions

BY JAMES REINDERS »

The latest Intel® Architecture Instruction Set Extensions Programming Reference includes the definition of Intel® Advanced Vector Extensions 512 (Intel® AVX-512) instructions. These instructions represent a significant leap to 512-bit SIMD support. Programs can pack eight double-precision or 16 single-precision floating-point numbers, or eight 64-bit integers, or 16 32-bit integers within the 512-bit vectors. This enables processing of twice the number of data elements that AVX/AVX2 can process with a single instruction and four times that of SSE.

Intel AVX-512 instructions are important because they offer higher performance for the most demanding computational tasks. Intel AVX-512 instructions offer the highest degree of compiler support by including an unprecedented level of richness in the design of the instructions. Intel AVX-512 features include 32 vector registers, each 512 bits wide, eight dedicated mask registers, 512-bit operations on packed floating-point data or packed integer data, embedded rounding controls (override global settings), embedded broadcast, embedded floating-point fault suppression, embedded memory fault suppression, new operations, additional gather/scatter support, high-speed math instructions, compact representation of large displacement value, and the ability to have optional capabilities beyond the foundational capabilities. It is interesting to note that the 32 ZMM registers represent 2K of register space!

More >

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

code segment for leaks. The errors are reported during runtime, so there is no need to wait until the application finishes.

Intel VTune Amplifier XE

Intel VTune Amplifier XE is a performance profiler. It shows the most CPU-consuming parts of your application with selectable granularity ranging over threads, modules, functions, instructions, etc. It provides details about the work balance between the threads, waiting time, and synchronization primitives causing the waits. It allows an inquisitive developer to dig into microarchitectural performance bottlenecks, such as cache misses, false sharing, and many others.

Detailed Reporting of Overhead Time

In a multithreaded program, some CPU time is inevitably spent for thread synchronization, managing computational tasks, and so on. The time that is not spent for "mainline" calculations is considered overhead and it's essential to minimize overhead to achieve good performance. Intel VTune Amplifier XE now provides detailed information about overhead, and spin time (busy waiting). You can see what part of the CPU time was spent in spin waiting or overhead in particular functions, modules, instructions, etc. The tool is able to detect overhead coming from OpenMP, Intel® Threading Building Blocks (Intel® TBB) or Intel® Cilk™ Plus parallel frameworks. Developers can either check time values in the grid for objects of interest, or refer to the timeline to see how spinning and overhead time are distributed across threads (see **Figure 8**).

7

```
int main(int argc, char* argv[])
{
    ANNOTATE_DISABLE_COLLECTION_PUSH
    // Do initialization work here
    ANNOTATE_DISABLE_COLLECTION_POP

    // Do interesting work here

    ANNOTATE_DISABLE_COLLECTION_PUSH
    // Do finalization work here
    ANNOTATE_DISABLE_COLLECTION_POP

    return 0;
}
```

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

Improved Analysis of OpenMP Applications

The updated version of Intel VTune Amplifier XE provides an extended ability to analyze OpenMP parallel regions. Grouping by frame domain in the bottom-up pane now shows OpenMP parallel regions as frames (see **Figure 9**). By means of filtering, you can narrow down a performance profile to a particular OpenMP parallel region, see how much time was spent there, what the performance issues were, and the work balance. "[No frame domain - Outside any frame]" node represents the serial time spent outside any parallel region, so that you can clearly see the serial/parallel balance of your application (you may recall Amdahl's law).

Overhead and spin time related to OpenMP execution is now recognized not only for Intel OpenMP, but also for GCC*, and Microsoft OpenMP* runtimes. If you refer to **Figure 9**: "[OpenMP worker]" and "[OpenMP for]" nodes correspond to the Microsoft OpenMP library, module vcomp100.dll.

View Source and Assembly in Command Line Tool

There are cases when using the command line interface is more convenient than using the GUI. For instance, say you are profiling on a remote Linux server connected via SSH. In addition to existing reporting capabilities, you can now drill down to the source code or assembly right from the command line tool. You may not even need to retrieve results from the remote machine or configure VNC–explore profiling data in the same shell you used for doing the collection (see **Figure 10**).

Explore Call Tree with Caller/Callee View

The new Caller/Callee view combines the best of bottom-up and top-down views. It shows you self and total time for each function in the left pane and, for the selected function, its parents in the top-right pane and its children in the bottom-right pane (see **Figure 11**). You can see call sequences top to bottom and understand the contribution of each function in CPU time. You can filter by any function that is done on a total time basis, so you get all sub-trees that include the selected function at any level. This kind of representation will allow you to identify the most time-critical call path in your code.

GPU Profiling

Intel VTune Amplifier XE can now profile code for Intel® Processor Graphics.You can monitor overall GPU activity. Is the GPU used for video decoding? What CPU threads are triggering GPU calculations? Are all GPU resources utilized? This capability is especially interesting for computational tasks running on the GPU via OpenCL kernels. Intel VTune Amplifier XE shows kernels in the grid, so you can see work sizes and hardware issues, such as GPU L3 cache misses (see **Figure 6**). On the timeline, OpenCL kernels are marked on the CPU threads

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

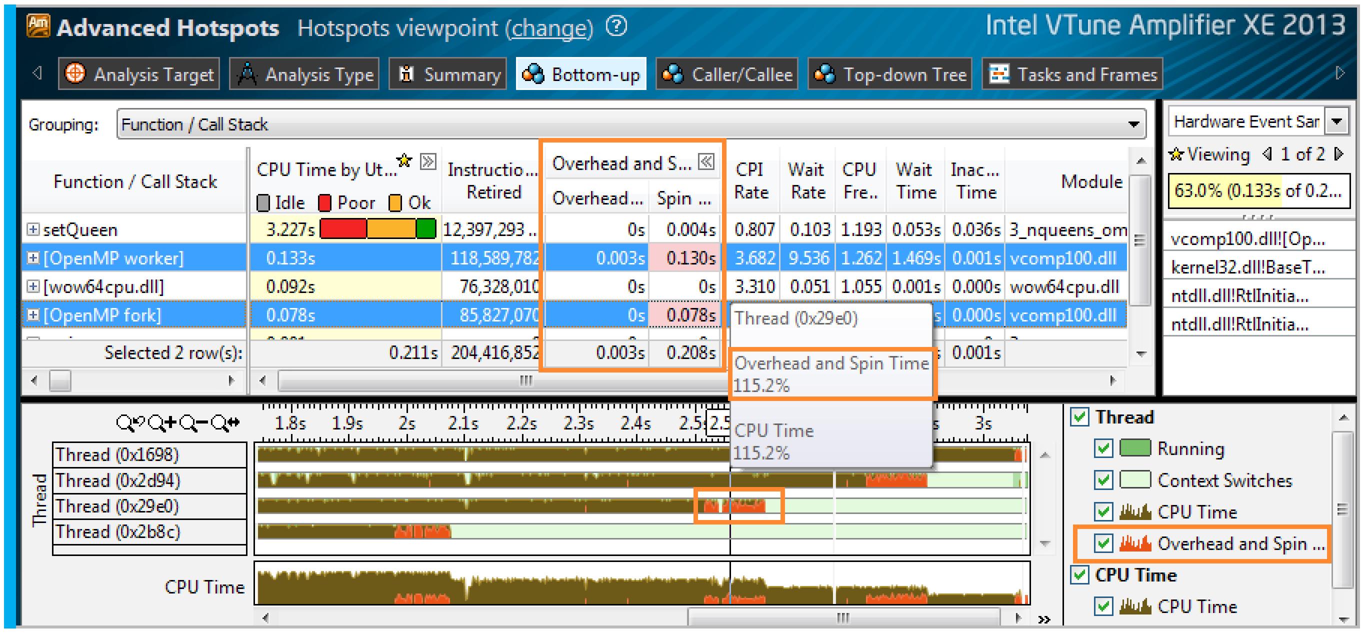
that invoked them. You can also check out what percentage of the GPU Execution Units is active, idle, or stalled. With the GPU profile, you can understand if your workload is CPU- or GPU-bounded, which are the hottest OpenCL kernels running on the GPU, and whether there is room for higher utilization of GPU resources.

Top-Down Performance Analysis for 4th Generation Intel Core Processors

Exploring microarchitectural issues is a complex task. It requires a deep understanding of CPU architecture and knowledge of hundreds of hardware events. To make deep performance analysis more structured and user-friendly, a top-down organization of performance data was implemented in the General Exploration profile (see **Figure 12**). Performance metrics are predefined and possible issues are highlighted. The hierarchical data display allows you to control the level of detail you need and provides a structured breakdown of high-level problem categories into detailed groups and individual issues, making navigation and understanding easier.

Summary

Intel Parallel Studio XE 2013 SP1 unveils new capabilities for software developers. With these tools you can more efficiently program for coprocessors, multiply your efforts in powerful parallel frameworks, and find and fix complex performance issues on the latest hardware—and thus produce high-end software products. Download the recent suite version and check out how it can improve the project you are currently working on. ●



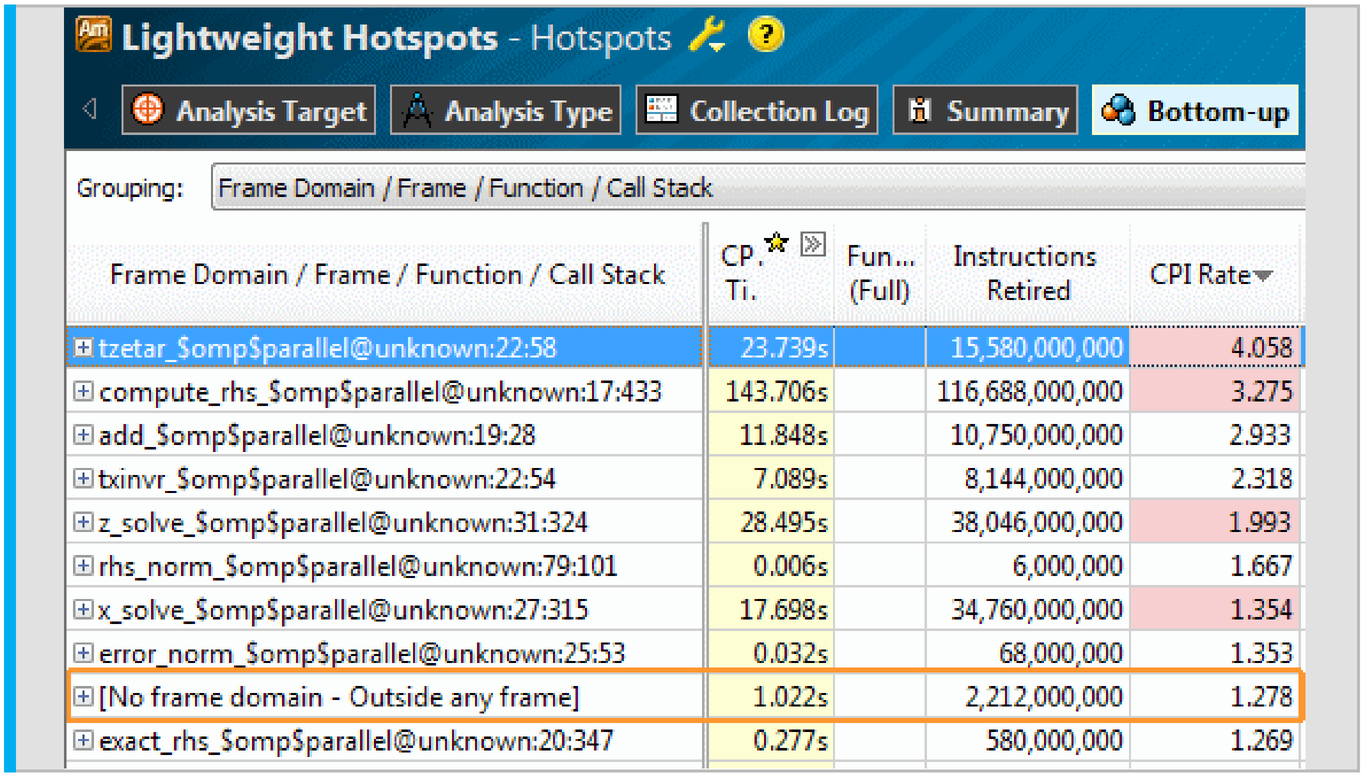
Grouping: Function / Call Stack

Function / Call Stack	CPU Time by Ut... (Idle / Poor / Ok)	Instructio... Retired	Overhead and S... Overhead...	Overhead and S... Spin ...	CPI Rate	Wait Rate	CPU Fre..	Wait Time	Inac... Time	Module
setQueen	3.227s	12,397,293 ..	0s	0.004s	0.807	0.103	1.193	0.053s	0.036s	3_nqueens_om
[OpenMP worker]	0.133s	118,589,782	0.003s	0.130s	3.682	9.536	1.262	1.469s	0.001s	vcomp100.dll
[wow64cpu.dll]	0.092s	76,328,010	0s	0s	3.310	0.051	1.055	0.001s	0.000s	wow64cpu.dll
[OpenMP fork]	0.078s	85,827,070	0s	0.078s					0.000s	vcomp100.dll
Selected 2 row(s):	0.211s	204,416,852	0.003s	0.208s					0.001s	

vcomp100.dll![Op...
kernel32.dll!BaseT...
ntdll.dll!RtlInitia...
ntdll.dll!RtlInitia...

Thread (0x1698)
Thread (0x2d94)
Thread (0x29e0)
Thread (0x2b8c)

8 Spin and overhead time in Intel® VTune™ Amplifier XE

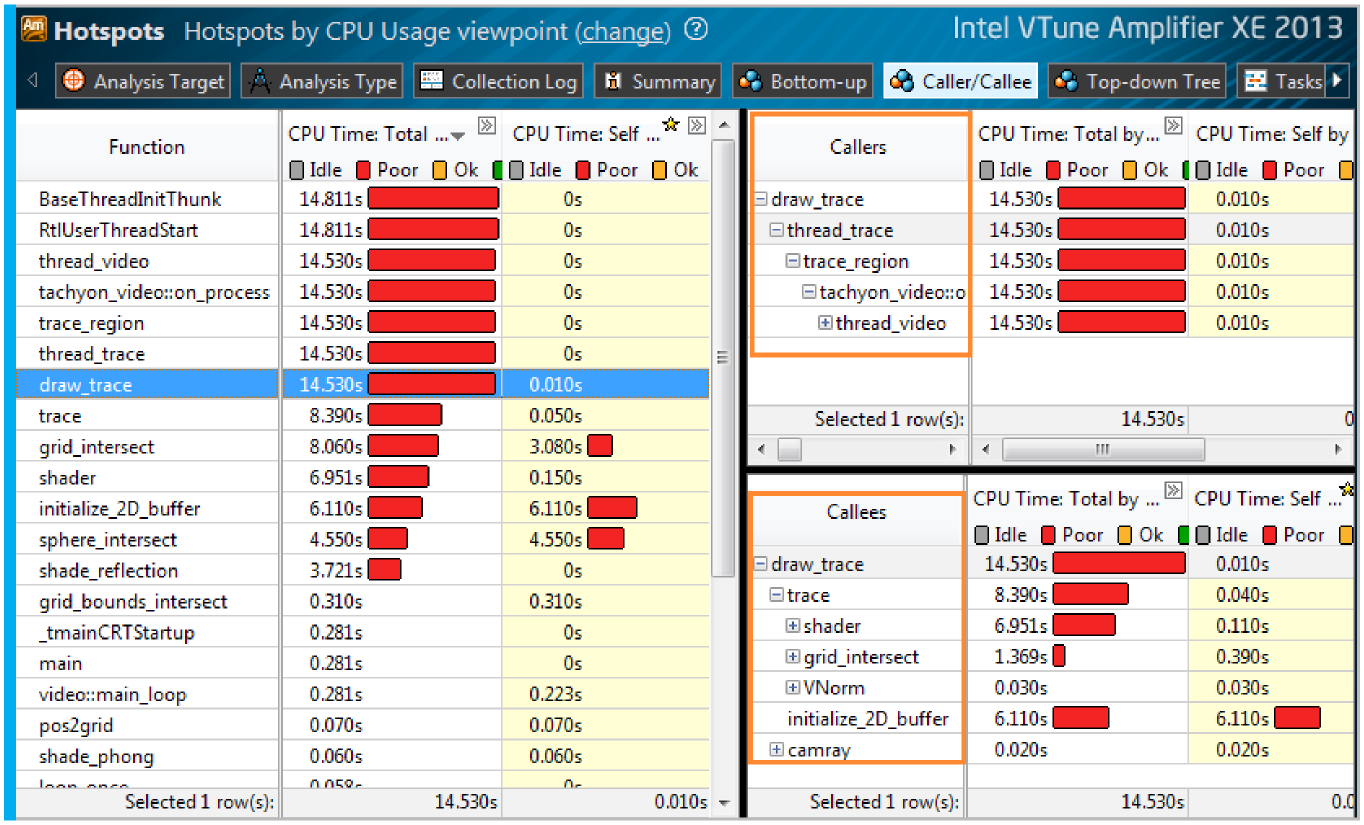

Lightweight Hotspots - Hotspots
Analysis Target
Analysis Type
Collection Log
Summary
Bottom-up
Grouping: Frame Domain / Frame / Function / Call Stack

Frame Domain / Frame / Function / Call Stack	CP. Ti.	Fun... (Full)	Instructions Retired	CPI Rate
tzetar_ompparallel@unknown:22:58	23.739s		15,580,000,000	4.058
compute_rhs_ompparallel@unknown:17:433	143.706s		116,688,000,000	3.275
add_ompparallel@unknown:19:28	11.848s		10,750,000,000	2.933
txinvr_ompparallel@unknown:22:54	7.089s		8,144,000,000	2.318
z_solve_ompparallel@unknown:31:324	28.495s		38,046,000,000	1.993
rhs_norm_ompparallel@unknown:79:101	0.006s		6,000,000	1.667
x_solve_ompparallel@unknown:27:315	17.698s		34,760,000,000	1.354
error_norm_ompparallel@unknown:25:53	0.032s		68,000,000	1.353
[No frame domain - Outside any frame]	1.022s		2,212,000,000	1.278
exact_rhs_ompparallel@unknown:20:347	0.277s		580,000,000	1.269

9 Analyzing OpenMP* parallel regions

10

```
localhost:/tmp/tachyon # amplxe-cl -report hotspots -source-object function=grid_intersect -r r000hs/
Source Line  Source                                                                   CPU Time:Self
-----------  -----------------------------------------------------------------------  -------------
460            return 1;
461          }
462
463
464          /* the real thing */
465          static void grid_intersect(grid * g, ray * ry)                                    0.036
466          {
467
468
469            flt tnear, tfar, offset;
470            vector curpos, tmax, tdelta, pdeltaX, pdeltaY, pdeltaZ, nXp, nYp, nZp;
471            gridindex curvox, step, out;
472            int voxindex;
473            objectlist * cur;
474
475            if (ry->flags & RT_RAY_FINISHED)
476              return;
477
478            if (!grid_bounds_intersect(g, ry, &tnear, &tfar))
479              return;
480
481            if (ry->maxdist < tnear)                                                        0.020
```



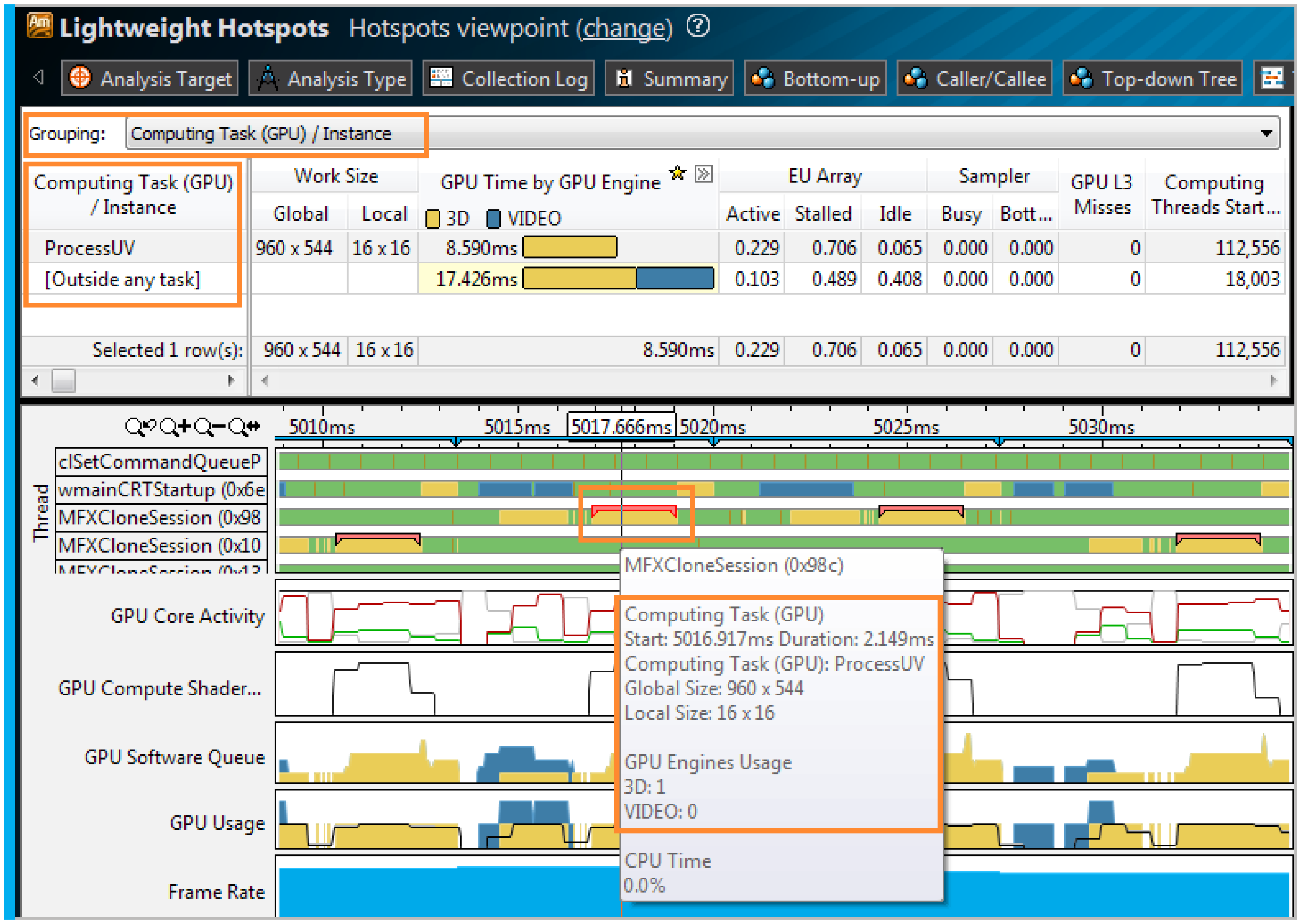
Function	CPU Time: Total ...	CPU Time: Self ...
BaseThreadInitThunk	14.811s	0s
RtlUserThreadStart	14.811s	0s
thread_video	14.530s	0s
tachyon_video::on_process	14.530s	0s
trace_region	14.530s	0s
thread_trace	14.530s	0s
draw_trace	14.530s	0.010s
trace	8.390s	0.050s
grid_intersect	8.060s	3.080s
shader	6.951s	0.150s
initialize_2D_buffer	6.110s	6.110s
sphere_intersect	4.550s	4.550s
shade_reflection	3.721s	0s
grid_bounds_intersect	0.310s	0.310s
_tmainCRTStartup	0.281s	0s
main	0.281s	0s
video::main_loop	0.281s	0.223s
pos2grid	0.070s	0.070s
shade_phong	0.060s	0.060s
Selected 1 row(s):	14.530s	0.010s

Callers	CPU Time: Total by...	CPU Time: Self by
⊟draw_trace	14.530s	0.010s
⊟thread_trace	14.530s	0.010s
⊟trace_region	14.530s	0.010s
⊟tachyon_video::o	14.530s	0.010s
⊞thread_video	14.530s	0.010s
Selected 1 row(s):	14.530s	0

Callees	CPU Time: Total by ...	CPU Time: Self ...
⊟draw_trace	14.530s	0.010s
⊟trace	8.390s	0.040s
⊞shader	6.951s	0.110s
⊞grid_intersect	1.369s	0.390s
⊞VNorm	0.030s	0.030s
initialize_2D_buffer	6.110s	6.110s
⊞camray	0.020s	0.020s
Selected 1 row(s):	14.530s	0.0

11 Caller/Callee view in Intel® VTune™ Amplifier XE

Download evaluation software at:
http://software.intel.com/en-us/intel-software-evaluation-center.



Grouping: Computing Task (GPU) / Instance

Computing Task (GPU) / Instance	Work Size		GPU Time by GPU Engine	EU Array			Sampler		GPU L3 Misses	Computing Threads Start...
	Global	Local	🟨 3D 🟦 VIDEO	Active	Stalled	Idle	Busy	Bott...		
ProcessUV	960 x 544	16 x 16	8.590ms	0.229	0.706	0.065	0.000	0.000	0	112,556
[Outside any task]			17.426ms	0.103	0.489	0.408	0.000	0.000	0	18,003
Selected 1 row(s):	960 x 544	16 x 16	8.590ms	0.229	0.706	0.065	0.000	0.000	0	112,556

MFXCloneSession (0x98c)

Computing Task (GPU)
Start: 5016.917ms Duration: 2.149ms
Computing Task (GPU): ProcessUV
Global Size: 960 x 544
Local Size: 16 x 16

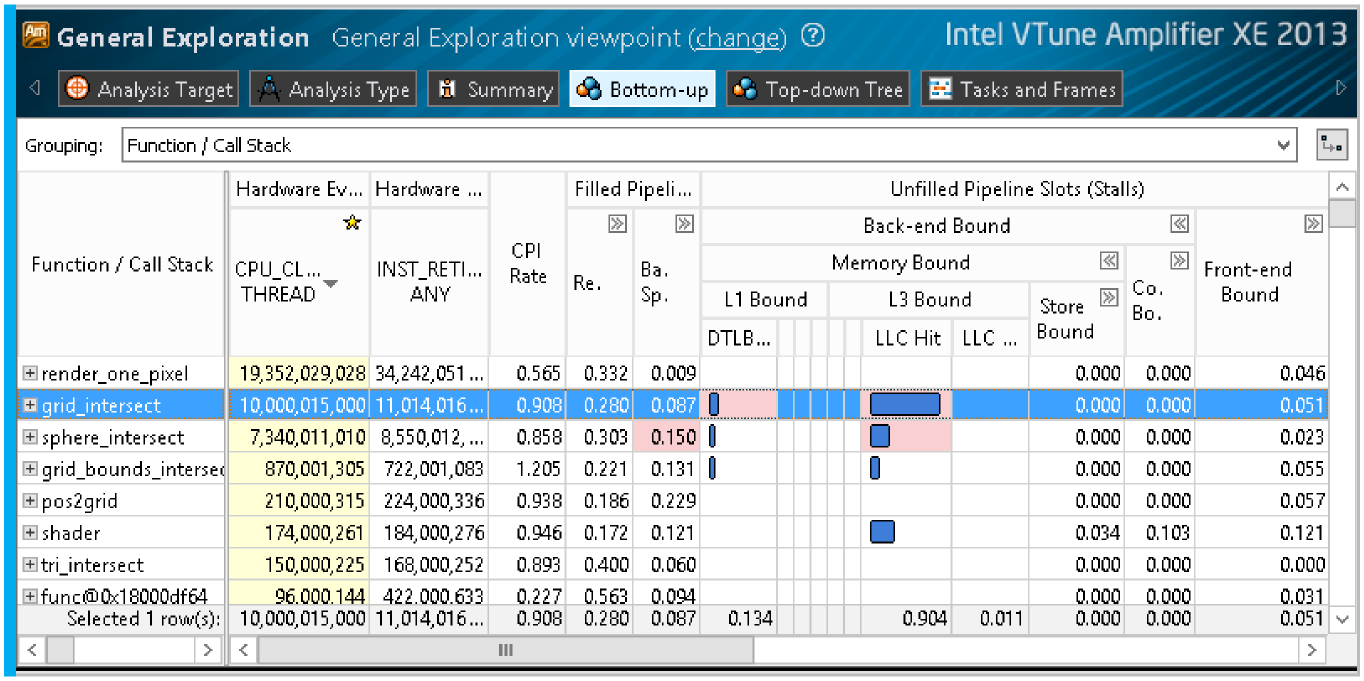
GPU Engines Usage
3D: 1
VIDEO: 0

CPU Time
0.0%

12 Profiling GPU computations

For more information regarding performance and optimization choices in Intel® software products, visit http://software.intel.com/en-us/articles/optimization-notice.

Sign up for future issues | Share with a friend



General Exploration General Exploration viewpoint (change) Intel VTune Amplifier XE 2013

Grouping: Function / Call Stack

Function / Call Stack	Hardware Ev...	Hardware ...	CPI Rate	Filled Pipeli...		Unfilled Pipeline Slots (Stalls)					
						Back-end Bound					
						Memory Bound				Co. Bo.	Front-end Bound
						L1 Bound	L3 Bound		Store Bound		
	CPU_CL... THREAD	INST_RETI... ANY		Re.	Ba. Sp.	DTLB...	LLC Hit	LLC ...			
render_one_pixel	19,352,029,028	34,242,051...	0.565	0.332	0.009				0.000	0.000	0.046
grid_intersect	10,000,015,000	11,014,016...	0.908	0.280	0.087	▮	▮		0.000	0.000	0.051
sphere_intersect	7,340,011,010	8,550,012,...	0.858	0.303	0.150	▮	▮		0.000	0.000	0.023
grid_bounds_intersec	870,001,305	722,001,083	1.205	0.221	0.131	▮	▮		0.000	0.000	0.055
pos2grid	210,000,315	224,000,336	0.938	0.186	0.229				0.000	0.000	0.057
shader	174,000,261	184,000,276	0.946	0.172	0.121		▮		0.034	0.103	0.121
tri_intersect	150,000,225	168,000,252	0.893	0.400	0.060				0.000	0.000	0.000
func@0x18000df64	96,000,144	422,000,633	0.227	0.563	0.094				0.000	0.000	0.031
Selected 1 row(s):	10,000,015,000	11,014,016...	0.908	0.280	0.087	0.134	0.904	0.011	0.000	0.000	0.051

13 Top-down structure of General Exploration profile for 4th generation Intel® Core™ processors

See the impact.

30-day trial software

Evaluate the latest Intel® Parallel Studio XE and Intel® Cluster Studio XE >

Evaluate Intel® System Studio for embedded and mobile system developers >

Coprocessor Debugging Support in Intel® Parallel Studio XE

By Keven Boell, *Software Engineer, Intel*

The recently introduced Intel® Xeon Phi™ coprocessors are connected via PCI Express* to host processor(s) to form a new type of hybrid computing system. They contain up to 61 cores, with four threads per core, all sharing their own coherent cache, and each core is equipped with 512-bit wide SIMD vector units. The Intel Xeon Phi coprocessor is able to run applications natively when they are cross-compiled for this architecture. Native means that the entire program runs completely on the coprocessor within an embedded Linux* system and nothing on the host.

Besides running programs solely on the host or on the target, Intel Xeon Phi allows the developer to offload performance-critical parts of a host application to the installed coprocessors. The offload model offers simplicity when adopting a program to offload selected kernels of computation. This programming model allows the user to offload code by using code decorations in C, C++, and Fortran. The decoration can be done by using the so-called Language Extensions for Offload (LEO), either with pragmas (see **Figure 1**) or by using shared keywords (see **Figure 2**) as language extensions for variables and functions. Intel has worked, with many others, in the OpenMP* standardization effort to promote inclusion of offload capabilities into OpenMP.

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

1

Pragma code decoration

```
void my_func() {
    int my_var = 0;
    #pragma offload target (mic)
    #pragma omp parallel for
    for (i=0; i < count; i++) {
          // do something
    }
}
```

2

Language extensions for offload (LEO)

```
_Cilk_shared int my_shared_var;

_Cilk_shared void my_func() {
    my_shared_var = 42;

    #pragma omp parallel for
    for (i=0; i < count; i++) {
          // do something
    }
}

// Call offload function
_Cilk_offload my_func();
```

3

Coprocessor example code (dot_product.c)

```
#include <stdio.h>
#include “offload.h”

#define SIZE 100

#ifdef OFFLOAD
#define MODE “OFFLOAD”
#else
#define MODE “NATIVE”
#endif

#ifdef OFFLOAD
__declspec(target(mic))
#endif
int A[SIZE], B[SIZE];

#ifdef OFFLOAD
__declspec(target(mic))
#endif
int dotproduct;

int compute_dotproduct () {
    int i;
    int ret;

    #ifdef OFFLOAD
```

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

3 CONT.

Coprocessor example code (dot_product.c)

```
    #pragma offload target (mic)
    #endif
    {
            #pragma omp parallel shared(A, B) private(i)
            {
                    #pragma omp master
                    {
                            #ifdef __MIC__
                            ret = 1;
                            #else
                            ret = 0;
                            #endif
                    }

                    #pragma omp for reduction(+:dotproduct)
                    for (i=0; i < SIZE; i++) {
                            dotproduct = dotproduct + (A[i] * B[i]);
                    }
            }
    }
    return ret;
}

void init () {
    int i;
    /* initialize with dummy values. */
    for (i = 0; i < SIZE; i++) {
            A[i] = i * 2;
            B[i] = i + 2;
    }
}

int main (int argc, char *argv[]) {
    int mode;
    init ();
    mode = compute_dotproduct ();
    printf("[%s] computed dot product on %s = %i\n",
            MODE, (mode == 1 ? "MIC" : "HOST"), dotproduct);
    return 0;
}
```

The draft OpenMP* 4.0[1] specification also provides a standardized version of the capabilities found in LEO and promises to be the new syntax of choice for new coding efforts. The main advantage for the developer is that offloading parallel regions to coprocessors can be done by just extending already existing OpenMP directives in the code. All these are supported by Intel® compilers and tools.

Every developer knows that no piece of software is perfect and absolutely bug-free. But what if the application is not doing what it is supposed to do on the coprocessor? The importance of debugging tools is evident to any developer who has spent more time finding bugs than developing new code. Debuggers can help here as they support the developer in tracking down, isolating, and removing bugs from the developed program. This gets especially tricky if you have a distributed execution model running across multiple targets, such as the offload model. This article will describe how such debugging tasks can be addressed using Intel® Parallel Studio XE for programs running completely, or partially, on an Intel Xeon Phi coprocessor.

Different Scenarios for Coprocessor Debugging

There are two main debug scenarios for the user. The first is debugging native applications, which means that they are running completely on the coprocessor side. To debug native applications, a target agent on the coprocessor and a debug engine on the host are required, where the host engine is connecting to the target agent. The advantage is that this mechanism allows a GUI (e.g., Eclipse*)[2] on top of the debug engine on the host, instead of doing command line debugging in a target terminal.

The second scenario is debugging offload applications which run on the host with selected portions offloaded onto one or more coprocessors. For this purpose several debug engines are required, because specific parts of the program run on different coprocessor targets. The corresponding code blocks for the host and target will be executed in separate processes, each of them being debugged by a different debug engine. To get this scenario as smooth as possible for the user, a transparent GUI solution is necessary to hide most of the debugging architecture underneath and give the user the impression of a transparent debug session across all the different processes.

Intel Parallel Studio XE supports both scenarios. It includes a coprocessor-aware GNU Debugger* (GDB*)[3] for debugging native applications, as well as a complete offload debug solution also based on GDB for Linux and Windows*. Both solutions and their usage will be further described in this article by using a native and an offload example application.

1. http://openmp.org/wp/openmp-specifications/
2. http://www.eclipse.org/
3. http://www.gnu.org/software/gdb/

Debugging Native Coprocessor Applications

Native coprocessor applications will usually be cross-compiled on the host, copied to the coprocessor, and subsequently executed there. Logically, it also makes sense to debug them directly on the target. Intel Parallel Studio XE contains a GDB target agent for debugging such applications on the coprocessor and a GDB for connecting to the target agent. Both are aware of all the architecture specialties, including its application binary interfaces (ABI) and the 512-bit wide vector capabilities and related mask registers.

Let's start with a debug session directly on the target by using a simple native application (see **Figure 3**). The next steps will show you how to get a basic debug session up and running:

1) Cross-compile the program for Intel® Xeon Phi™-based Many Integrated Core architecture (MIC) using the Intel compiler:

```
icc –g –O0 –mmic –openmp –o dot_product dot_product.c
```

2) Copy the application and the sources to the target:

```
scp dot_product* mic0:~
```

3) In addition, it is required to copy MIC shared objects which are required by the compiler runtime system and the application to the coprocessor target. For example, this could be the OpenMP* library (libiomp5.so), which can be found in the Intel Parallel Studio installation.

4) Copy the target agent to the coprocessor:

```
scp gdbserver mic0:~
```

5) Start the target agent on the coprocessor:

```
ssh mic0 "~/gdbserver --once :1234 ~/dot_product"
```

6) Launch GDB on the host and connect to the target:

```
gdb ./dot_product

(gdb) target remote mic0:1234
```

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

GDB is now connected to the target agent on the coprocessor and you will have a fully functional debug session for your application directly on the target. To get to the main entry point of your C/C++ application, type "`break main`" and "`continue`" in the GDB console. This will set a breakpoint to the main function and continue the debuggee under control of the debugger. When stopped inside a parallel region, the developer may be interested in the current threads and their state. An overview of all threads, which are currently active, can be seen by typing "`info threads`". Another interesting aspect is the contents of the Intel Xeon Phi coprocessor-specific registers. They can be accessed using "`print $zmm0`" to "`print $zmm31`" for all 32 ZMM (512-bit SIMD) registers. A lot more can be done with GDB during a debug session, which will not be covered here as it will go beyond the scope of this article. Comprehensive online help, which is accessible by typing "`help`", and the official documentation of GDB,[4] will provide more insight on what can be done through the command line.

4. http://sourceware.org/gdb/current/onlinedocs/gdb/

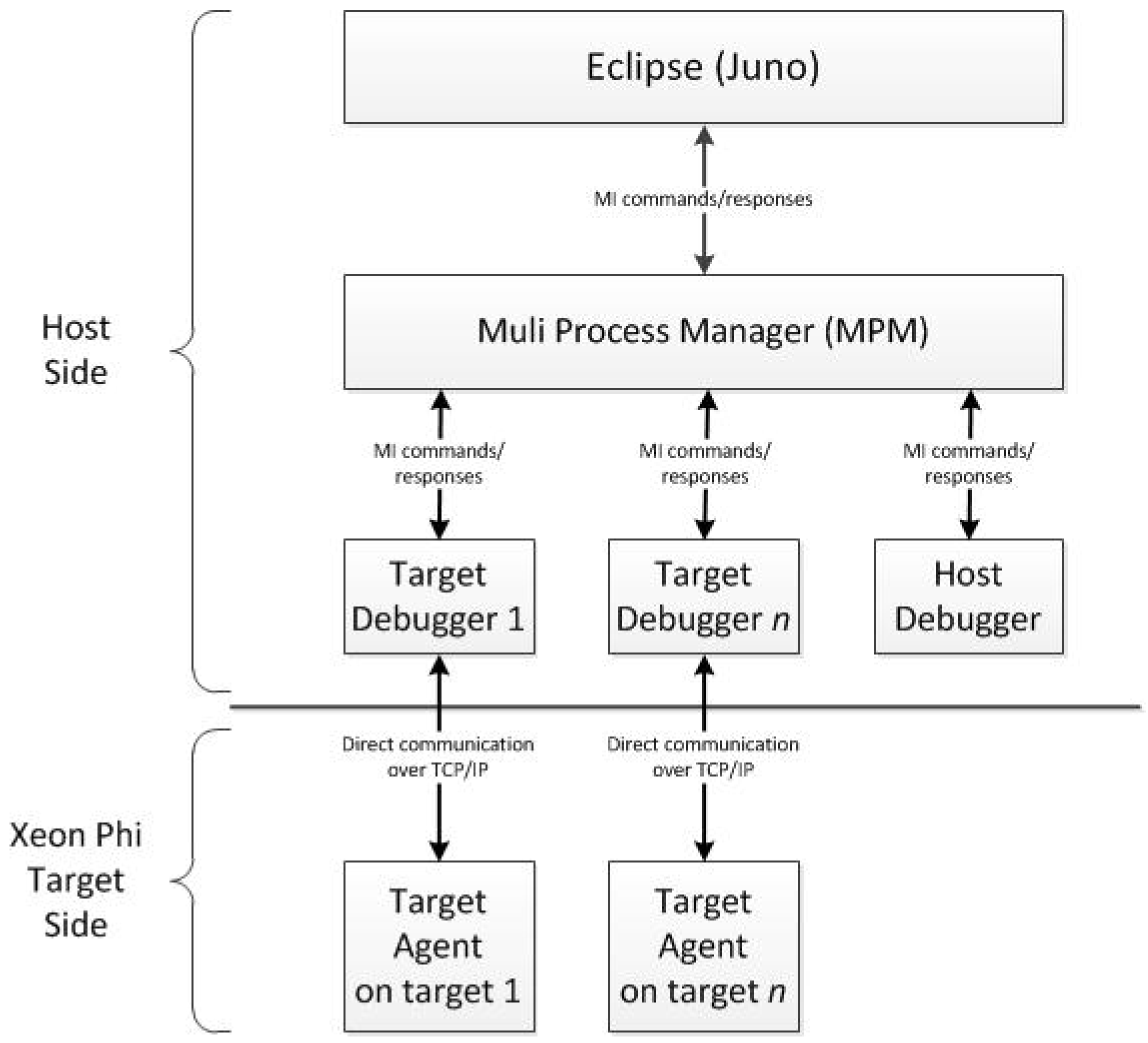
Debugging Offload Applications

Besides the traditional debugging approaches for native coprocessor applications, the offload model presents a new challenge for debugging tools: debug host- and target-side code of an offload application running in different processes at the same time. The ultimate goal of such a debugging solution is to present to the developer a transparent, single, source-level view of the program at the control flow and data levels. Since different parts of the program run on different hosts in different processes, the challenge is to reconstruct the single control and data views.

The offload model, coupled with the possibility of having multiple coprocessor cards in one system, creates a large, heterogeneous programming environment, which cannot be debugged with only one GDB debug engine. Because the host-side code runs in a process and each offload part runs in another process, the developer would have to run n+1 debugging sessions and attach them to the respective offload processes. It would take one debug session for the host side and n debug sessions for the target side processes, where n is the amount of Intel Xeon Phi coprocessors in the system. Once a debug engine is attached to one of the processes, only the specific code which runs in this process can be debugged (e.g., one offload region). The main disadvantage is that the developer needs to have a good knowledge about which debugging session is currently active and which session is doing what. It's clear that this is not intuitive. The challenge is to hide these sessions from the user as much as possible, and let it look like a single debugging session. In order to achieve this level of debugging transparency, the Intel Parallel Studio XE 2013 contains tools for Linux debugging (Multi Process Manager) and Windows debugging (Visual Studio* Debug Extension) that do the trick for the user.

Linux: Multi Process Manager (MPM)

The MPM serves as a communication layer between the multiple GDB instances and the Eclipse GUI on a Linux host. The main principle of MPM is simple: it takes the machine interface (MI), commands from the GUI and redirects them to the right debug engine underneath. MI is a line-based, text-oriented interface to GDB and Eclipse. In another way, MPM is aggregating the MI answers from the debug engines, for example when a breakpoint was hit, and forwards the response to the GUI, which can then update the respective windows. **Figure 4** shows in detail what this looks like.



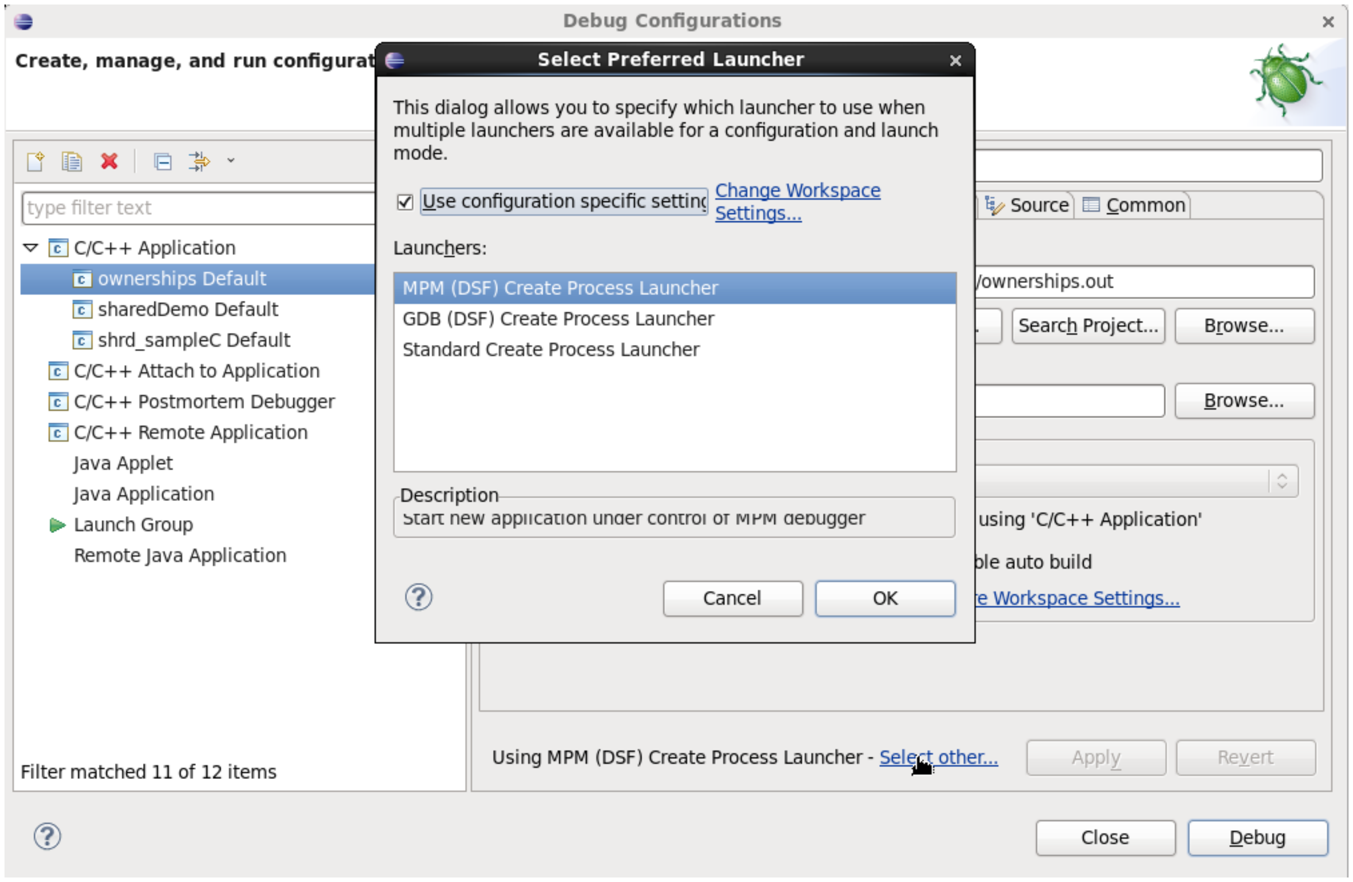
4 MPM* internal workflow

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

The MPM plugin for Eclipse is located in the Intel Parallel Studio installation and can be installed using the standard Eclipse plugin installation procedure.

Once the plugin is installed, the developer needs to create a project either by importing existing code or by starting from scratch. **Figure 3** shows a sample application, which needs to be compiled with -DOFFLOAD to act as an offload application. If one wants to debug a Fortran offload program, it is required to make Eclipse aware of Fortran by installing the "Photran" component from the Eclipse standard repository.

To start the actual debug session, Eclipse requires a so-called "debug configuration." Such a configuration defines the basic debugging settings and can be created by clicking on "Run" > "Debug Configurations" in the Eclipse main dialog. In the following dialog, MPM needs to be chosen as process launcher (see **Figure 5**), and the application to be debugged needs to be set. The rest will be automatically configured by MPM.


Debug Configurations

Create, manage, and run configurat

Select Preferred Launcher

This dialog allows you to specify which launcher to use when multiple launchers are available for a configuration and launch mode.

☑ Use configuration specific setting | Change Workspace Settings...

Launchers:
- MPM (DSF) Create Process Launcher
- GDB (DSF) Create Process Launcher
- Standard Create Process Launcher

Description: Start new application under control of MPM debugger

Cancel | OK

type filter text
- C/C++ Application
  - ownerships Default
  - sharedDemo Default
  - shrd_sampleC Default
- C/C++ Attach to Application
- C/C++ Postmortem Debugger
- C/C++ Remote Application
- Java Applet
- Java Application
- Launch Group
- Remote Java Application

Filter matched 11 of 12 items

Source | Common

/ownerships.out

Search Project... | Browse... | Browse...

using 'C/C++ Application'

ble auto build

e Workspace Settings...

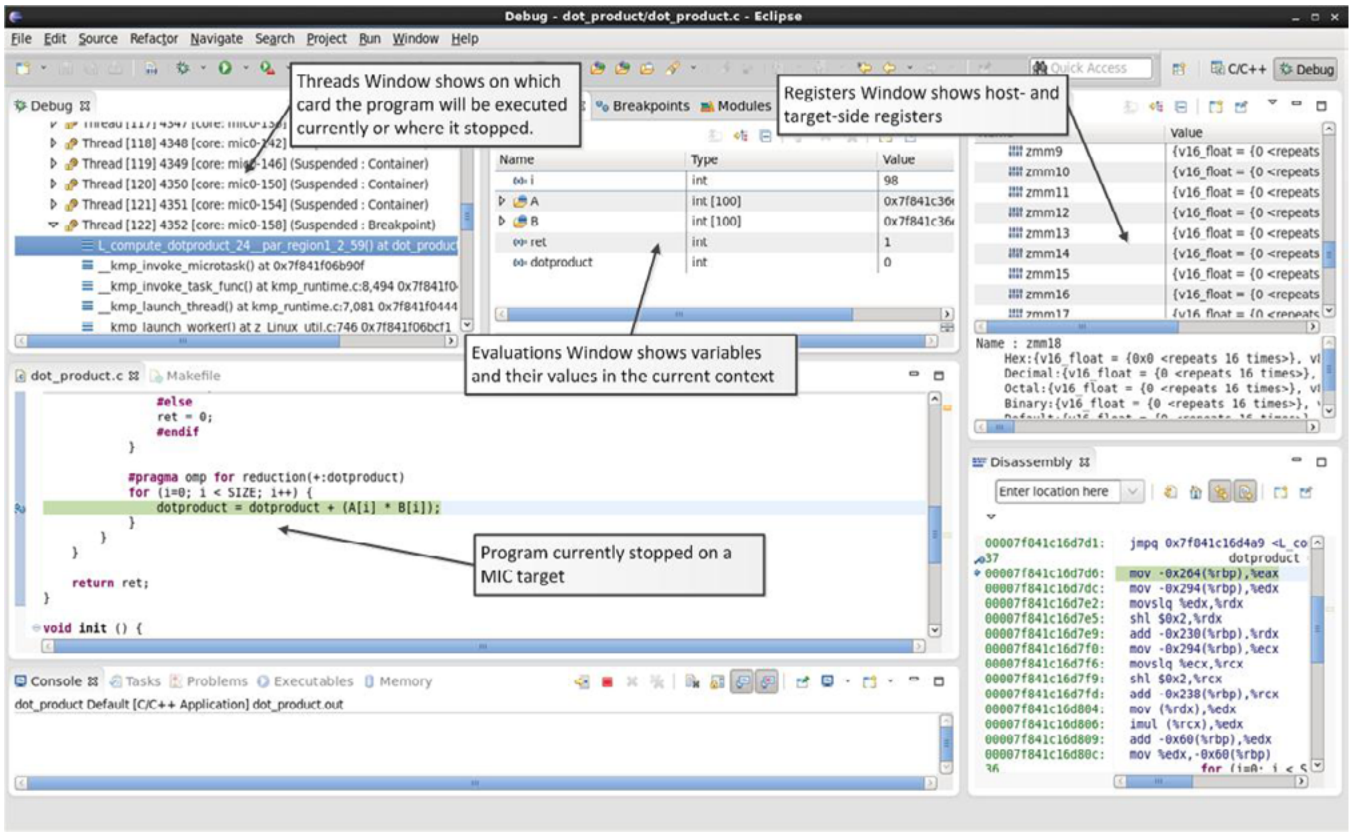
Using MPM (DSF) Create Process Launcher - Select other...

Apply | Revert | Close | Debug

5 Eclipse MPM plugin selection while creating a new debug configuration

5. http://www.eclipse.org/photran/

After the debug session starts, Eclipse turns into its debug view, where all the common debug-relevant information can be seen, such as the registers of the host and target side, the values of variables currently in scope, the stack trace, and much more. **Figure 6** shows a typical debug session of an offload program in Eclipse.



Threads Window shows on which card the program will be executed currently or where it stopped.

Registers Window shows host- and target-side registers

Evaluations Window shows variables and their values in the current context

Program currently stopped on a MIC target

```
Thread [118] 4348 [core: mic0-142]
Thread [119] 4349 [core: mic0-146] (Suspended : Container)
Thread [120] 4350 [core: mic0-150] (Suspended : Container)
Thread [121] 4351 [core: mic0-154] (Suspended : Container)
Thread [122] 4352 [core: mic0-158] (Suspended : Breakpoint)
    L_compute_dotproduct_24__par_region1_2_59() at dot_produc
    __kmp_invoke_microtask() at 0x7f841f06b90f
    __kmp_invoke_task_func() at kmp_runtime.c:8,494 0x7f841f0
    __kmp_launch_thread() at kmp_runtime.c:7,081 0x7f841f0444
    __kmp_launch_worker() at z_Linux_util.c:746 0x7f841f06bcf1
```

Name	Type	Value
i	int	98
A	int [100]	0x7f841c36
B	int [100]	0x7f841c36
ret	int	1
dotproduct	int	0

Name	Value
zmm9	{v16_float = {0 <repeats
zmm10	{v16_float = {0 <repeats
zmm11	{v16_float = {0 <repeats
zmm12	{v16_float = {0 <repeats
zmm13	{v16_float = {0 <repeats
zmm14	{v16_float = {0 <repeats
zmm15	{v16_float = {0 <repeats
zmm16	{v16_float = {0 <repeats
zmm17	{v16_float = {0 <repeats

```
Name : zmm18
    Hex:{v16_float = {0x0 <repeats 16 times>}, v
    Decimal:{v16_float = {0 <repeats 16 times>},
    Octal:{v16_float = {0 <repeats 16 times>}, v
    Binary:{v16_float = {0 <repeats 16 times>},
```

```
            #else
            ret = 0;
            #endif
        }

        #pragma omp for reduction(+:dotproduct)
        for (i=0; i < SIZE; i++) {
            dotproduct = dotproduct + (A[i] * B[i]);
        }
    }
  }

  return ret;
}

void init () {
```

dot_product Default [C/C++ Application] dot_product.out

```
00007f041c16d7d1:   jmpq 0x7f041c16d4a9 <L_co
37                                 dotproduct
00007f841c16d7d6:   mov -0x264(%rbp),%eax
00007f841c16d7dc:   mov -0x294(%rbp),%edx
00007f841c16d7e2:   movslq %edx,%rdx
00007f841c16d7e5:   shl $0x2,%rdx
00007f841c16d7e9:   add -0x230(%rbp),%rdx
00007f841c16d7f0:   mov -0x294(%rbp),%ecx
00007f841c16d7f6:   movslq %ecx,%rcx
00007f841c16d7f9:   shl $0x2,%rcx
00007f841c16d7fd:   add -0x238(%rbp),%rcx
00007f841c16d804:   mov (%rdx),%edx
00007f841c16d806:   imul (%rcx),%edx
00007f841c16d809:   add -0x60(%rbp),%edx
00007f841c16d80c:   mov %edx,-0x60(%rbp)
36                                 for (i=0; i < S
```

6 Eclipse* offload debug session showing the most important windows, containing variables, registers, disassembly, and threads

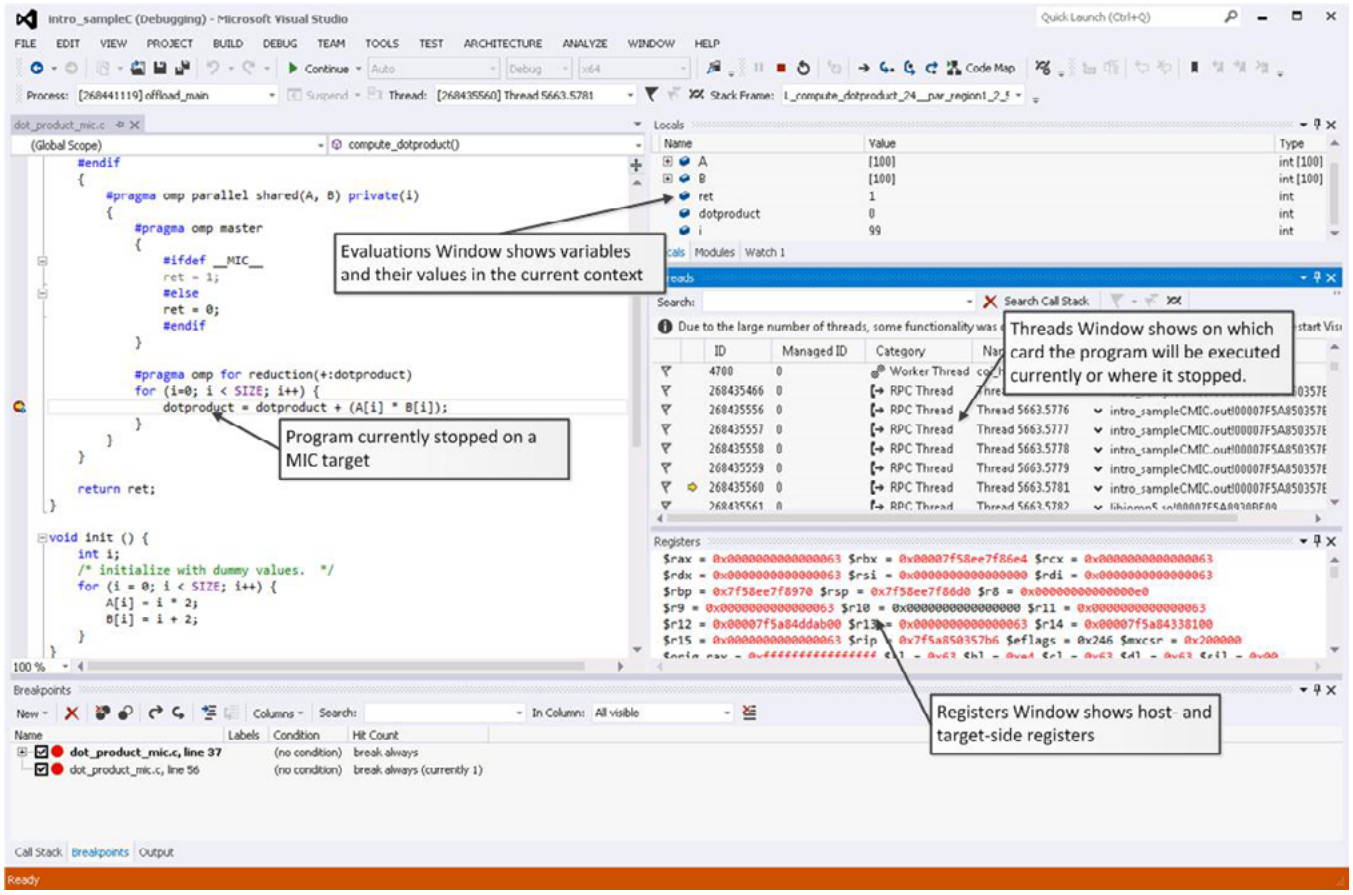
Windows: Visual Studio (VS) Debug Extension

Intel Parallel Studio XE also supports debugging of the offload model under Windows. When installing the package, the Intel Xeon Phi coprocessor debug extensions for Visual Studio will be installed and integrated similarly to the Intel compiler. The internal functionality of the Visual Studio Debug Extension for the Intel Xeon Phi coprocessor is pretty much the same as the MPM. The difference is that the extension is translating GUI requests coming from VS into MI, which can be understood by GDB, as VS is not using MI to communicate with the debug engines. In the other direction, MI answers from the GDB debug engines will be translated into GUI updates for VS. Similar to MPM, this solution gives the developer a transparent debug session across multiple coprocessor targets.

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

To start debugging offload applications, a Visual Studio solution is required. The code from **Figure 3** can also be used here as a sample application. The difference from the Linux Eclipse debugging solution is that the developer doesn't need to configure anything. A coprocessor debugging session can be started just by clicking the "Local Windows Debugger" button in the main symbol bar of Visual Studio.

Visual Studio will turn into its debug view where all the required debugging information can be seen. **Figure 7** shows such a debug session example in Visual Studio* 2012.


intro_sampleC (Debugging) - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ARCHITECTURE ANALYZE WINDOW HELP
Continue
Code Map
dot_product_mic.c
Locals
Modules
Watch 1
Threads
Search Call Stack
Registers
Breakpoints
Columns
Call Stack
Breakpoints
Output
Ready
Evaluations Window shows variables and their values in the current context
Threads Window shows on which card the program will be executed currently or where it stopped.
Program currently stopped on a MIC target
Registers Window shows host- and target-side registers

Process: [268441119] offload_main   Thread: [268435560] Thread 5663.5781   Stack Frame: L_compute_dotproduct_24__par_region1_2_5

(Global Scope)   compute_dotproduct()

```
    #endif
    {
        #pragma omp parallel shared(A, B) private(i)
        {
            #pragma omp master
            {
                #ifdef __MIC__
                ret = 1;
                #else
                ret = 0;
                #endif
            }

            #pragma omp for reduction(+:dotproduct)
            for (i=0; i < SIZE; i++) {
                dotproduct = dotproduct + (A[i] * B[i]);
            }
        }
    }

    return ret;
}

void init () {
    int i;
    /* initialize with dummy values.  */
    for (i = 0; i < SIZE; i++) {
        A[i] = i * 2;
        B[i] = i + 2;
    }
}
```

Name	Value	Type
A	[100]	int [100]
B	[100]	int [100]
ret	1	int
dotproduct	0	int
i	99	int

Due to the large number of threads, some functionality was

	ID	Managed ID	Category	Name	
	4700	0	Worker Thread	co	
	268435466	0	RPC Thread	Thre	
	268435556	0	RPC Thread	Thread 5663.5776	intro_sampleCMIC.out!00007F5A850357E
	268435557	0	RPC Thread	Thread 5663.5777	intro_sampleCMIC.out!00007F5A850357E
	268435558	0	RPC Thread	Thread 5663.5778	intro_sampleCMIC.out!00007F5A850357E
	268435559	0	RPC Thread	Thread 5663.5779	intro_sampleCMIC.out!00007F5A850357E
➡	268435560	0	RPC Thread	Thread 5663.5781	intro_sampleCMIC.out!00007F5A850357E
	268435561	0	RPC Thread	Thread 5663.5782	libiomp5.so!00007F5A8930BE09

```
$rax = 0x0000000000000063 $rbx = 0x00007f58ee7f86e4 $rcx = 0x0000000000000063
$rdx = 0x0000000000000063 $rsi = 0x0000000000000000 $rdi = 0x0000000000000063
$rbp = 0x7f58ee7f8970 $rsp = 0x7f58ee7f86d0 $r8 = 0x00000000000000e0
$r9 = 0x0000000000000063 $r10 = 0x0000000000000000 $r11 = 0x0000000000000063
$r12 = 0x00007f5a84ddab00 $r13 = 0x0000000000000063 $r14 = 0x00007f5a84338100
$r15 = 0x0000000000000063 $rip = 0x7f5a850357b6 $eflags = 0x246 $mxcsr = 0x200000
$orig_rax = 0xffffffffffffffff $al = 0x63 $bl = 0xe4 $cl = 0x63 $dl = 0x63 $sil = 0x00
```

Name	Labels	Condition	Hit Count
dot_product_mic.c, line 37		(no condition)	break always
dot_product_mic.c, line 56		(no condition)	break always (currently 1)

7 Visual Studio* offload debug session showing the variables, threads, and registers

Summary

Aside from debugging native coprocessor applications, debugging offload processes can be challenging, due to the program execution spread over different processes and targets. To help developers focus on isolating issues, Intel Parallel Studio XE contains tools which offer debug solutions for native applications, as well as for offload programs. All tools are aware of the Intel Xeon Phi coprocessor specialties and provide a single source-line view of the program's control flow and variables on the command line, as well as in Eclipse and Visual Studio. ●

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

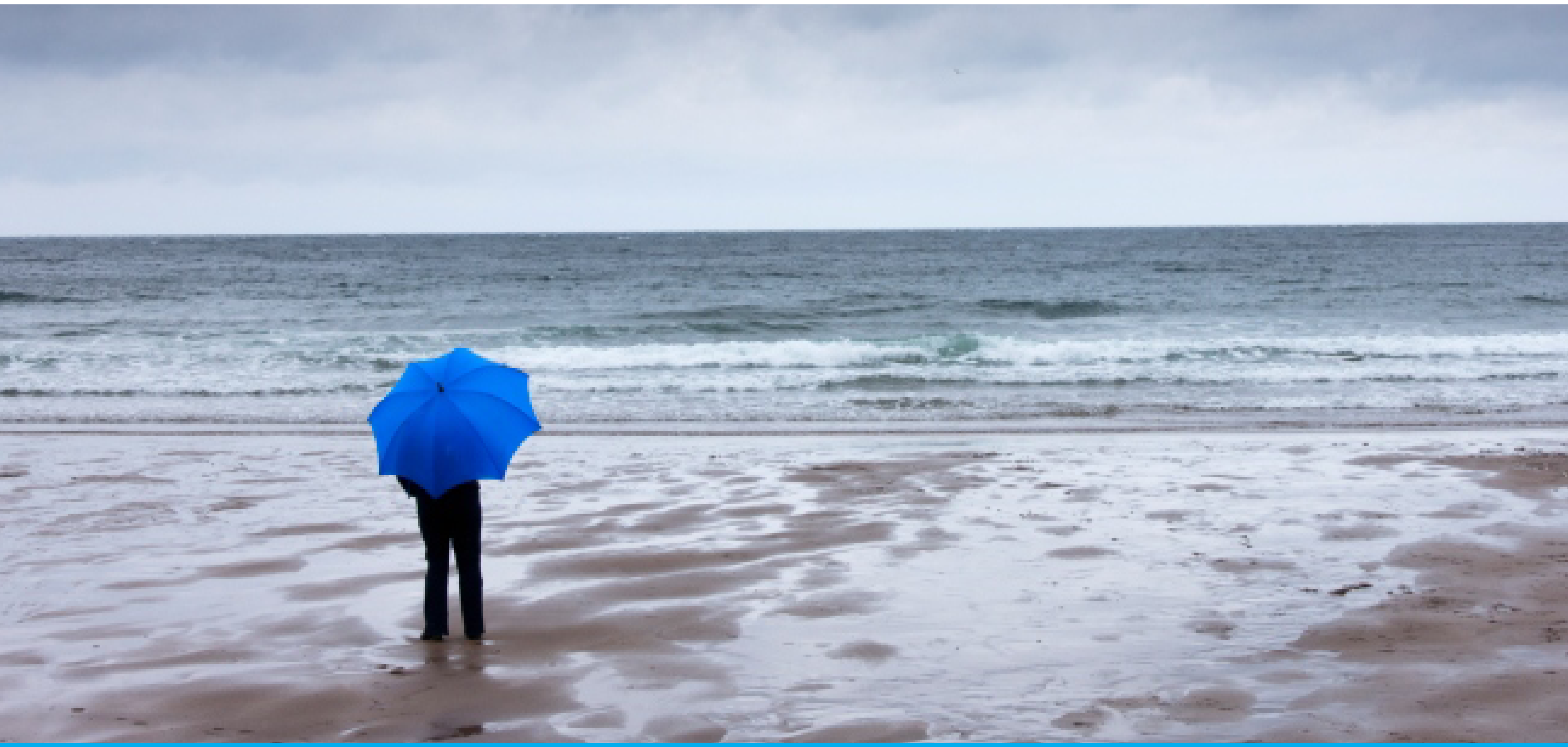
FURTHER READINGS

- Building a Native Application for Intel® Xeon Phi™ Coprocessors
  http://software.intel.com/en-us/articles/building-a-native-application-for-intel-xeon-phi-coprocessors
- Programming and Compiling for Intel® Many Integrated Core Architecture
  http://software.intel.com/en-us/articles/programming-and-compiling-for-intel-many-integrated-core-architecture

Threading, tuning, debugging, and more.
Advanced skills for high performance application developers.

Free Intel Software Technical Fall Webinar Series 2013
Sessions from Sept. 17–Nov. 12

SEE FULL SERIES AND REGISTER NOW >



Full Scale Ahead: The Weather Research and Forecast (WRF) Model and Intel® Cluster Studio XE 2013

by Mark Lubin and Scott McMillan, *Intel Corporation,* **Christopher G. Kruse and Davide Del Vento,** *National Center for Atmospheric Research (NCAR),* **and Raffaele Montuoro,** *Texas A&M University*

Abstract

The Weather Research and Forecast (WRF) Model is a next-generation, mesoscale weather prediction system that is extensively used across a wide range of meteorological applications. We have successfully scaled the WRF Model up to 65,536 Intel® Xeon® E5-2670 cores with the Intel® MPI Library and DAPL UD. A new workload, based on high-resolution (1km) simulations of hurricane Katrina was selected for this study. The objective was to demonstrate WRF scalability on "commodity" supercomputers using Intel® Cluster Studio XE 2013 software tools, including Intel® compilers and the Intel MPI Library. You'll learn about the process and collaboration that teams at the National Center for Atmospheric Research (NCAR), Texas A&M University, and Intel used to successfully scale WRF on the NCAR Yellowstone system (debuted at #13 on the Nov. 2012 Top500 list). Communication protocol choices between DAPL and DAPL UD were easily explored with the standards-driven Intel MPI Library—enabling researchers to scale to the limits of the Yellowstone system.

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

The Weather Research and Forecast Model

The Weather Research and Forecast Model (WRF) is one of the premier open source numerical weather forecasting models used in operational, research, and educational settings. It has been collaboratively developed by the National Center for Atmospheric Research (NCAR), the National Oceanic and Atmospheric Administration (NOAA), the Air Force Weather Agency (AFWA), the Naval Research Laboratory (NRL), the University of Oklahoma, and the Federal Aviation Administration (FAA). WRF is currently in operational use at NCAR, NCEP, AFWA, and other centers and has a large user base in the environmental sciences (see Skamarock et. al., 2008). We used WRF's latest version, 3.5, (April 18, 2013), built with the Advanced Research WRF (ARW) dynamical core, developed and maintained by the Mesoscale and Microscale Meteorology (MMM) Division of NCAR's Earth System Laboratory (NESL).

The recently released version 4.1 of the Intel® MPI Library includes features designed for scalability on very large systems, so WRF was the perfect candidate to demonstrate these scaling properties.

WRF has an extensible design, which makes it possible to easily add cutting-edge physics, chemistry, and hydrology models, as well as other features contributed by the research community. WRF is also known to scale well to large numbers of cores (Michalakes et. al., 2008). The recently released version 4.1 of the Intel MPI Library includes features designed for scalability on very large systems, so WRF was the perfect candidate to demonstrate these scaling properties. A large, high-resolution dataset of Hurricane Katrina (Patricola et. al., 2012) was benchmarked on Yellowstone, an Intel® Xeon® processor-based petascale HPC system in the NCAR-Wyoming Supercomputering Center (NWSC). Given the initial and boundary conditions on a grid covering the Gulf of Mexico at 1km horizontal resolution starting August 25, 2005 at 0000 UTC, the WRF simulation attempts to predict the landfall time and location of Hurricane Katrina (the real Katrina made landfall on August 29, 2005).

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

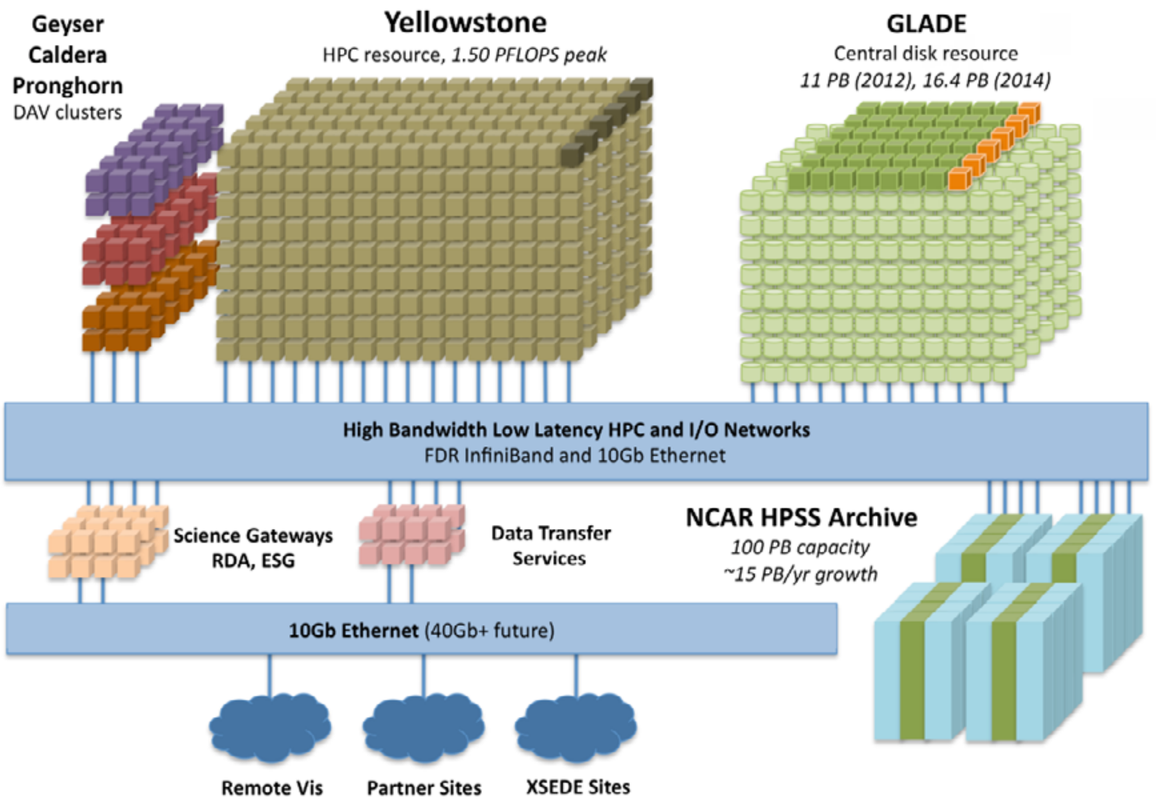
Computational Environment: Yellowstone, Intel Cluster Studio XE 2013, Intel MPI Library, and DAPL UD.

This study collects results obtained from the very first attempts to run real-life WRF simulations at full scale on the Yellowstone system after its installation in late 2012.

1 Yellowstone cluster

Yellowstone consists of 4,518 compute nodes, each with 2 Intel® Xeon® E5-2670 processors (8 cores each), for a total of 72,288 cores. The nodes are connected using a fat tree topology InfiniBand* network.

WRF version 3.5 was built using Intel® Composer XE 2013 with Intel MPI Library version 4.1. Version 4.2 of the netCDF library was used. The key compiler options were "-O3 -align all -fno-alias -fp-model precise". Minor modifications were made to the WRF source code in order to facilitate this study. WRF's bundled Runtime System Library (RSL) comes with a hardcoded limit of 10,000 MPI processes. In order to use more MPI processes, the value of RSL_MAXPROC was increased, and all formatting instances for the standard output and error file names were modified appropriately. Disk I/O was problematic at core counts of 16K and above. Since our goal was to evaluate simulation speed, we toggled off output writing. Additionally, the "Known Problems and Fixes" patch on the WRF Model user's site accessed June 19, 2013, was applied (**http://www.mmm.ucar.edu/wrf/users/wrfv3.5/known-prob-3.5.html**).



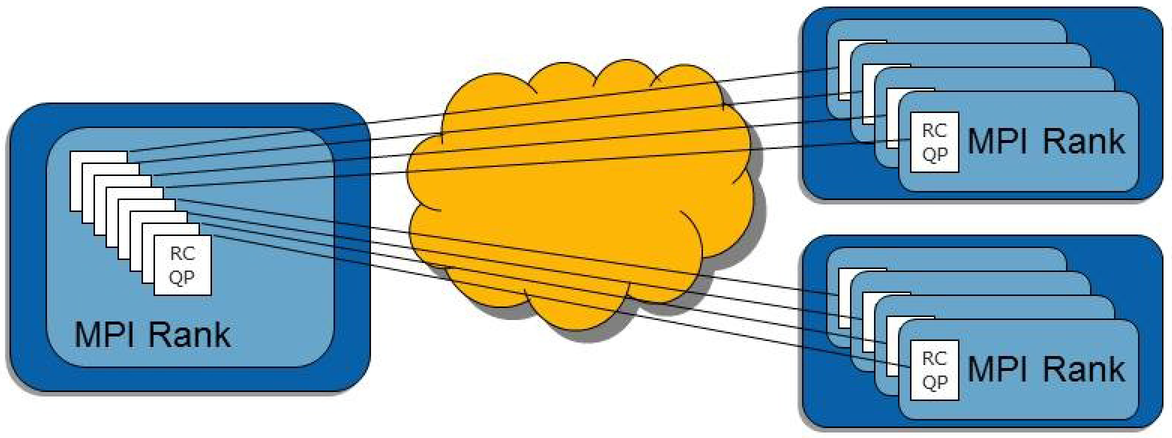
2 Yellowstone cluster environment

The Intel MPI Library provides several interfaces to InfiniBand networks, including DAPL. InfiniBand uses a Queue Pair (QP) object to initiate data transfers. An InfiniBand QP can be created to provide a specific type of service. Next, we will describe the two types used in the Intel MPI Library. They are Reliable Connected (RC) and Unreliable Datagram (UD).

The RC QP type provides reliable transfer of all messages and RDMA operations. However, it requires substantial resource allocation to occur for every MPI rank. This memory is unavailable to the application.

This study collects results obtained from the very first attempts to run real-life WRF simulations at full scale on the Yellowstone system after its installation in late 2012.

Reliable Connection mode

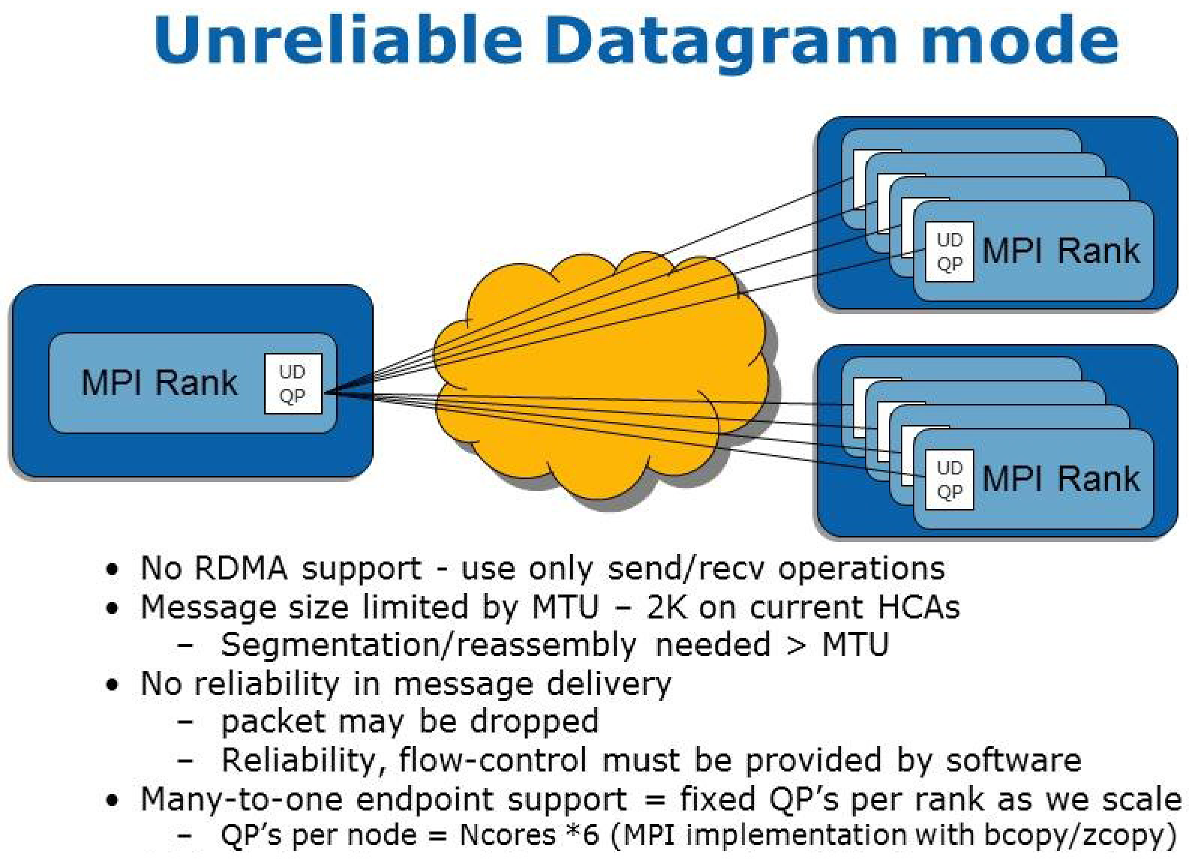


- RDMA support
- Large messages, no segmentation is needed
- Reliable, in-order delivery, flow-control
- One-to-One endpoint = increasing QP footprint as we scale
  - QP's per node = $Ncores^2$ * (Nnodes-1)

3 Reliable connection mode in the Intel® MPI Library

As the name implies, the UD QP type does not guarantee delivery of the messages it sends and cannot perform RDMA operations. After some time, messages that are not delivered trigger retransmission requests—which actually leads to increased resilience from transient network issues. Unlike the RC QP type, a UD QP is not connected to any other QP and therefore can receive messages from any other UD QP in the fabric that knows its "address." Therefore, the maximum amount of memory to support UD operations is typically much smaller than in an RC case. The downside is that the Intel MPI library must do segmentation and reassembly of each message in MTU size fragments. However, this is acceptable in large clusters where dynamic connections ultimately end up connected all-to-all. It not only results in better memory utilization, but depending on the application, often yields better scalability, as well.

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.



Unreliable Datagram mode

- No RDMA support - use only send/recv operations
- Message size limited by MTU – 2K on current HCAs
  - Segmentation/reassembly needed > MTU
- No reliability in message delivery
  - packet may be dropped
  - Reliability, flow-control must be provided by software
- Many-to-one endpoint support = fixed QP's per rank as we scale
  - QP's per node = Ncores *6 (MPI implementation with bcopy/zcopy)

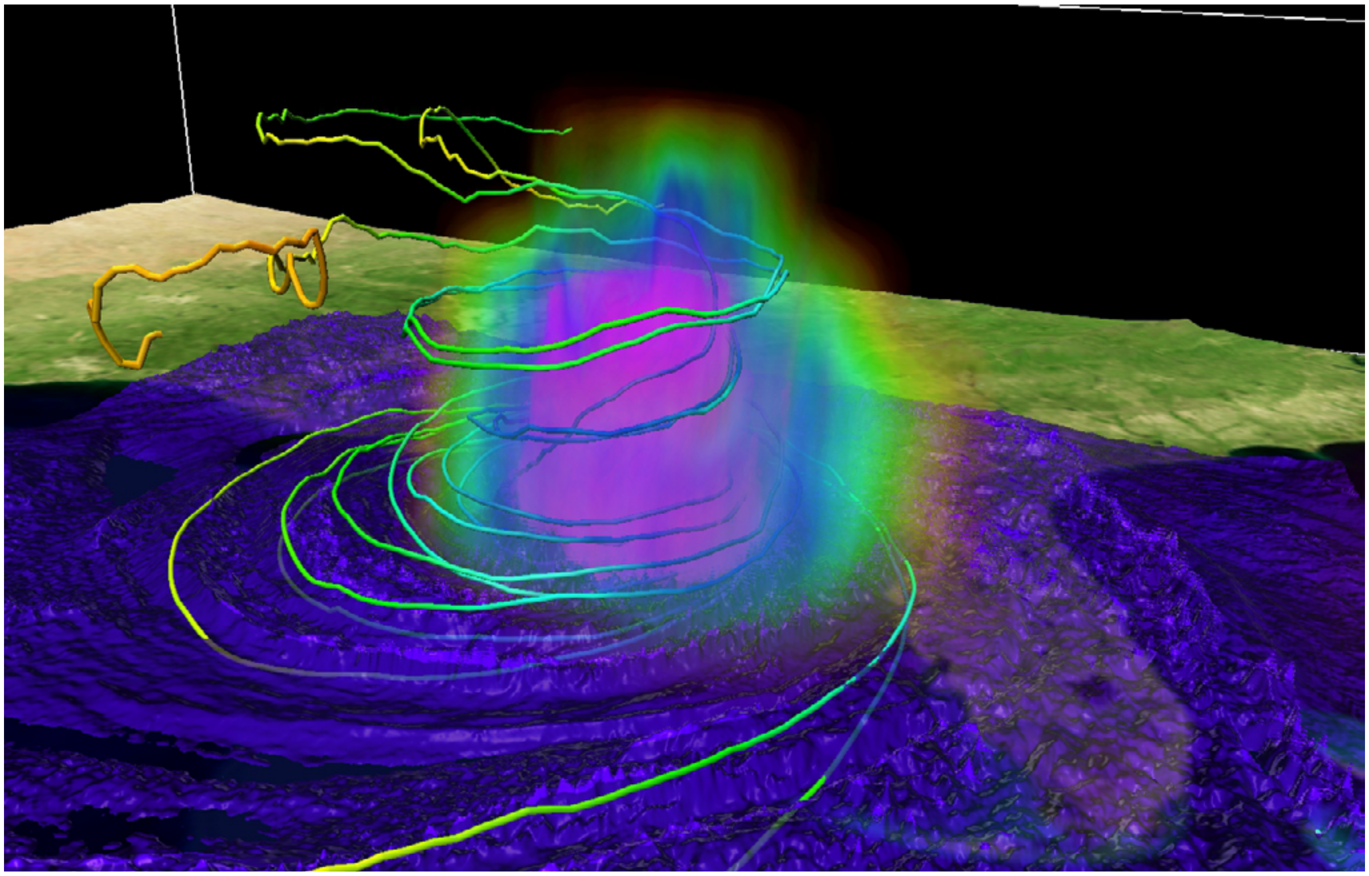
4 Unreliable datagram mode in the Intel® MPI Library

Therefore, we have used Intel MPI Library's DAPL UD mode to achieve greater scalability on Yellowstone. The default DAPL UD values were modified to further reduce the memory usage and increase the timeout settings (see **Appendix A**). We used DAPL version 2.0.36.

Here, we demonstrated the efficient parallel implementation of WRF, the high-performance scalability of the Intel® MPI Library and associated Intel compilers and analysis tools, and the scalability of the Intel® Xeon® processor-based Yellowstone system.

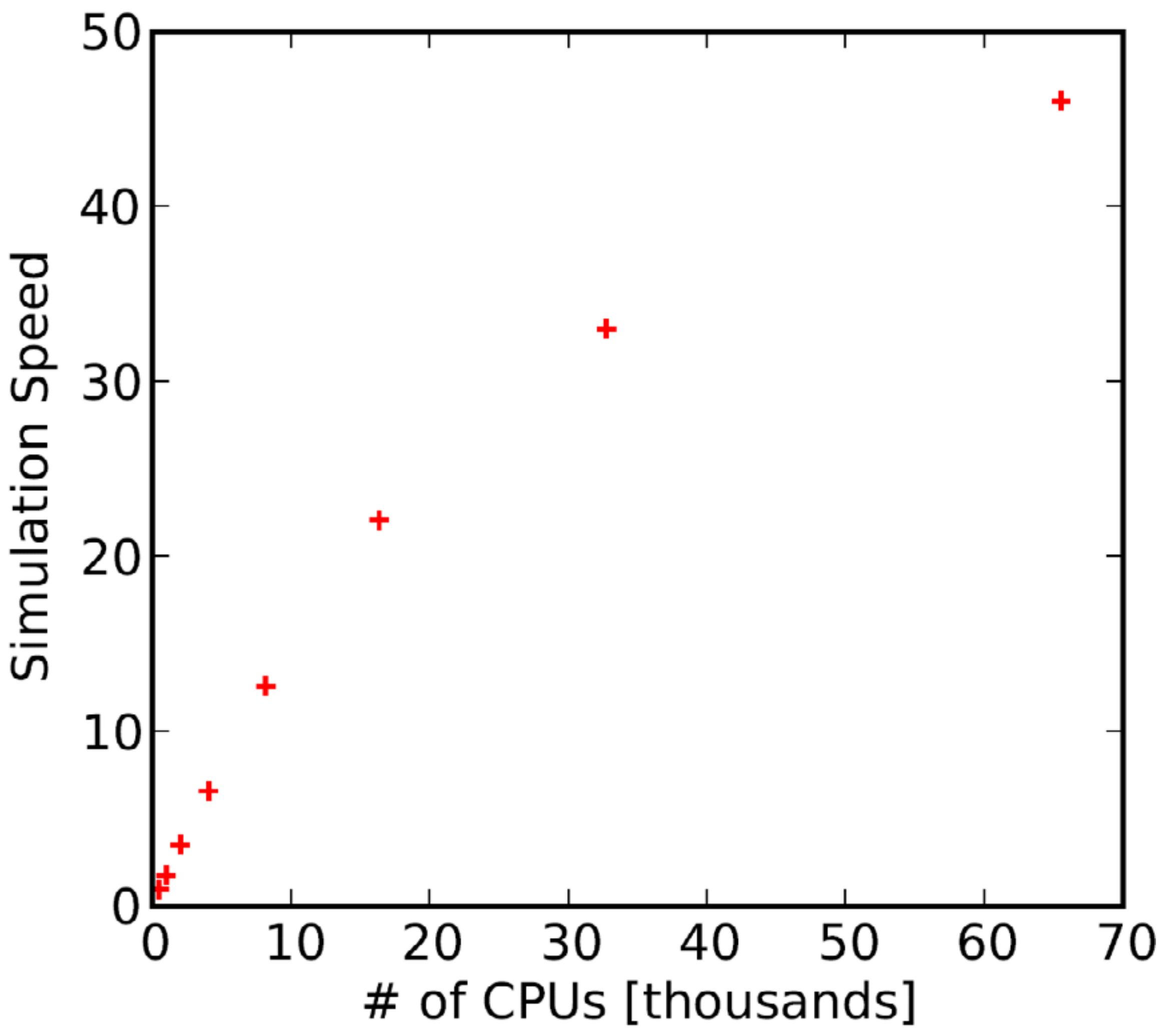
Scaling Results

The atmosphere simulation of Hurricane Katrina at 1km resolution was run with 35 vertical levels and a 3-second time step, involving a total of 371,227,850 grid points (3,665 x 2,894 x 35). **Figure 5** illustrates some of the results obtained during our runs.



5 Visualization of simulated Hurricane Katrina at 1km resolution on Aug 28, 2005, at 0000 UTC, 11 hours before making landfall on the Gulf Coast. Wind speed and flow are visualized by the colorful volume rendering and 3D streamlines. The colorful volume in the middle of the hurricane represents the wind speed, with increasing opacity with higher wind speeds (up to 89 miles/hour shown). The water vapor mixing ratio is visualized as a blue isosurface at 11.2 g/kg. The visualization was produced by using the Visualization and Analysis Platform for Ocean, Atmosphere, and Solar Researchers (VAPOR).

To evaluate scalability, we measured the simulation speed for replicate runs at different core counts. We define simulation speed for a run as the ratio of simulated time to wall clock time, not including time spent for disk I/O, averaged over the number of time steps in the run (e.g., a speed of 48 corresponds to needing 1 hour of simulation to forecast 48 hours of weather). Four runs were conducted at core counts of 16K and less. Three benchmark runs were conducted at 32K cores, and a single run at 64K cores. **Figure 6** shows the best simulation speeds for our 1km Katrina workload up to 65,536 cores on Yellowstone.



6 Scaling assessment of the Katrina 1km workload on Yellowstone cluster with up to 64K cores. WRF version 3.5 was built using Intel® Composer XE 2013, Intel® MPI Library 4.1, netCDF v.4.2 with the following compiler options: "-O3 -align all -fno-alias -fp-model precise". DAPL UD settings are described in Appendix A.

Run-to-run variability was small using the Intel MPI library, with relative standard deviations (standard deviation/mean) in simulation speed near 3.5% for the 16K and 32K runs and less than 0.2% for all CPU counts of 8K and less. In early runs, larger variability was observed. Those issues were attributed to defective InfiniBand cables, suboptimal IB network routing, and occasional contention due to other jobs running on the machine. In one instance, we found that a single "bad" node—unidentified by the system health checks—decreased performance by about 50% for all runs that happened to include it. All these issues were particularly severe during the first months after the opening of the NWSC, but improved significantly over time.

The Katrina 1km workload scales approximately linearly through 16K cores. Between 16K and 64K cores, the scaling ceases to be linear, but appreciable increases in simulation speed were still observed. Scaling remains linear as long as each core has more than approximately 500 grid points to compute. When the computational granularity becomes finer, the fraction of grid points involved in halo exchanges for each MPI task is no longer negligible.

Conclusion

Only a few days passed from when Hurricane Katrina entered the Gulf of Mexico to when it made landfall in Louisiana. Safely responding to extreme weather events like this requires the earliest possible advance warning to the impacted areas, and thus very fast and accurate simulation capabilities. The ability to perform larger and more detailed weather forecasts depends on effective use of large-scale computational resources, which are becoming increasingly common even at relatively small supercomputing centers. Here, we demonstrated the efficient parallel implementation of WRF, the high-performance scalability of the Intel MPI Library and associated Intel compilers and analysis tools, and the scalability of the Intel Xeon processor-based Yellowstone system.

Acknowledgments

The authors are very grateful to C. M. Patricola, P. Chang, and R. Saravanan (Texas A&M University) for kindly providing the Katrina workload. M. Lubin and S. McMillan would like to acknowledge support of their Intel colleagues Gergana Slavova and Arlin Davis. Additionally, R. Montuoro would like to acknowledge support from the Office of Science (BER), U.S. Department of Energy, through Grant N. DE-SC0006824, and C. Kruse would like to acknowledge support through the Summer Internships in Parallel Computational Science program from Rich Loft, NCAR, and the NSF. ●

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

Appendix A

Intel® MPI DAPL UD library runtime parameters used with high core counts runs.

I_MPI_FABRICS	shm:dapl
I_MPI_DAPL_UD_PROVIDER	ofa-v2-mlx4_0-1u
I_MPI_DAPL_UD	on
I_MPI_DAPL_UD_RNDV_EP_NUM	2
I_MPI_DAPL_UD_DIRECT_COPY_THRESHOLD	65536
I_MPI_DAPL_UD_SEND_BUFFER_NUM	8208
I_MPI_DAPL_UD_RECV_BUFFER_NUM	8208
I_MPI_DAPL_UD_ACK_RECV_POOL_SIZE	8704
I_MPI_DAPL_UD_CONN_EVD_SIZE	2048
I_MPI_DAPL_UD_REQUEST_QUEUE_SIZE	80
DAPL_UCM_REP_TIME	16000
DAPL_UCM_RTU_TIME	8000
DAPL_UCM_RETRY	10
DAPL_UCM_QP_SIZE	8000
DAPL_UCM_CQ_SIZE	8000
DAPL_UCM_TX_BURST	100
DAPL_MAX_INLINE	64
DAPL_ACK_RETRY	10
DAPL_ACK_TIMER	20
DAPL_RNR_TIMER	12
DAPL_RNR_RETRY	10

For more information regarding performance and optimization choices in Intel® software products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

Sign up for future issues | Share with a friend

References

1. Intel® Composer XE 2013. **http://software.intel.com/en-us/intel-composer-xe**.
2. Intel® MPI Library. **http://www.intel.com/go/mpi**.
3. Michalakes, J., et al., 2008: WRF nature run. J. of Physics: Conference Series, 125, doi: 10.1088/1742-6596/125/1/012022.
4. NCAR, 2013: NCAR-Wyoming Super Computer. **http://www2.cisl.ucar.edu/resources/yellowstone**.
5. Patricola, C. M., P. Chang, R. Saravanan, and R. Montuoro, 2012: The Effect of Atmosphere-Ocean-Wave Interactions and Model Resolution on Hurricane Katrina Coupled Regional Climate Model. Geophys. Res. Abs., 14, **http://meetingorganizer.copernicus.org/EGU2012/EGU2012-11855.pdf**.
6. Skamarock, W. C., et al., 2008: A description of the advanced research wrf version 3. Tech. rep., National Center for Atmospheric Research. NCAR Tech. Note NCAR/TN-4751STR, 113 pp. Available online at **http://www.mmm.ucar.edu/wrf/users/docs/arw_v3.pdf**.
7. Research-Community Priorities For WRF-System Development. Prepared by the WRF Research Applications Board, December 2006.

See the impact.

30-day trial software

Evaluate the latest Intel® Parallel Studio XE and Intel® Cluster Studio XE >

Evaluate Intel® System Studio for embedded and mobile system developers >

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

BLOG HIGHLIGHTS

Avoiding Potential Problems—Memory Limits on the Intel® Xeon Phi™ Coprocessor

BY FRANCES ROTH (INTEL) »

Exceeding Memory Space

As with any Linux* system, the operating system on the coprocessor will sometimes allow you to allocate more memory space than is physically available. On many systems, this is taken care of by swapping some pages of memory out to disk. On those systems, if the amount of physical memory plus the amount of swap space is exceeded, the operating system will begin killing off jobs.

The situation on the Intel® Xeon Phi™ coprocessor is complicated by the lack of directly attached disks. So:

> There is, currently, at most, 8GB of physical memory available (I say this, having come from a generation where my first personal computer had 4KB of memory)

> By default, some of the memory is used for the coprocessor's file system

> By default, there is no swap space available for the coprocessor

Limiting Memory Used for RAM Disk

The Intel Xeon Phi coprocessor does not have any directly accessible disk drives. Because of this, the default root file system for the coprocessor is stored on a RAM disk. This not only affects file storage, but also reduces the memory available to running programs.

The size of the root file system is kept small by using BusyBox* to replace many of the common Linux commands, such as sh, cp and ls, and by limiting the number of shared libraries that are copied to the root file system. Even given these reductions, about 10 MBs of memory are still consumed for file storage.

More >

Software

The Parallel Universe

© 2013, Intel Corporation. All rights reserved. Intel, the Intel logo, Intel Core, Intel Inside, Cilk Plus, Pentium, VTune, VPro, Xeon and Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.