

THE PARALLEL UNIVERSE

Issue 11
September 2012



Intel® Parallel Studio XE 2013

10 Feature Highlights for Accelerated Performance

by James Reinders

The advertisement features a man with arms crossed on the right. On the left, a circular diagram illustrates the development lifecycle with four stages: Design, Build, Verify, and Tune. In the center of the circle is the Intel Parallel Studio XE 2013 software box. The background is a blue gradient with faint, stylized code snippets like `LayoutKind.Sequential` and `Height*3/4`.

CodeBook
Intel® Parallel Studio XE 2013

intel
Software

Boost performance and accuracy

This downloadable CodeBook provides “how-to” guidance and a comprehensive resource toolkit to help you efficiently produce fast, scalable, reliable applications throughout the development lifecycle.

Look for guidance and techniques for C++ and Fortran developers:

Tools and techniques across
the development lifecycle

Features for accelerated performance

Technical guides, white papers,
articles, and blogs

And much more

DOWNLOAD THE FREE CODEBOOK NOW



CONTENTS

Letter from the Editor

Putting Intel® Parallel Studio XE 2013 to Work for the “New Normal,” BY JAMES REINDERS.....

4

Intel® Parallel Studio XE 2013: 10 Feature Highlights for Accelerated Performance,

BY JAMES REINDERS.....

6

Get up to speed fast on the components and new feature sets in the Intel® Parallel Studio XE 2013 suite—and consider the potential for your applications.

Using Intel® Software Development Tools to Analyze HMMER, BY WALTER SHANDS.....

8

Explore techniques for developing applications like HMMER for the latest generation of multicore processors—from thread and memory error checking to performance and code optimization.

Pointer Checker: Easily Catch Out-of-Bounds Memory Access, BY KITTUR GANESH.....

20

Pointer Checker is designed to catch any out-of-bounds memory accesses before memory corruption occurs. Find out how to use Pointer Checker effectively, and to balance the trade-offs of security and runtime.

New Parallel Programming Features in Intel® (Visual) Fortran Composer XE,

BY STEVE LIONEL.....

22

This overview of two new features, DO CONCURRENT and coarrays, brings insight into achieving excellent parallelism results with Fortran.

Using the Intel® Math Kernel Library and Intel® Compiler to Obtain Run-to-Run Numerical Reproducible Results,

BY TODD ROSENQUIST AND SHANE STORY.....

26

How do you balance demands for accelerated performance with reproducible results and runtime consistency? These techniques can help you generate reproducible results within applications under a manageable set of constraints.

Sign up for future issues | Share with a friend

The Parallel Universe is a free quarterly magazine. [Click here](#) to sign up for future issue alerts and to share the magazine with friends.





James Reinders explores the development capabilities of the mature parallelism tool suite, Intel® Parallel Studio XE 2013.

LETTER FROM THE EDITOR

PUTTING INTEL® PARALLEL STUDIO XE 2013 TO WORK FOR THE “NEW NORMAL”

Has parallelism changed everything or nothing?

On one hand, parallelism is everywhere and parallel programming is the “new normal.” On the other hand, writing, debugging, and tuning an application remains our work to do as programmers. Did we just raise the bar to create the new normal? Perhaps. We expect more from our programming and, in turn, we need more from our tools.

At Intel, we have invested heavily to support this new normal with a wealth of new capabilities in the latest edition of Intel® Parallel Studio XE.

The launch of Intel® Parallel Studio XE 2013 updates a mature toolset for application development with support we need for the new normal—spanning many aspects of software development.

In this issue, we look at some of the top new features and capabilities of the Intel Parallel Studio XE 2013 product. In the 10 Feature Highlights article, we highlight efforts which are significant new capabilities in their own right. Each could have a whole issue dedicated to it filled with interesting examples and tales on how they work.

We’ve selected three to dive into in this issue beyond the information in the 10 Feature Highlights article. One covers “Pointer Checker,” and one discusses Fortran capabilities. Another article covers a new run-to-run (and processor-to-processor) numerical reproducible results capability. This capability helps deal with the inherently non-associative nature of floating point numeric representations with a new unequaled set of options in the latest Intel Parallel Studio XE.

We also have a real-world usage case covered in Using Intel® Software Development Tools to Analyze HMMER. This study makes use of event-based sampling analysis in Intel® VTune™ Amplifier XE and the optimization features of the Intel® Composer XE compiler to build and analyze hmmsearch and hmmbuild, components of SPECint*.

I think you’ll find this issue full of exciting new capabilities. Maybe the end result is a “new normal”—but it is an exciting new place to be.

James Reinders

Director of Parallel Programming Evangelism at Intel Corporation. James is a co-author of a new book *Structured Parallel Programming* from Morgan Kaufmann, 2012. His other books include *Intel® Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism*, available in English, Japanese, Chinese, and Korean.

Intel® Parallel Studio XE 2013:

10 Feature Highlights for Accelerated Performance

by James Reinders, *Director of Parallel Programming Evangelism*

Intel® Parallel Studio XE 2013 not only delivers the latest optimizations and new processor support, but it also includes a number of highly innovative features that are likely to surprise and delight you.

The suite plugs seamlessly into Microsoft Visual Studio* and the GNU toolchain, thereby preserving investments in your development environment of choice.

With Intel Parallel Studio XE 2013, accelerated application performance is often just a recompilation away. Rebuild with the latest compilers and link in the latest libraries to benefit from the latest processors.

I have chosen 10 features to highlight from this powerful Intel tool suite.

1. Processor Support Updated to Include the Latest Intel® Processors

New support includes AVX2, TSX, and FMA3. This extends our support to both the newly released 3rd Generation Intel® Core™ vPro™ processor (codenamed Ivy Bridge) microarchitecture, as well as the forthcoming Haswell microarchitecture. This enables you to take advantage of the latest performance enhancements in the newest Intel® products, while preserving compatibility with prior Intel and compatible processors.

2. Support for Intel® Many Integrated Core (Intel® MIC) Architecture

Used for more than a year on prototype and preproduction systems, support for Intel® MIC architecture is now available in our products. No additional new tools are needed for the first Intel® Xeon Phi™ coprocessor (codenamed Knights Corner). Instead, we have integrated this support in tools you already know and use. The power of these familiar tools is now available to help generate, debug, and optimize code for the Intel® MIC architecture.

3. Advanced Numerical Reproducibility Capabilities

The most praised new feature by beta testers. An innovative new “Conditional Numerical Reproducibility” capability offers unique controls over nonassociative floating-point operations, allowing run-to-run and processor-to-processor reproducibility options—often with very low performance penalties. Increased options for floating-point arithmetic reproducibility with Intel® Math Kernel Library, special Intel support in OpenMP*, and new capabilities in Intel® Threading Building Blocks open up new possibilities.

4. Additional Profiling Data and Easier to Use

Intel® VTune™ Amplifier XE offers new and powerful bandwidth and memory access analysis to reduce time spent puzzling over cryptic performance data.

5. Pointer Checker

A new compiler-based diagnostic tool allows you to find code that accesses memory addresses beyond the allocated addresses. This helps with security hardening and finding difficult memory corruption issues.

6. New Threading Assistant: Intel® Advisor XE

Intel® Advisor XE assists in producing scalable, maintainable C, C++, C#, and Fortran code. Simplifies adding parallelism to threaded or unthreaded applications, and allows you to evaluate alternatives before investing in implementation.

7. Fortran Standards Support

Intel® Fortran supports widely used features of the Fortran 2003 standard and key parts of the 2008 standard, including coarrays. As a leader, Intel is committed to supporting Fortran with our products. Of course, we maintain a rich backward compatibility with decades of Fortran support including VAX Fortran*, Compaq Visual Fortran*, Fortran 95, Fortran 90, Fortran 77, and Fortran 66, as well as library support for BLAS, LAPACK, ScalAPACK, sparse solvers, fast Fourier transforms, vector math, and more.

8. C++ Performance Guide

Everyone can appreciate the new C++ Performance Guide, featuring a quick five-step process for increasing performance.

9. C and C++ Standards Support

Outstanding support for C and C++ are now accompanied by leading support for many of the new C++11 and C11 features. We also maintain our extensive support for prior standards including C99, and industry-leading support for IEEE 754-2008 Decimal Floating-Point Arithmetic.

10. Find and Eliminate Errors with Intel® Inspector XE

Intel® Inspector XE provides an efficient way to increase your application reliability to ensure performance in C, C++, C#, and Fortran. The new heap growth analysis feature offers an important new way to find memory leaks.

**DOWNLOAD A FREE
30-DAY EVALUATION**



The power of this suite stems from four key components:

1. Optimized C++ and Fortran Compilers and Libraries:

Intel® Composer XE is a highly optimizing performance-oriented developer tool that includes Intel® C++ and Fortran compilers, and threading, math, multimedia, and signal processing performance libraries. Intel® Cilk™ Plus, Intel® Threading Building Blocks, and OpenMP* support provide parallelism models to make it easier to take advantage of today's and tomorrow's high-performance computing systems. Industry-leading Intel® Math Kernel Library and Intel® Integrated Performance Primitives include a wealth of routines to improve performance and reduce development time.


2. Innovative Threading Assistant for Linux* and Windows*:

Intel® Advisor XE is a threading assistant for C, C++, C#, and Fortran developers. It helps find regions with the greatest performance potential from parallelism and highlights critical synchronization issues. With Advisor XE, you can evaluate alternatives before investing in implementation, estimate the speed-up, identify correctness issues and select the options with the best return on investment. The "magic" here is in the ability to evaluate approaches before committing to coding and debugging. This is a remarkable tool when considering how to add parallelism into your code.

3. Optimize Serial and Parallel Performance: Intel® VTune™

Amplifier XE is the premier performance and thread profiler to tune application performance. Use it to profile C, C++, C#, Fortran, assembly code, and Java code, and receive rich performance data for hotspots, threading, locks and waits, DirectX*, bandwidth, and more.

4. Deliver More Reliable Applications: Intel® Inspector XE 2013

is an easy-to-use memory and threading error detector for serial and parallel applications on Windows* and Linux*. Static analysis for C, C++, and Fortran developers is included in Intel® Studio XE products. The ability to pinpoint active and latent problems before shipping an application to customers is strongly supported by this acclaimed and unique Intel capability. 

TO LEARN MORE, VISIT
INTEL PARALLEL STUDIO XE



TECHNICAL SPECIFICATIONS AT A GLANCE

Processor Support	Validated for use with multiple generations of Intel and compatible processors including, but not limited to: Intel® Xeon® processors, Intel® Core™ processors, and Intel® Xeon Phi™ coprocessors.
Operating Systems	Windows* and Linux*. Compiler and library components are also available as Apple OS* X add-ons for Apple's XCode* development environment.
Development Tools and Environments	Compatible with compilers from vendors that follow platform standards (e.g., Microsoft, GNU, Intel). Can be integrated with GNU toolchain*, Microsoft Visual Studio* 2008, and 2010, and next-generation tools.
Programming Languages	Extensive support for C, C++ and Fortran development. Additional support included for programs that also include Java or .NET languages such as C#.
Support	All product updates, Intel® Premier Support services, and Intel® Support Forums are included for one year. Intel Premier Support gives you access to confidential support, technical notes, application notes, and the latest documentation.
Community	Join the Intel® Support Forums community to learn, contribute, or just browse: http://software.intel.com/en-us/forums
System Requirements	For details on hardware and software requirements: www.intel.com/software/products/systemrequirements/





by Walter Shands,
*Software Development
Engineer*

Using Intel® Software Development Tools to **Analyze HMMER**

This paper will highlight the features of Intel® Parallel Studio XE 2013 by using them to build and analyze HMMER (<http://hmmerr.janelia.org/>). HMMER is a set of applications, which includes two, `hmmsearch` and `hmmbuild`, which are components of SPECint. We make use of event-based sampling analysis in Intel® VTune™ Amplifier XE to find out which code paths, context switches, or threading inactivity cause performance problems in `hmmsearch`. And, we'll utilize the code optimization features of the Intel® Composer XE compiler to improve the performance of `hmmsearch` on Intel® Xeon® E5 processors. In addition, we will show you how to use Intel® Inspector XE to locate memory and threading errors introduced into `hmmsearch`.

hmmsearch is used to search a protein sequence database for homologs of protein sequences using profiles called hidden Markov models. `globins4.hmm` contains the profiles and `uniprot_trembl.fasta` is a 10 GB sequence database.

`hmmsearch` is available in an MPI version, but we restricted our experiments to the non-MPI flavor. We ran `hmmsearch` on a computer with an 8-core Intel® Xeon® E5-2680 hyperthreaded processor at 2.7 GHz with 23.4 GB of memory. We ran the application using GCC and the Intel® C compiler, in both cases using the settings provided by the `configure` script. The initial GCC default switches were:

```
gcc -std=gnu99 -O3 -fomit-frame-pointer -malign-double -fstrict-aliasing -pthread -msse2
```

The application requires support for the SSE2 instruction set at a minimum to support an algorithm optimized using intrinsics oriented toward SSE2.

The default Intel® compiler flags were:

```
icc -O3 -ansi_alias -pthread
```

A challenge in porting applications from one compiler to another is making sure that there is support for the compiler options you use to build your application. The Intel C compiler supports many of the options that are valid on other compilers you may be using, such as GCC. The compiler generates object files that are compatible with GCC-generated object files, so you can compile part of your application using the Intel compiler and the rest using GCC.

The `-fomit-frame-pointer` option is set when you specify option `-O1`, `-O2`, or `-O3` when using the Intel C compiler (so there is no need to include it). The `-malign-double` option aligns double, long-double, and long-long types for better performance for systems based on IA-32 architecture and is available in the Intel C compiler.

We started the application with this command line:

```
./hmmsearch globins4.hmm ../../uniprot_trembl.fasta
```

The next step is to locate the hotspots in the application using Intel VTune Amplifier XE. This profiler tool uses low overhead techniques to quickly find multicore performance bottlenecks, without needing to know the processor architecture or assembly code. Note that we do not need to add code to the application to collect data.

To view source code lines of `hmmsearch` in VTune Amplifier XE, we need to include symbols in the release build—so we add the `-g` flag. We added the `-fno-inline-functions` flag as well; this allows us to see all of the code in question in the VTune Amplifier XE source view.

The VTune Analyzer XE hotspots analysis shows where most of the CPU activity is occurring in the application and the amount of CPU activity on the threads over time. (Figure 1)

The VTune Amplifier XE hotspots view tells us that the function consuming the most CPU time is `p7_MSVFilter`, and double-clicking on the function name displays the SSE intrinsics calls used in optimizing the performance of the function. The assembly view shows us that the Intel compiler utilized vector instructions, but is not taking advantage of the 256-bit registers or AVX instructions on the Intel Xeon processor. (Figure 2)

It's possible that we could compile the original C code for `p7_MSVFilter` with the Intel compiler and help the compiler vectorize the function for the instruction set available on the target machine, so that the function is not limited to using 128 bit registers.

The thread timeline view shows that there is not much CPU time used in the worker threads, but a large amount is used in one thread. This turns out to be the thread that is reading the sequence database file. (Figure 3)

“To achieve more significant performance gains, the problem of serialization of the application due to the file read has to be solved.”

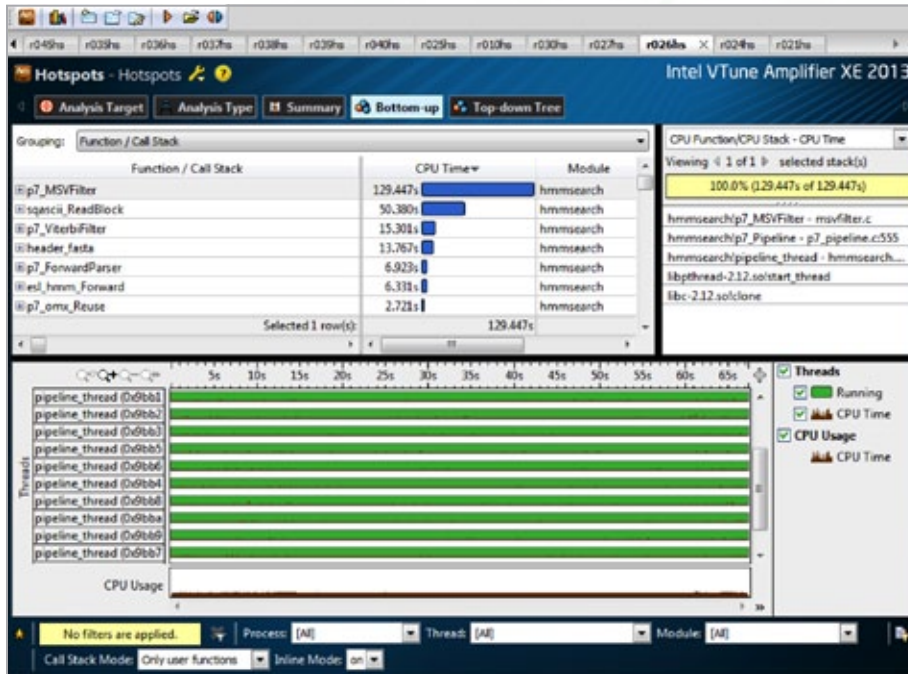


Figure 1

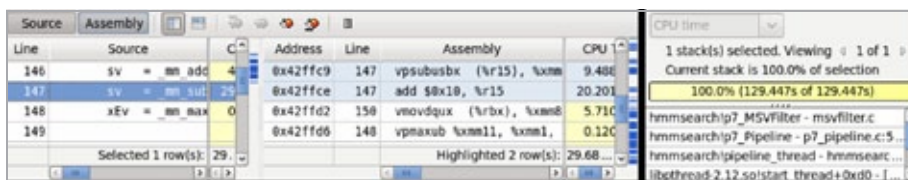


Figure 2

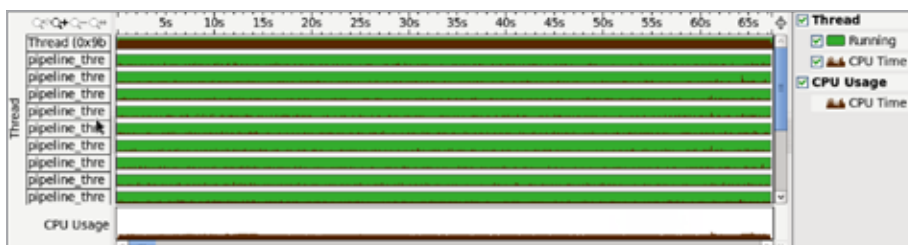


Figure 3

The application creates a number of threads equal to the number of HW threads on the machine plus one, which in the case of a hyperthreaded machine is equal to the number of hyperthreads plus one. In this case, there are 17 threads running. If we use the `hmmsearch -cpu 4` flag to limit the threads to five threads, VTune Amplifier XE shows that the application scales well—unlike the situation with 17 threads. (Figure 4)

Evidence of this is the 67.418-second runtime with 17 threads, which is worse than the 62.561-second runtime with four threads.

We can see that the top thread is the one reading the data file by filtering the results by thread in the five-thread hotspot display. (Figure 5)

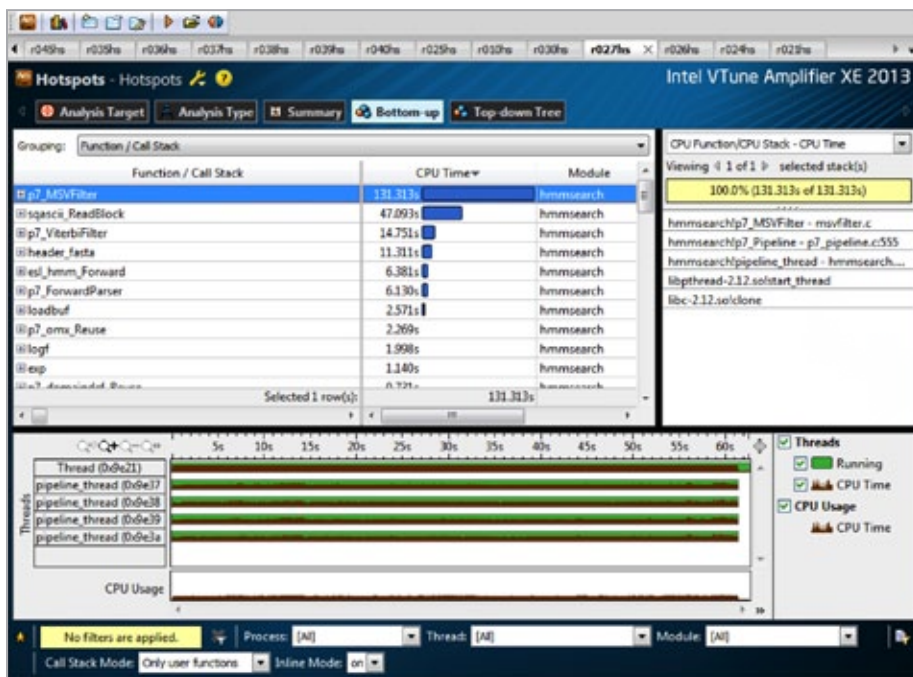


Figure 4

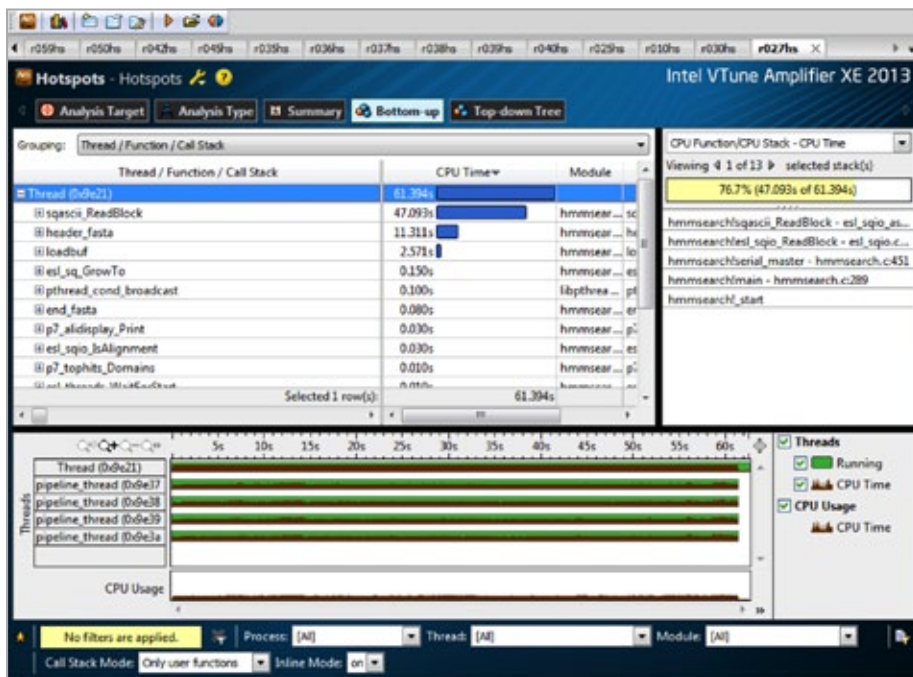


Figure 5

If we use the VTune Amplifier XE Locks and Waits feature on the run with 17 threads it shows us a large number of transitions, indicated by yellow lines from the thread reading the sequence database file to worker threads. (Figure 6)

hmmsearch uses a producer consumer model. This is where a producer thread (labeled Thread (0xa0) in the graphic) puts data to be processed on a queue that worker threads (labeled pipeline_thread in the graphic) remove when the producer thread signals them with a broadcast message, resulting in a thread transition from the producer thread to the worker thread.

By zooming in, we can see that the amount of thread running time (dark green) is less than thread waiting time (light green), indicating lost time to do productive work. (Figure 7)

Compare this with a zoom-in on the thread view for hmmsearch using only four threads. Note that thread transitions from the thread reading the data file, the top thread, typically result in productive work to the worker thread. (Figure 8)

However, when using 17 threads in hmmsearch, many thread transitions do not result in work being done. (Figure 9)

Zooming in even closer on the 17 thread case, we can see these thread transitions are the result of a pthread_cond_broadcast call that tells the worker threads that a block of data is ready on the work queue to be processed. Only one thread at a time can grab the block of data—so the other threads must wait again. (Figure 10)

When only five threads are used, only about two threads are waiting to get a block of data to process, and only one thread goes unsatisfied. (Figure 11)

All of this indicates that with more than four threads, the hmmsearch pipeline threads become starved for data. In other words, the thread reading the data file cannot provide data fast enough to keep up with computation in the worker threads.

From our analysis using VTune Amplifier XE, we know that the most time-consuming code is the MSV algorithm, which has been optimized with SSE intrinsics in p7_MSVFilter in the file msvfilter.c. The intrinsic-optimized code also contains some optimizations over and above vectorization, so it will be faster.

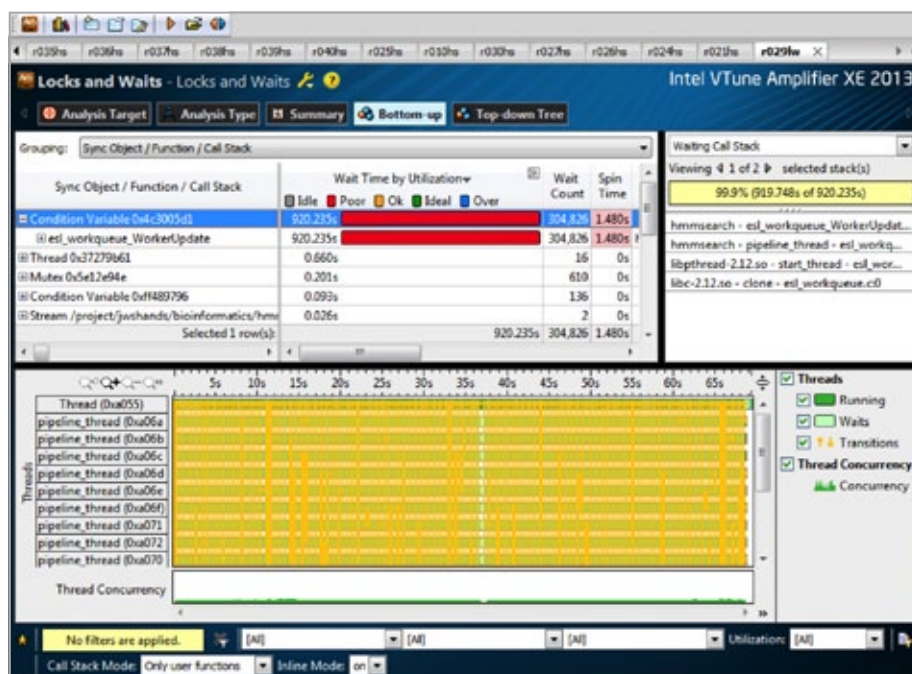


Figure 6

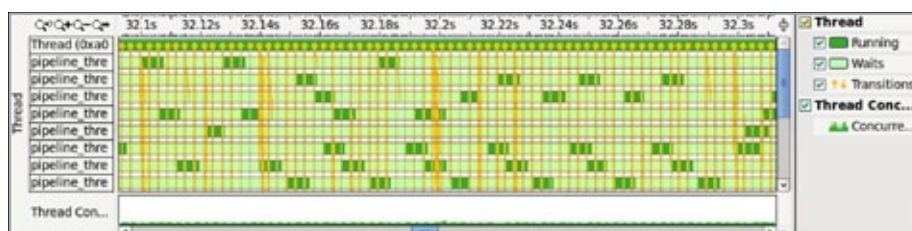


Figure 7

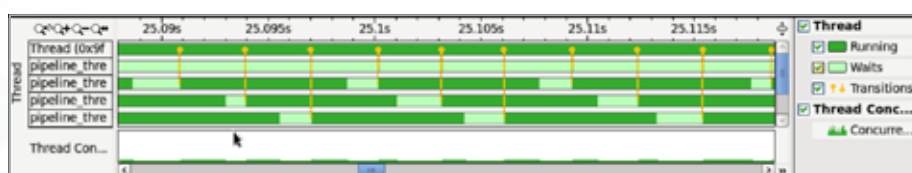


Figure 8

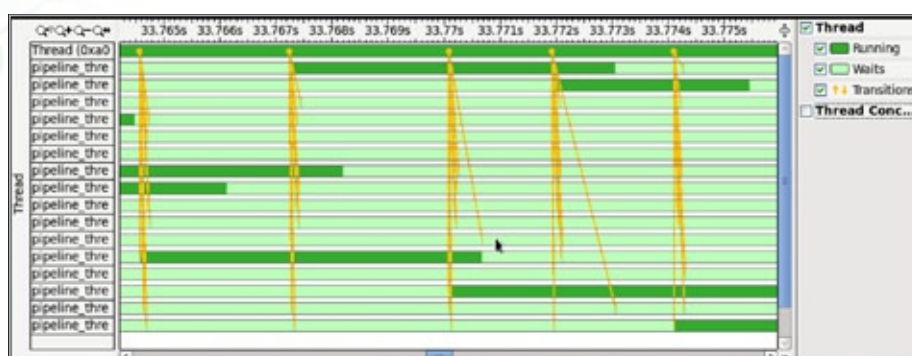


Figure 9

To see if the Intel compiler can effectively vectorize the nonintrinsic optimized code, we compiled the application to use the unoptimized C code in the function `p7_GMSV` in the file `generic_msv.c`. VTune Amplifier XE again shows that the MSV algorithm is the hotspot. (Figure 12)

VTune Amplifier XE also shows that the most time-consuming part of the MSV algorithm is a single loop that is not taking advantage of AVX instructions or YMM registers on the Intel Xeon processor.

(Figure 13)

The runtime of `hmmsearch` using this code is about four minutes and 30 seconds.

```
# CPU time: 4137.39u 5.02s
01:09:02.41 Elapsed: 00:04:30.08
```

If we use the `-opt-report` flag for the Intel compiler, it will tell us what inlining, loop, memory, vectorization, and parallelization optimizations have been done for each function. For the `p7GMSV` function, it tells us the loop was not vectorized.

```
generic_msv.c(80:7-80:7):VEC:p7_GMSV: loop
was not vectorized: existence of vector
dependence
```

By restructuring the code, we can enable the compiler to vectorize the loop and generate code that takes advantage of Intel Xeon architecture. The optimization report from the compiler indicates that the two loops resulting from the restructuring were vectorized:

```
generic_msv.c(88:7-88:7):VEC:p7_
GMSV: LOOP WAS VECTORIZED

generic_msv.c(108:7-108:7):VEC:p7_
GMSV: LOOP WAS VECTORIZED
```

In addition, the VTune Amplifier XE assembly view shows that AVX instructions are being used along with the larger YMM registers. (Figure 14)

The resulting runtime of the application is close to half of the original runtime.

```
# CPU time: 2207.74u 4.96s
00:36:52.69 Elapsed: 00:02:28.16
```

We can use Intel Inspector XE to check `hmmsearch` for threading and memory errors. It gives detailed insight into application memory and threading behavior to improve application reliability, and its powerful thread checker and debugger make it easier to find latent errors on the executed code path. Intel Inspector XE also finds intermittent and nondeterministic errors, even if the error-causing timing scenario does not happen.

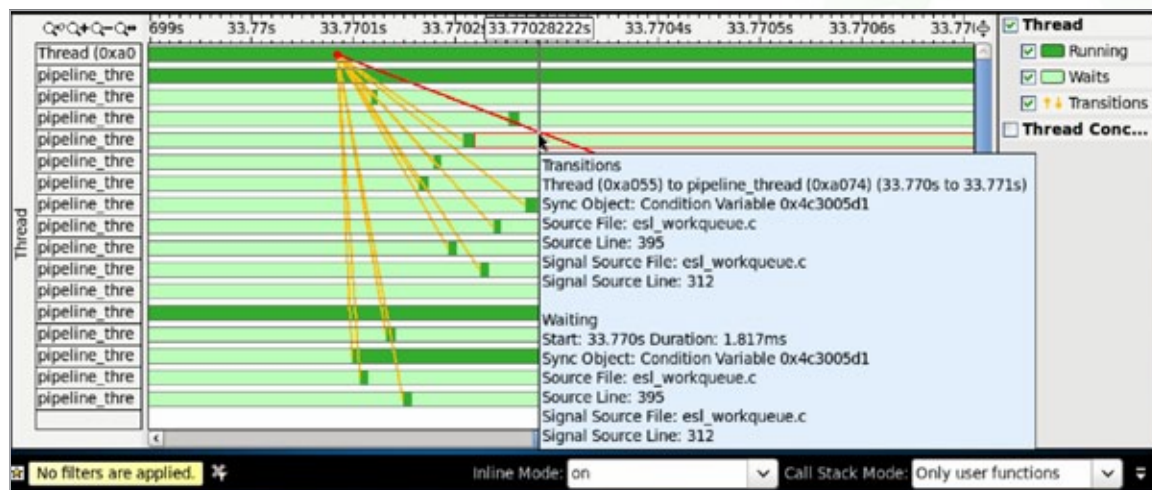
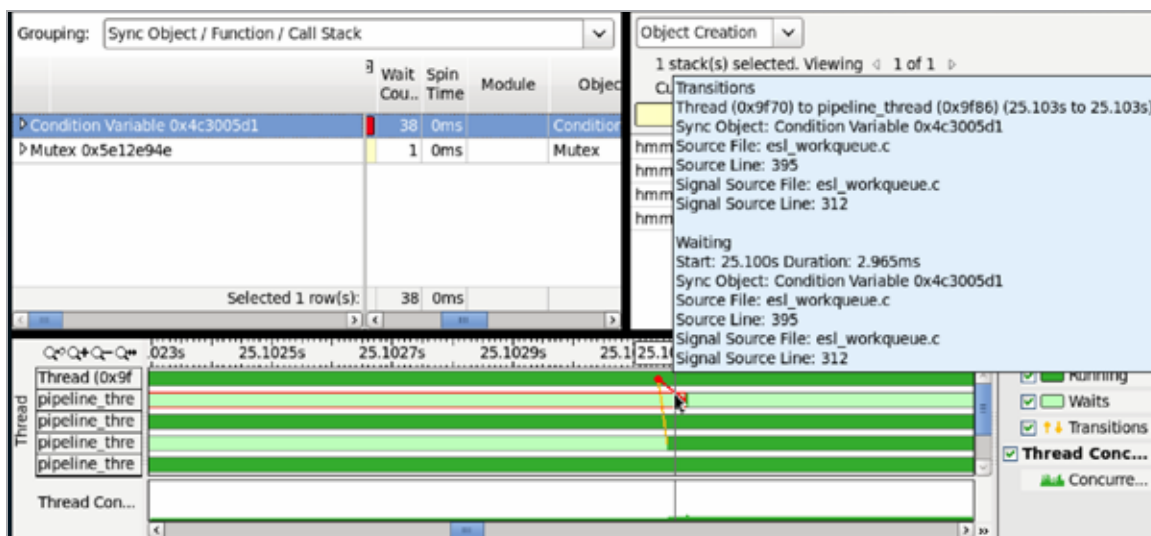
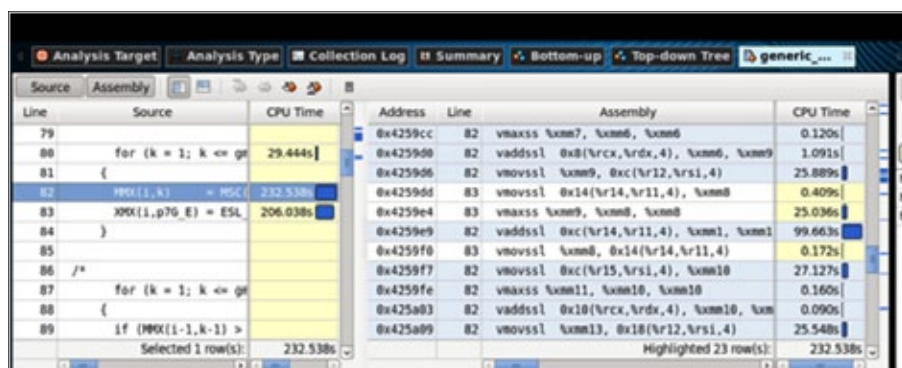
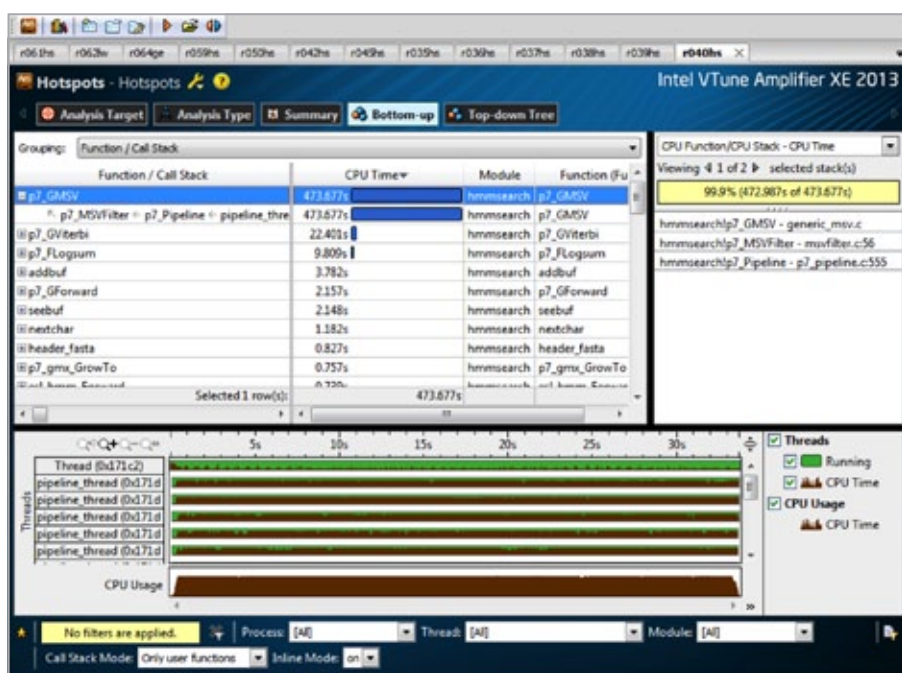


Figure 10



“The Intel® C compiler and libraries create faster code, Intel® VTune™ Amplifier XE finds bottlenecks, and Intel® Inspector XE pinpoints memory and threading errors before they happen. All this is of critical importance when developing applications like HMMER.”



Intel Inspector XE finds memory leaks, corruption, and inconsistent memory API usage, as well as data races, deadlocks, and memory accesses between threads.

As with Intel VTune Amplifier XE, we don't need to create a special build or add code to the application to collect data.

Because there is significant overhead in detecting memory and threading bugs, we launch `hmmsearch` using a smaller sequence database file, as well as an application option that reduces the number of threads.

When we run Intel Inspector XE in the Detect Memory Problems mode, a few uninitialized memory accesses are exposed. (Figure 15)

Right-clicking on a line in the Detect Memory Problems pane brings up a description of an uninitialized memory access problem: (Figure 16)

Intel Inspector XE running in Locate Deadlocks and Data Races mode did not detect any issues. (Figure 17)

In order to increase application performance, we can take advantage of Intel® Cilk™ Plus in the Intel compiler. Cilk Plus is an extension to C and C++ that offers a quick, easy, and reliable way to improve the performance of programs on multicore processors. It is an open standard and will soon be available in GCC 4.7. Cilk Plus, included in the Intel® C/C++ compiler, allows you to improve performance by adding parallelism to new or existing C or C++ programs using only three keywords: `cilk_for`, `cilk_spawn`, and `cilk_sync`.

Line	Source	CPU Time	Address	Line	Assembly	CPU Time
84	<code>XPK(i,p7_E) = ESL_MAX(XP</code>		0x425a56	93	<code>vinstrft128 \$0x1, %xmm11, %ymm10, %ymm</code>	4.845s
85	<code>);</code>		0x425a5c	93	<code>vaddps %ymm12, %ymm2, %ymm2</code>	0.680s
86	<code>*/</code>		0x425a61	91	<code>vextractf128 \$0x1, %ymm2, %xmm13</code>	8.612s
87			0x425a67	91	<code>vmovssl %xmm2, 0xc(%r14,%r12,1)</code>	4.520s
88	<code>for (k = 1; k <= gn->M; k</code>	9.126s	0x425a6e	91	<code>vextractpsl \$0x1, %xmm2, 0x18(%r14,%r</code>	0.500s
89	<code>{</code>		0x425a76	91	<code>vextractpsl \$0x2, %xmm2, 0x24(%r14,%r</code>	8.007s
90	<code>if (MPOX(i-1,k-1) > (XPK(i</code>	55.452s	0x425a7e	91	<code>vextractpsl \$0x3, %xmm2, 0x30(%r14,%r</code>	4.347s
91	<code>MPOX(i,k) = MPOC(k) + MPO</code>	109.441s	0x425a86	91	<code>vmovssl %xmm13, 0x3c(%r14,%r12,1)</code>	4.719s
92	<code>else</code>		0x425a8d	91	<code>vextractpsl \$0x1, %xmm13, 0x48(%r14,%</code>	5.185s
93	<code>MPOX(i,k) = MPOC(k) + XPK</code>	74.768s	0x425a95	91	<code>vextractpsl \$0x2, %xmm13, 0x54(%r14,%</code>	4.694s
94	<code>);</code>		0x425a9d	91	<code>vextractpsl \$0x3, %xmm13, 0x60(%r14,%</code>	4.716s

Figure 14

ID	Problem	Sources	Modules	Object Si...	State
P4	Uninitialized memory access	p7_trace.c	hmmse...		New
P5	Uninitialized partial memor...	esi_sq.c; p7_alidi...	hmmse...		New
P6	Uninitialized partial memor...	p7_oprofile.c	hmmse...		New
P7	Uninitialized partial memor...	p7_tophits.c	hmmse...		New

ID	Description	Source	Function	Module	Object Size	Offset
1730	Allocation site	esi_sq.c:1732	sq_init	hmmsea...		
1731	ESL_ALLOC(sq->name, sizeof(char) * sq->nalloc);					
1732	ESL_ALLOC(sq->acc, sizeof(char) * sq->nalloc);					
1733	ESL_ALLOC(sq->desc, sizeof(char) * sq->dalloc);					
1734	ESL_ALLOC(sq->source, sizeof(char) * sq->srcalloc);					

Figure 15

Uninitialized Partial Memory Access

Occurs when a read instruction references a block (2-bytes or more) of memory where part of the block is uninitialized.

1 Allocation site → 2 Read

ID	Code Location	Description
1	Allocation site	If present, represents the location and associated call stack from which the memory block containing the offending address was allocated.
2	Read	Represents the instruction and associated call stack responsible for the partial uninitialized access. If no allocation or deallocation is associated with this problem, the memory address might be in stack space.

Example

```

struct person
{
    unsigned char age;
    char firstInitial;
    char middleInitial;
    char lastInitial;
};
struct person *p1, *p2;
p1 = (struct person *) malloc(sizeof(struct person));
  
```

Figure 16



Figure 17

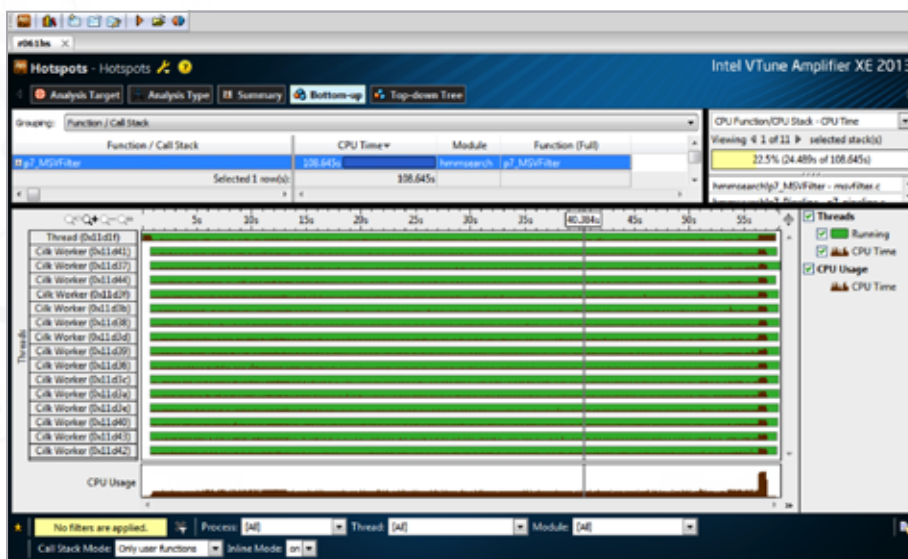


Figure 18

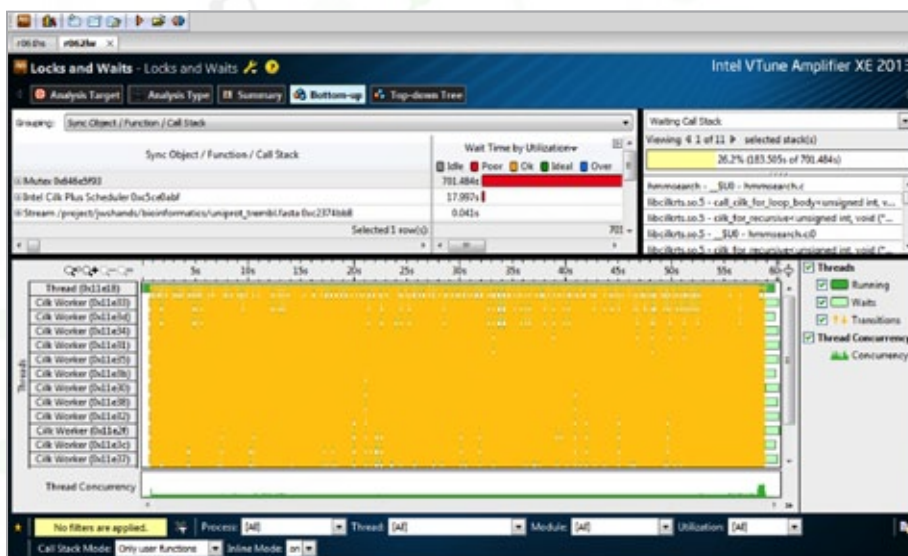


Figure 19

We use Cilk Plus to replace the code that manages threads, mutexes, condition variables, and the work queue with the added benefit of better scheduling. However, we must still synchronize threads on the data file read, which results in serializing a portion of the application.

In the Intel VTune Amplifier XE Hotspots graphic of an `hmmsearch` run, you can see that because of the synchronization resulting from mutexes around the code reading the sequence database file, the CPUs are not fully utilized. But the Cilk Plus implementation has a shorter runtime at 58.272 seconds compared to the original runtime of 67.418 seconds. (Figure 18)

If we run a VTune Amplifier XE locks and waits analysis we can see that there are still many thread transitions. (Figure 19)

If we zoom into the thread pane in the locks and waits analysis, we see that the thread transitions are between worker threads, and that they involve the mutex that protects the file read, which is now carried out by each worker thread. (Figure 20)

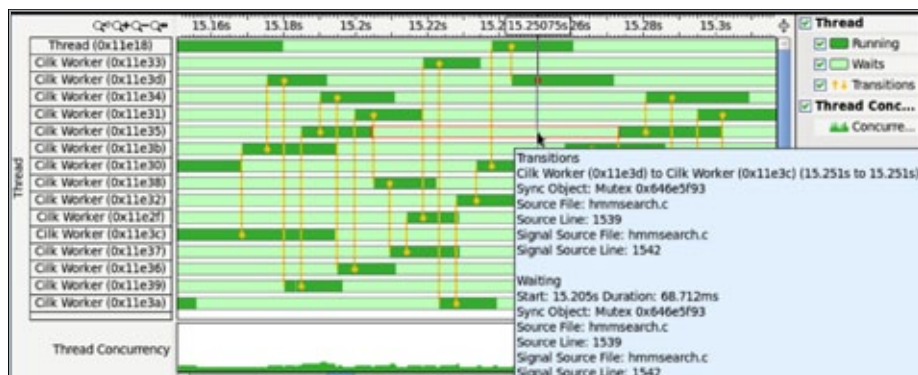


Figure 20

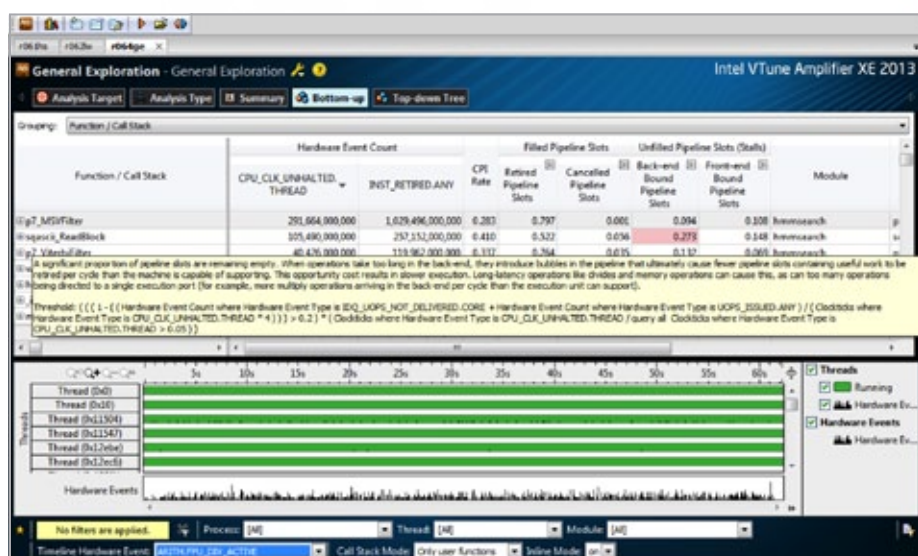


Figure 21

One of the other powerful features of Cilk Plus is the C/C++ language extension for array notations. This Intel-specific language extension provides data parallel array notations, which enable compiler parallelization and vectorization with less reliance on alias and dependence analysis.

To achieve more significant performance gains, the problem of serialization of the application due to the file read has to be solved. Reading the data into memory prior to computation is not realistic when using the `uniprot_trembl.fasta` data file, because we would exceed memory capacity on our machine, although if enough memory was available it would speed up subsequent computations using the same data.

Further performance gains can be achieved by taking advantage of Intel compiler options. Since the Intel compiler default instruction set is SSE2 and the target

machine is Intel Xeon, it would be a good idea to take advantage of AVX instructions and larger register size by using the `-xhost` switch that will generate an instruction set up to the highest level supported on the compilation host.

Another important compiler option is `-ipo`, which enables interprocedural optimization between files. This is also called multi-file interprocedural optimization (multifile IPO) or whole program optimization (WPO). When you specify this option, the compiler performs inline function expansion for calls to functions defined in separate files.

For help on finding out what to do to help the Intel compiler vectorize or parallelize loops we can use the `-guide` flag, which provides a report without producing objects or executables. The guided auto-parallelization feature of the Intel compiler is a tool that offers selective advice, resulting in better

performance of serially coded applications. The advice typically falls under three broad categories: source code modification, use of pragmas, and addition of compiler options.

Here is one of the suggestions after using the option in `hmmsearch`:

```
esl_vectorops.c(161):
remark #30536: (LOOP) Add
-fargument-noalias option
for better type-based
disambiguation analysis by
the compiler, if appropriate
(the option will apply for
the entire compilation).
This will improve
optimizations such as
vectorization for the loop
at line 161.
```

Adding the `-parallel` switch allows the Intel compiler to detect simply structured loops that may be executed in parallel, and automatically generates multithreaded code for them. If you use guided auto-parallelization options along with `-parallel`, the compiler may suggest advice on further parallelizing opportunities in your application:

```
msvfilter.c(106): remark #30525: (PAR)
Insert a "#pragma loop count min(1024)"
statement right before the loop at line
106 to parallelize the loop. [VERIFY]
Make sure that the loop has a minimum
of 1024 iterations.
```

We can also use the VTune Amplifier XE hardware event counter collection to get insight into bottlenecks in application code affecting performance. VTune Amplifier XE highlights collected data indicative of performance problems that should be investigated. Here is one example of an `hmmsearch` run. (Figure 21)

Conclusion

Intel® Software Development Tools help you boost application performance and increase the code quality, security, and reliability needed by high performance computing and enterprise applications. The Intel C compiler and libraries create faster code, Intel VTune Amplifier XE finds bottlenecks, and Intel Inspector XE pinpoints memory and threading errors before they happen. All this is of critical importance when developing applications like HMMER for the latest generation of multicore processors. □

Learn how Intel® Advisor XE can help improve parallelization productivity.

BY RAVI VEMURI

What do space exploration, oil and natural gas exploration, Hollywood movies, and military operations have in common? Modeling, simulation, exploration, storyboarding, and reconnaissance are some of the phrases that come to mind. They are intended to reduce the cost of wrong choices, failures, and missteps, and help projects succeed and be more productive.

Software parallelization likewise can also benefit from parallelism reconnaissance in which code is evaluated for suitability for parallelization. Until now, there have been limited tools support to do this. However, Intel® Advisor XE 2013 changes this and helps the world of parallelization leapfrog forward. Intel® Advisor XE is the newest component of the Intel® Parallel Studio XE suite of products.

Software parallelization is potentially destabilizing to code, risky, expensive, and complex. Current trial and error approaches are not productive and there is considerable risk of dead ends. Embarking on code parallelization based on measured data (for example, hotspots) is perhaps better, but is likewise mostly a hit or miss. Code may or may not scale well. Stability issues due to incorrect parallelization also may lurk and surface long after the code is productized, and become costly to fix.

Intel® Advisor XE is built to help you find where to add parallelism to your code. Use it to discover the parallel performance (scalability) and code/data sharing issues (correctness) of possible parallel code regions. It lets you model several different regions within your program at once for parallel scalability and correctness. The results help you make judicious choices about which regions of code to not parallelize (to avoid dead ends), and which regions of code to actually parallelize to reap the multicore performance benefits.

Using this methodology helps you fix data sharing issues *before they happen*. Even as you prepare the code for parallelization by fixing the correctness issues, you can continue to use your existing test frameworks to validate your program—as it remains functionally unchanged and correct.

Use of Intel® Advisor XE in your parallelization efforts is very likely to reduce risk and increase the reward. Moreover, the tool empowers everyone in the software organization with the skill to productively parallelize, instead of the current situation where just the architects and senior engineers have this capability.

You can see how exciting the potential is for your applications. Please explore the product in greater detail at the [Intel® Advisor XE product page](#), and let us know what you think. □

POINTER CHECKER: Easily Catch Out-of-Bounds Memory Accesses

by Kittur Ganesh, *Technical Consulting Engineer*

This article introduces a powerful new feature called Pointer Checker, which precisely and easily isolates elusive bugs in programs. Found in the Intel® C++ Composer XE 2013 product, its integration into the compiler adds powerful functionality in a way that slides seamlessly into build systems. Clever implementation and powerful error reporting provide precise information about latent program defects. We are excited that during beta testing of this new feature, customers reported that this tool found numerous defects.

Although C/C++ pointers have well-defined semantics, many applications could still make out-of-bounds memory accesses which can go undetected, risking data corruption and increasing vulnerability to malicious attacks. The Pointer Checker provides full checking of all memory accesses through pointers. A pointer-checked enabled application will therefore catch out-of-bounds memory accesses before memory corruption occurs.

With the advent of multicore processors, there is a need to program for data and thread parallelism where data is frequently created, stored, shared, and accessed in memory through pointers. The C and C++ languages define good semantics for memory access through pointers, but they also permit the use of these pointers without any restrictions. This provides no built-in protection against accessing or writing most user data in memory. This means you can perform any number of arbitrary operations on the pointers—resulting in severe unforeseen errors in the program whose effects often appear random due to unintentional modification of data—causing out-of-bounds (OOB) memory accesses which may often go undetected.

Although pointers have well-defined lower and upper bounds, languages (and therefore the compilers) typically don't enforce bounds checking due to performance and speed concerns. This paves way for potential buffer overflows and overruns in various parts of the application code—causing data corruption, erratic program behavior, breach of system security, etc.—and is the basis for many software vulnerabilities to malicious attacks.

The launch of Intel® Parallel Studio XE 2013 brings a key new feature: the Pointer Checker, which performs bounds checking—providing full checking of all memory accesses through pointers—and identifies any out-of-bounds access in Pointer Checker-enabled code. This article presents a comprehensive overview and usage model of Pointer Checker, enabling you to quickly get started using this key debugging feature on your critical applications.

Overview

The Pointer Checker is a key feature of Intel Parallel Studio XE 2013. The main functionality of Pointer Checker is to find buffer overflows or overruns occurring in applications developed in high-level C and C++ languages on Windows* or Linux* operating systems. A buffer overflow or a buffer overrun is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory. This is a special case of violation of memory safety. For example, consider an array as the buffer as shown in the short code snippet in [Figure 1](#).

```
char *buf = (char *)malloc(5);
for (int i=0; i<=5;i++) {
    buf[i] = 'A' + i;
}
```

Figure 1

A buffer overflow occurs when you try to put more items in the array than what the array can hold. It occurs generally from writing or a store operation. On the other hand, a buffer overrun occurs when you are iterating over the buffer and keep reading past the end of the array. It generally occurs from reading or a load operation. Additionally, simple coding errors are often very hard to locate and rectify. For example, pointers are invariably masked by casting to a void pointer and then recasting to other pointers, making it very difficult to identify the cause of errors in the application. As mentioned earlier, since a pointer has a well-defined lower and upper bound, Pointer Checker performs bounds checking for all memory accesses through pointers—ensuring that a pointer is within bounds before its use for either a read or a write operation.

The Pointer Checker feature can be enabled via *compile time switches*. When you build your application with the Pointer Checker-enabled option, it will identify and report out all out-of-bounds memory accesses occurring in the application, including subscripted array accesses. In addition, the Pointer Checker can also detect *dangling pointers*, meaning pointers that point to memory that has been freed. When you build your application with the dangling pointer detection-enabled option, using a dangling pointer in an indirect access will also cause the Pointer Checker to report out an out-of-bounds error. Another useful feature that Pointer Checker offers is to check bounds for *arrays without dimensions*, which is especially important since applications are integrated with many different modules developed by different developers who often extern shared data.

[SEE FULL ARTICLE](#)



BLOG highlights



Minimize frustration and maximize tuning effort with Amdahl's Law

BY SHANNON CEPEDA

I recently had a question from a customer who had introduced a successful optimization to a hot function in his application, but did not see as much improvement in the overall application as he expected. This is a fairly common occurrence in the iterative process of performance tuning. Usually it happens for one of two reasons.

1. Introducing an improvement in one area resulted in inefficiencies somewhere else. This is par for the course with performance tuning, and part of the reason why the process is **iterative**. It can be hard to anticipate whether a code change you are making in one function will decrease performance somewhere else down the road, and so landing in this situation from time to time is unavoidable. Although you may not be able to always prevent it, using good documentation practices and a tool like Intel® VTune™ Amplifier XE to quantify performance changes can help you see when it is happening...

SEE THE REST OF SHANNON'S BLOG:



Visit [Go-Parallel.com](#)

Browse other blogs exploring a range of related subjects at Go Parallel: Translating Multicore Power into Application Performance.



New Parallel Programming Features in

Intel® (Visual) Fortran Composer XE



by Steve Lionel,
Developer Products Division

Fortran programmers have been doing parallel processing for many years using methods outside the Fortran standard such as auto-parallelization, OpenMP*, and MPI. Fortran 2008, approved as an international standard in late 2010, brought parallel programming into the language for the first time with not one, but two language features. (Of course, you can't be parallel with just one.)

This article provides a brief overview of these two new features, DO CONCURRENT and coarrays. The former is pretty easy to get one's head around; the latter is not.

DO CONCURRENT

Back in the early 1990s, an attempt was made at extending the Fortran 90 language for high performance computing and parallel processing. Called High Performance Fortran or HPF, it attempted to build on Fortran 90's array syntax in a way that permitted array operations to be done in parallel. HPF introduced the FORALL and WHERE constructs, PURE procedures with no side effects, and a number of intrinsic procedures for operations such as scatter/gather. While HPF was not widely adopted, some pieces of it were incorporated into the Fortran 95 standard, approved in 1997.

Of the various HPF features that persisted, none was perhaps more misunderstood than FORALL. Here's an example of a FORALL construct:

```
REAL :: A(10, 10), B(10, 10) = 1.0
...
FORALL (I = 1:10, J = 1:10, B(I, J) /= 0)
  A(I, J) = REAL(I + J + 2)
  B(I, J) = A(I, J) + B(I, J) * REAL(I * J)
END FORALL
```

The parenthesized list after the FORALL keyword is called the *forall-header*. It has one or more *forall-triplets* that specify the range of values taken on by the index-name. In this example we have two forall index names, I and J, which are each specified to take values from 1 to 10. The increment, if not specified by the third element in the triplet, is 1, just as in Fortran 90 array notation. The last part is the mask expression that determines the conditions under which the FORALL construct body (the two assignments) is executed.

Many Fortran programmers looked at FORALL and saw a loop, or in this case, two nested loops, perhaps with an IF at the top that skips to the next iteration, if the expression is false. But FORALL is **not** a loop construct, it is a "masked array assignment." If you try to think of this as a loop, you might expect each iteration to execute both assignments, and that these could be done in parallel. But that's not how FORALL was defined. Instead, the first assignment is executed completely, across all combinations of all the index names, filtered by the mask. Then, the second assignment is executed completely, again across all combinations and filtered by the mask. Inside a FORALL construct, an assignment statement may reference functions if they are PURE, but the only statement types allowed in a FORALL are assignment statements, WHERE constructs, or other FORALLs.

FORALL was a noble experiment, but the rules were too restrictive to be amenable to doing the assignments in parallel and it did not meet the needs of the Fortran community. So, Fortran 2008 brings what I call "FORALL Done Right": DO CONCURRENT.

A DO CONCURRENT construct looks like a blend of traditional DO and FORALL. In fact, the beginning of a DO CONCURRENT uses the FORALL header syntax. For example:

```
DO CONCURRENT (I=1:N)
  T = A(I) + B(I)
  C(I) = T + SQRT(T)
END DO
```


As with FORALL, the mask is optional. If present, it reduces the set of active combinations of the index names to those where the mask expression is true. Unlike FORALL, each range of a DO CONCURRENT is an iteration and is executed independently for all the active index combinations.

There are some restrictions on what you can have in a DO CONCURRENT. For example, you can't RETURN or GO TO out of the construct, and you can't reference a variable that is defined or made to be undefined by another iteration. You can even do I/O in a DO CONCURRENT, so long as a record written by one iteration is not read by another. As with FORALL, any procedure called from within the construct must be PURE (which guarantees that it has no side effects). Note that it is the programmer's responsibility to ensure that there are no dependencies between loop iterations—the compiler is not required to check these for you.

DO CONCURRENT is supported as of Intel® [Visual] Fortran Composer XE 2011 and the compiler will attempt to execute the construct in parallel if you have enabled auto-parallelization (/Qparallel or -parallel). However, there is no guarantee that any particular DO CONCURRENT will be run in parallel, and, of course, the order in which the iterations run is unpredictable. As a side effect, use of DO CONCURRENT can also help with automatic vectorization, as you are guaranteeing that there are no loop-carried dependencies.

Coarrays

If you are an MPI programmer, you know the basic drill: collect some data, call MPI_SEND to send it to a copy of your program running on another "node," and then use MPI_RECV to get results back. (This is a simplification, of course.) Wouldn't it be nice to be able to "reach out and touch" the other copies of your program using normal Fortran syntax, and not have to worry about adding calls to move data around?

Coarray Fortran, first proposed in the 1990s as an extension of Fortran 90 called F- (F minus minus), provides simple syntax for adding parallelism to a Fortran program. (The syntax is simple, though the definition and implementation is not.) It was implemented by Cray for its T3E and X1 supercomputers in the early 2000s, and was added, in a modified and somewhat reduced form, into the Fortran 2008 standard. Intel released the first full implementation of Fortran 2008's coarrays for mainstream computers in the Intel (Visual) Composer XE 2011 release for Linux* and Windows.*

The fundamental concepts of Coarray Fortran are these:

- Image: Multiple copies of your application run in parallel; each is called an *image*.
- Coarray: Variables become *coarrays* when they are given the CODIMENSION attribute. Somewhat confusingly, scalars can also be coarrays – the standard defines a coarray as any entity with a non-zero corank, and these can be scalars or arrays. *Codimensions* (and *coindices*) are denoted with square brackets [].

Coarrays are split up across all the images of your application, so that a portion of each coarray resides in the local memory of an individual image. This property is associated with the Partitioned Global Address Space (PGAS) parallel programming concept. Here, coarrays exist in a shared "address space," but image-specific segments are individually addressable. Let's look at a simple example.

We will declare an array A with dimension 10x20 and with one codimension:

```
real, dimension(10,20), codimension[*] :: A
```

It helps if you think of codimensions as additional dimensions, and indeed the Fortran standard limits the sum of the number of dimensions and codimensions to fifteen. (Intel® Fortran supports 31 as an extension.) The last upper *cobound* in the codimension must be ***; at runtime this takes on the value of the number of images. If when run there are eight images, the cobounds of A are 1:8.

As with dimensions, you can have multiple codimensions with lower and upper bounds, and as with dimensions, only the last one may have *** as an upper bound. So we might have:

```
integer, codimension[4,2:6,3:*] :: B
```

When you reference a coarray, you can do so with or without the coindices, which are enclosed in square brackets. If no coindices are present, you are referencing your image's piece of the coarray. If the coindices are present, you are specifying the coindex of the image you want.

Now, at this point you might be asking what happens if there aren't enough images to fill up the coindices, just as you would with a regular array that's an error. Unlike a regular array, the "shape" of a coarray may be ragged. Using the B example above, 20 images are needed to fill in each "layer" of the coarray. If there are, say, 39 images, there is a coindex [3,6,4], but not [4,6,4]. (Remember that Fortran does things in column-major order where the left subscript varies the fastest.)

Intrinsic procedures are provided to allow you to find the number of images, index of your own image, and the cobounds of any coarray.

What makes coarrays so nice is that they are integrated thoroughly into the Fortran language. You can use a coarray in most places where a regular variable is allowed, such as:

- Expressions and assignments
- Arguments to procedure calls
- I/O statements

This makes a program using coarrays look clean. For example, consider a Jacobian solver that breaks up the problem into blocks of data. Most of the calculation involves an image's local block, but at the edges of each block it needs to consider values from "halo cells," those on the edge of adjacent image's chunks. Here's what such code might look like using coarrays: (Figure 1)

“With Intel® [Visual] Fortran Composer XE you get coarray support in a “shared memory” mode, running on a single system. To build a coarray program just add the `-coarray` (or `/Qcoarray`) compiler option and then run the executable as normal. No special configuration is required.”

```
my_subgrid( 0, 1:my_M) = my_subgrid( my_N, 1:my_M)[my_north_P,me_Q]
my_subgrid( my_N+1, 1:my_M) = my_subgrid( 1, 1:my_M)[my_south_P,me_Q]

my_subgrid( 1:my_N, my_M+1) = my_subgrid( 1:my_N, 1 )[me_P, my_east_Q]
my_subgrid( 1:my_N, 0 ) = my_subgrid( 1:my_N, my_M )[me_P, my_west_Q]
```

Figure 1

Fortran defines additional coarray behaviors that ease programming. For example:

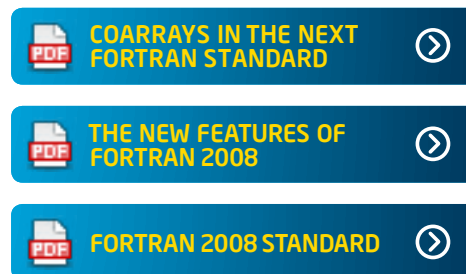
- > You can have ALLOCATABLE coarrays (and in fact this is the most common usage), where every allocation is a synchronization point, to make sure that all images have allocated their coarrays consistently and completely.
- > All images can do I/O. Normally, each has its own set of unit numbers, but the language says that “standard output” (unit 6) is preconnected on all images. While an implementation is not required to “merge the streams,” Intel Fortran does, so all standard output writes get displayed on the console where the image is run. “Standard input” (unit 5) is preconnected on image 1 only.
- > Every image has an implicit synchronization point at its start and again at its end.

The language provides several methods of synchronization among images. The `SYNC ALL` statement causes all images to wait until all of them have executed that `SYNC ALL` the same number of times. `SYNC MEMORY` makes sure that all memory updates have completed before continuing. `SYNC IMAGES` is like `SYNC ALL`, but you restrict the synchronization to a specified set of images.

There are also locks, declared using the `LOCK_TYPE` defined in intrinsic module `ISO_FORTRAN_ENV`, and `LOCK` and `UNLOCK` operations on these. Lastly, there is the ability to do atomic (uninterrupted) reads and writes of integer and logical variables through the `ATOMIC_DEFINE` and `ATOMIC_REF` intrinsic procedures (these last are newly supported as of Intel® Fortran Composer XE 2013).

With Intel® [Visual] Fortran Composer XE you get coarray support in a “shared memory” mode, running on a single system. To build a coarray program just add the `-coarray` (or `/Qcoarray`) compiler option and then run the executable as normal. No special configuration is required. To add support for a “distributed memory” model across a cluster requires that you also have a license for Intel® Cluster Studio (in addition to having a cluster.) Yes, this applies to Windows clusters too. (Support for distributed-memory coarray applications on Windows was added in Update 6 of Intel® Visual Fortran Composer XE 2011).

For further reading about Fortran 2008, including coarrays and `DO CONCURRENT`, you can refer to the following documents from the Fortran standards committee:



Using the Intel® Math Kernel Library (Intel® MKL) and Intel® Compilers to Obtain Run-to-Run Numerical Reproducible Results

by Todd Rosenquist, *Technical Consulting Engineer, Intel® Math Kernel Library*
and Shane Story, *Manager of Intel® MKL Technical Strategy*

Floating-point applications from Hollywood to Wall Street have long faced the challenge of providing both great performance and exactly the same results from run to run, or in other words, reproducible results. While the main factor causing a lack of reproducible results is the non-associativity of most floating point operations, there are other contributing factors such as runtime, selectable optimized code paths, non-deterministic threading and parallelism, array alignment, and even the underlying hardware floating-point control settings.

In this article for Intel® software tool users and programmers, we outline how to use the Intel® Math Kernel Library (Intel® MKL) and Intel® compiler features to balance performance with the reproducible results applications require. These new reproducibility controls in Intel® Parallel Studio XE 2013 help make consistent results from run to run possible:

INTEL® SOFTWARE TOOLS REPRODUCIBILITY CONTROLS

Intel® MKL 11.0	<code>mkl_cbwr_set()</code> MKL_CBWR (environment variable)
Intel® Composer XE 2013	<code>-fp-model</code> or <code>/fp</code> KMP_DETERMINISTIC_REDUCTION=yes

After many years of seeing software performance increase with processor clock speed, the last half-decade has seen the flattening of clock rates and the increasing availability of multicore systems. With each successive generation of microprocessors, improvement in software performance requires the use of newly added instructions to exploit the capabilities of the processor, as well as threaded algorithms designed to leverage the growing number of computational cores. To keep up with these changes, many developers turn to software tools. Optimizing compilers exploit opportunities for instruction and data-level parallelism and can automatically thread computationally intensive portions of a program. Software libraries provide tools to thread your code or allow you to extract parallelism automatically through calls to highly optimized, threaded functions. Many software programmers have adopted and use these high performance tools to extract greater levels of performance. In doing so, the likelihood of generating inconsistent results from run to run has grown.

Let's consider two scenarios. Artists in animation studios work every day with advanced modeling tools that allow them to move their actors through a virtual world. These modeling tools include physics engines that can simulate the real-world behavior of clothes, hair, or fluids, and therefore will naturally use floating-point models similar to those used in science and engineering applications. While accuracy and precision may not always be the first concern, especially in early stages of the process, getting the same results can be of the utmost importance. If a cloak follows a slightly different trajectory each time the artist runs through a multi-second sequence, the artist has lost some control over the creative process. Which trajectory will be used when the scene goes through further rendering and post-processing steps? The problem would be compounded by the fact that a single scene may have many such models that may interact to produce completely unpredictable results.

A second scenario involves mathematicians on Wall Street who develop algorithms for various applications from options pricing to risk analysis. In this field, getting results quickly means money—and sometimes a lot of money. The “quants” who develop these algorithms are faced with a balancing act between getting the answer quickly and the simulation time required to provide the most reliable answer. An increase in the performance of an algorithm can mean a decision sooner or a better decision in the same amount of time—a win in either case. However, optimized floating-point calculations that are a part of these models can often introduce rounding error. This means that if an earlier decision must be revisited and the model run again, it is possible that the result might be slightly different. The uncertainty can result in questions or issues later that programmers would prefer to avoid.

These are just two of many scenarios¹ encountered over the last few years by users of Intel MKL. This is a popular library of highly optimized parallel floating-point math functions that has been successfully used by customers in many application areas for over 15 years. For application programmers who demand reproducible results, there have not been any guarantees and only the limited option of running a sequential version of the library.

“Floating-point applications from Hollywood to Wall Street have long faced the challenge of providing both great performance and exactly the same results from run to run, or in other words, reproducible results.”

So, what exactly is the reproducibility problem? The issue is rooted in the way floating-point numbers are represented, the order in which they are operated on by the computer, and the rounding errors that may be introduced. It is a well-known fact that for general floating-point numbers represented in an IEEE single or double precision format², the mathematical associative property does not in general hold.³ In simpler terms, $(a + b) + c$ may not equal $a + (b + c)$.

It may help to consider a specific example. With pencil and paper, $2^{63} + 1 + -1 = 2^{63}$. If, instead we do this same computation on a computer using double precision floating-point numbers, we get $(2^{63} + 1) + (-1) \approx 1 + (-1) = 0$ since $(2^{63} + 1)$ rounds to 1, or possibly $2^{63} + (1 + (-1)) \approx 2^{63} + 0 = 2^{63}$ through a slight modification in the order of operations. Clearly 0 does not equal 2^{63} , so the order of operations not only influences how and when rounding occurs but also the final computed result. Compilers typically refer to this ordering ambiguity as re-association.

Introducing application-level parallelism further increases the likelihood of producing nonreproducible results. The reason is a direct carryover from the order of operations argument just described. Whenever work is distributed among multiple threads or processes, any change in the order of operations within a computational dependency chain may result in a difference not only in the intermediate results, but also in the final computed results. Straightforward array element sum and product reduction operations are simple examples when the array elements have been distributed across multiple threads; partial sums or products are computed and then combined across threads into a single value. Any change in how the arrays are distributed, or the order in which a thread-specific sum or product is combined with another, may influence the final reduced sum or product. More broadly, how to handle parallelism in a consistent and predictable way falls under the category of deterministic parallelism.⁴

When you consider that a typical application may do millions of floating-point operations, it becomes readily apparent how the order of operations influences the final computed results.

Intel Math Kernel Library

Intel MKL 11.0 introduces Conditional Numerical Reproducibility functions to help users obtain reproducible floating-point results from Intel MKL functions under certain conditions.⁵ When using these new features, Intel MKL functions are designed to return the same floating-point results from run to run, subject to the following limitations:

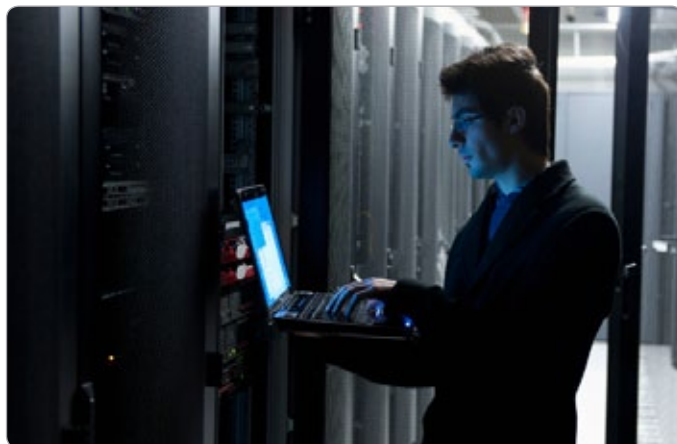
- Input and output arrays in function calls must be aligned on 16-, 32-, or 64-byte boundaries on systems with SSE/AVX1/AVX2 instructions support respectively.
- Control over the number of threads must remain the same from run to run for the results to be consistent.

The application-related factors within a single executable program that affect the order in which floating-point operations are computed include code path selection based on runtime processor dispatching, data array alignment, variation in number of threads, threaded algorithms, and internal floating-point control settings. Up until now, users were unable to control the library's runtime dispatching and how its functions were internally threaded. However, they were able to manage the number of threads, check the floating-point settings, and take steps to align memory when it is allocated.⁶

Intel MKL does runtime processor dispatching in order to identify the appropriate internal code paths to traverse for the Intel MKL functions called by the application. The code paths chosen may differ across a wide range of Intel® processors and IA-compatible processors, and may provide varying levels of performance. For example, an Intel MKL function running on an Intel® Pentium® 4 processor may run an SSE2-based code path. On a more recent Intel® Xeon® processor supporting Intel® Advanced Vector Extensions (AVX) that same library function may dispatch to a different code path that uses AVX instructions. This is because each unique code path has been optimized to match the features available on the underlying processor. This feature-based approach to optimization, by its very nature, amplifies the reproducibility challenges already described. If any of the internal floating-point operations are done in a different order, or are re-associated, then the computed results may differ.

[SEE FULL ARTICLE](#)

“With each successive generation of microprocessors, improvement in software performance requires the use of newly added instructions to exploit the capabilities of the processor, as well as threaded algorithms designed to leverage the growing number of computational cores.”



RESOURCES AND SITES OF INTEREST



Go Parallel



The mission of Go Parallel is to assist developers in their efforts toward “Translating Multicore Power into Application Performance.” Robust and full of helpful information, the site is a valuable clearinghouse of multicore-related blogs, news, videos, feature stories, and other useful resources.

“What If” Experimental Software



What if you could experiment with Intel's advanced research and technology implementations that are still under development? And then what if your feedback helped influence a future product? It's possible here. Test drive emerging tools, collaborate with peers, and share your thoughts via the What If blogs and support forums.

Intel® Software Network



Check out a range of resources on a wide variety of software topics for a multitude of developer communities ranging from manageability to parallel programming to virtualization and visual computing. This content-rich collection includes Intel® Software Network TV, popular blogs, videos, tools, and downloads.

Step Inside the Latest Software

See these products in use, with video overviews that provide an inside look into the latest Intel® software. You can see software features firsthand, such as memory check, thread check, hotspot analysis, locks and waits analysis, and more.

[Intel® Inspector XE](#)

[Intel® VTune™ Amplifier XE](#)

Intel® Software Evaluation Center



The Intel® Software Evaluation Center makes 30-day evaluation versions of Intel® Software Development Products available for free download. For high performance computing products, you can get free support during the evaluation period by creating an Intel® Premier Support account after requesting the evaluation license, or via Intel® Software Network Forums. For evaluating Intel® Parallel Studio, you can access free support through Intel® Software Network Forums ONLY.

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

Sign up for future issues | Share with a friend

The Parallel Universe is a free quarterly magazine. [Click here](#) to sign up for future issue alerts and to share the magazine with friends.



Announcing Intel® Parallel Studio XE 2013

Your performance toolset.



Get the mature toolset with an incomparable breadth and depth of features for developers—and accelerate application performance. Intel® Parallel Studio XE combines industry-leading compilers, performance and parallel libraries, error checking and performance profiling tools for C/C++ and Fortran.

**DISCOVER THE
PERFORMANCE IMPACT
FOR YOUR APPLICATIONS**



intel®
Software

©2012, Intel Corporation. All rights reserved. Intel and the Intel logo 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.

INTEL® SOFTWARE ADRENALINE



Introducing Intel® Software Adrenaline

Enter a world of doers, dreamers, and software industry luminaries.

This new magazine puts you on the forefront of the software industry. Roadmaps, R&D, industry pioneers and game changers, news, and more.



FREE SUBSCRIPTION
softwareadrenaline.intel.com

