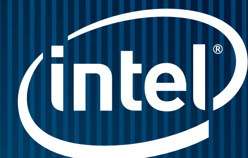
THE PARALLEL UNIVERSE



Issue 9
February 2012

Parallelizing DreamWorks Animation* Fur Shader

By Sheng Fu

Letter from the Editor

By James Reinders

New Analysis Tools in Intel® Cluster Studio XE

By David Mackay, Ph.D. and Krishna Ramkumar

HPC wire
Editors' Choice Awards 2011

intel

Spark Extreme Performance

Winner of the 2011 Editor's Choice Award:
Best HPC Software Product or Technology

Intel® Parallel Studio XE software development suite combines Intel's industry-leading C/C++ compiler and Fortran compiler; performance and parallel libraries; error checking, code robustness, and performance profiling tools into a single suite offering.

Choose this award-winning suite to help boost application performance and increase your code quality, security, and reliability for high-performance computing and enterprise applications.

© 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.

CONTENTS

Letter from the Editor
Parallel Performance from Feature Films to Advanced Clusters
BY JAMES REINDERS 4
Examine the impact of applying data parallelism to a geometry generator, and analyzing massively parallel applications for correctness and performance.

Parallelizing DreamWorks Animation* Fur Shader
How Intel® tools help add parallelism in large applications
BY SHENG FU 6
Learn how Intel® parallelization tools help DreamWorks Animation's fur shader take advantage of the performance and capabilities of multicore processors, while preserving legacy libraries needed in production.

New Analysis Tools in Intel® Cluster Studio XE
BY DAVID MACKAY, PH.D. AND KRISHNA RAMKUMAR 18
Improve hybrid application analysis with new cluster tools, including Intel® VTune™ Amplifier XE. MPI programs can now be tuned more precisely and easily—regardless of the shared-memory programming model utilized in the hybrid—based on insight into the detailed activities on the nodes of a distributed program.

© 2012, Intel Corporation. All rights reserved. Intel, the Intel logo, Intel Core, Cilk, VTune, and Xeon are trademarks of Intel Corporation in the U.S. and other countries. *Other names and brands may be claimed as the property of others.

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.

PARALLEL PERFORMANCE

EXPANDS WHAT'S POSSIBLE FROM FEATURE FILMS TO ADVANCED CLUSTERS.

LETTER FROM THE EDITOR

James Reinders Chief Software Evangelist at Intel Corporation. His articles and books on parallelism include *Intel Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism*, which has been translated into Japanese, Chinese, and Korean. Reinders is also widely interviewed on the subject of parallelism.

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

Explaining what you do for a living to your children is easier if you can take them to a movie they like, and tell them that you had something to do with making it. Working on the Intel® tools, I have had that opportunity multiple times over the years with movies that were assisted by our tools. I'm pleased to share some of that excitement with you in this issue.

Do you know what panda bears, reluctant dragons, hybrid applications, and cluster analysis all have in common? They represent the innovations made possible by the latest parallel programming tools from Intel, and they come to life in this issue of *The Parallel Universe* magazine.

Animators create over two hundred attributes of fur, which are enabled by developers who move easily through loops and threads at DreamWorks Animation.

Parallelizing DreamWorks Animation* Fur Shader explains the application of data parallelism to a geometry generator, and shows the impact of tools such as Intel® Threading Building Blocks (TBB), Intel® Math Kernel Library (MKL), Intel® Inspector XE 2011, Intel® Code coverage tool (part of Intel® C++ Composer XE), and Intel® VTune™ Amplifier XE. You can apply the insights to other domains as you make dependent libraries thread safe, and improve scalability on multicore platforms.

New Analysis Tools in Intel® Cluster Studio XE takes us from the microscopic world of the texture of hair and blades of grass to the macro-level of accelerating cluster performance. It illustrates the ways Intel® Inspector XE and Intel® VTune™ Amplifier XE help you analyze a system, even if it is a system with hundreds or thousands of processors. The latest versions can run these tools across a cluster to analyze massively parallel applications for correctness and performance, and efficiently understand what is happening on each node. Whether you care about one processor or thousands, these are tools well worth knowing and using.

Our movies are better because of our high performance tools, and you'll learn more about how that happens in this issue. You'll also learn about two amazing tools that can make you an expert on what your program is really doing, and locate threading, memory, and performance issues on any system. Don't forget to use your knowledge to spark the imagination of a young future computer scientist by heading out to a theater, and enjoying a movie and some popcorn.

James Reinders
February 2012

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

Parallelizing DreamWorks Animation* Fur Shader

How Intel® tools help add parallelism in large applications

by Sheng Fu, *Software and Services Group*

In computer-animated movies, furry surfaces are crucial in making 3D objects look more realistic. The fur shader developed by DreamWorks Animation and used in production is a powerful geometry generator that simulates different furry surfaces, such as fur, hair, and grass. Fur shader tends to consume large computing resources; however, it is also an excellent candidate for data parallelism since the computation for each hair is on its own data set and is independent of each other. In this paper, we will talk about how Intel parallel tools help DreamWorks Animation to parallelize the fur shader to generate deterministic result, make dependent libraries thread safe, and improve scalability on multicore platforms. Although this paper discusses parallelizing DreamWorks Animation fur shader, the techniques, the tools, and the methodologies are generic enough to be applied in parallelizing applications in other software domains as well.

Sign up for future issues | Share with a friend

Introduction

In computer-animated movies, furry surfaces are crucial in making 3D objects look more realistic. The fur shader developed by DreamWorks Animation and used in production is a powerful geometry generator that simulates different furry surfaces, such as fur, hair, and grass. Fur shaders tend to consume large computing resources since millions of fur/hair/grass surfaces are needed in production. A scene with even a few furry surfaces may take hours to render. **Figures 1** to **3** show examples of scenes from popular movies produced by DreamWorks Animation.

The DreamWorks Animation fur shader is an excellent candidate for data parallelism since the computation for each hair is on its own data set and is independent of each other. However, the fur shader is also quite complex: 200+ fur attributes that artists can tweak, thousands of lines of code, and dependency on many other DreamWorks Animation libraries. In this paper, we discuss how Intel® tools, such as Intel® Threading Building Blocks (Intel® TBB), Intel® Math Kernel Library (Intel® MKL), Intel® Inspector XE 2011, Intel® Code coverage tool (part of Intel® C++ Composer XE), and Intel® VTune™ Amplifier XE, help parallelize the DreamWorks Animation fur shader, make dependent libraries thread safe, and improve scalability on multicore platforms.

Although this paper discusses parallelizing the DreamWorks Animation fur shader, the techniques, the tools and the methodologies are generic enough to be applied in parallelizing applications in other software domains as well.

Figure 1: Scene with grass from "Kung Fu Panda 2"

Figure 2: Scene with hair from "How to Train Your Dragon"

Figure 3: Scene with fur from "Kung Fu Panda 2"

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

Overview of the fur shader

The fur shader is a geometry generator. It generates a large number of small hairs to add details to a surface on which hair needs to be grown. Each hair goes through steps identified in **Figure 4**.

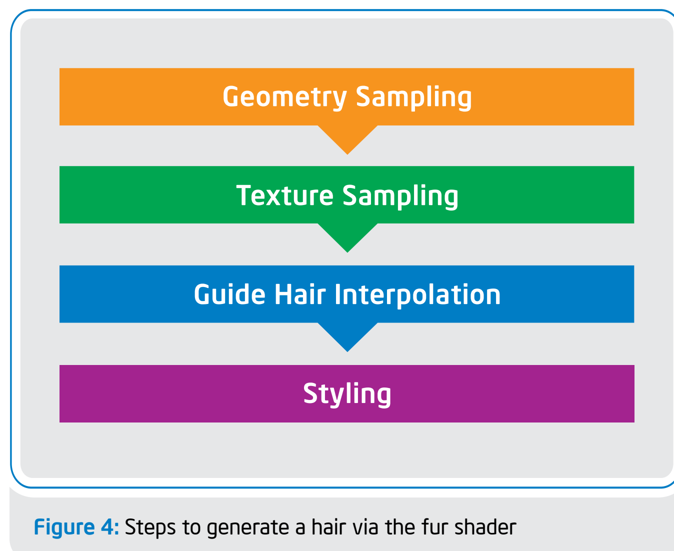


Figure 4: Steps to generate a hair via the fur shader

In the geometry sampling step, the fur shader randomly picks a location on source geometry and evaluates the geometry attributes, including position, normal texture coordinates, and derivatives, on that location.

Many attributes of hair can be bound to a texture. In the texture sampling step, the fur shader evaluates hair attributes by looking up texture maps.

During guide hair interpolation, the fur shader finds the guide hairs that are neighbors to the current hair, and then determines its shape based on the interpolation of the neighboring guide hairs' shapes.

Finally, during styling, the fur shader adds attributes such as kink, curl, gravity, and wind to change the look of a hair, and adds other subtle details to make a hair look realistic.

Clearly, the computation cost for each hair is very high, and the number of hairs needed in production is also very large. Since the computations for each hair are independent of each other, the fur shader is an excellent candidate for parallel processing.

Parallelizing the fur shader with Intel® Threading Building Blocks

The fur generation loop for each hair is the most time-consuming region in the DreamWorks Animation fur shader. There are several parallel programming models available from Intel® software products to choose from: OpenMP*, Intel Threading Building Blocks, and Intel® OpenCL. Since parallelism is being added to an existing C++ object-oriented application, OpenCL required significant code rewrites. OpenMP is very good for parallelizing loop oriented code—just add a `pragma` before the `for` loop. Intel TBB, however, may be a better choice if the code base is heavily C++ and object oriented, which is the case for the fur shader. Since Intel TBB is widely used at DreamWorks Animation in other projects, it was a natural choice for the fur shader parallelization.

Changing a regular `for` loop into TBB `parallel_for` is quite straightforward. **Figure 5** shows the existing fur code structure and the modifications needed for Intel® TBB `parallel_for` side by side. TBB `parallel_for` requires two input parameters: the first parameter is a `blocked_range` object, which defines the range of the for loop; the second parameter is a loop body object for which a developer must define a loop body class consisting of at least a copy constructor, destructor, and the operator( ). The code shown in **Figure 5,** however, uses the C++ lambda expression, which is defined in the C++0x standard. With the lambda expression, the compiler (e.g., Intel's C++ compiler) automatically generates the required loop body class, etc., which meet the requirements for the `parallel_for`. A nice aspect of the lambda expression is that the code inside the lambda expression can access the local variables defined before the `parallel_for` statement.

As seen from this example, it is quite easy to build parallelism in existing applications with Intel TBB especially since TBB runtime handles creation of thread pool, scheduling tasks to threads, etc.

```
void Fur::generateFur()
{
  for(size_t i=0; i<numHairs; i++)
  {
      // compute each hair
  }
}
```

```
void Fur::generateFur()
{
  tbb::parallel_for(
      tbb::blocked_range<size_t>(0,numHairs),
      [=](const tbb::blocked_range<size_t> &r)
      {
          for(size_t i = r.begin() i<r.end(); i++),
          {
              // compute each hair
          }
      } );
}
```

Figure 5: Left-hand side—original fur shader code. Right-hand side—modified code using Intel® TBB `parallel_for` with C++ lambda expression

Adding determinism to the fur shader using Intel® Math Kernel Library random number generation

Even though it is easy to parallelize the fur generation loop, there were a few other important issues to be resolved before production use. The first critical one was to ensure that the parallel implementation of the fur shader was deterministic; that is, it produced results independent of the number of threads or processor cores used.

DreamWorks Animation fur shader used random number generation function extensively to make generated fur look more natural. For example, the fur shader samples density maps randomly to find the location to grow hair on a surface. In addition, most of fur attributes have user-defined randomness as well. In the original DreamWorks Animation implementation, the fur shader used a proprietary implementation of drand48 algorithm to generate the various random numbers. However, the use of this random number generation function raised two challenges. First, the proprietary implementation was not thread safe. Even if a mutex is used to make the drand48 implementation thread safe, it would become a performance bottleneck since the fur shader called random number generation very often. Second, since each hair is generated in parallel, the random numbers generated for each hair might be different in different parallel runs, which would make the hairs look different in parallel runs.

However, in the fur shader code, it turned out that the maximum number of random numbers needed for each hair was fixed and known before the fur generation loop. Since the series of pseudo random numbers is fixed once the seed is specified, each hair can use the same sequence of random numbers in that series by its hair id (hair id is the loop iteration number which is independent of serial or parallel run). For example, if each hair needed 10 random numbers, and there were a total 10 hairs need to be generated, then the random numbers for the hairs may be distributed as follows:

Random numbers: r1, r2,......r10, r11, r12,......r20, ... r91, r92,......r100

Hair Id: hair_1 hair_2 hair_10

The offset to the beginning of the random number sequence for kth hair would then be:

Offset for hair_k = k * (maximum number of random numbers needed per hair)

One deterministic implementation would be to generate all the random numbers prior to the fur generation loop and store it in an array for use later through the hair_id offset. But there are millions of furs to be generated, and a more memory-efficient implementation would be to let the random numbers be generated as needed but with guaranteed sequence independent of parallelization. Intel MKL library supports a variety of random number generation (RNG) functions that have the behavior we needed. These RNG functions are thread safe and lock free. Most importantly, Intel MKL provides a function to skip ahead a certain number of random numbers in a stream/sequence, and then generate random numbers as needed. This is exactly what was needed for parallel fur generation. The code snippet in **Figure 6** shows the use of Intel MKL random number functions in the fur shader:

```
// generate a base random number stream
vslNewStream(&myBaseRandomBStream, VSL_BRNG_MCG59, seed);

...

// the code in fur generation loop body

for each hair do
{
  vslCopyStream(&currentRandomStream, myBaseRandomStream);
  vslSkipAheadStream(currentRandomStream, hair_id * offset_for_each_hair);

  //assume we want to get a single random number as we need
  double random_number;
  vdRngUniform(VSL_METHOD_DUNIFORM_STD, currentRandomStream,
               1, &random_number, range_low, range_high);

  // Use random number in fur generation code
  ...
}
```

Figure 6: Use of Intel® MKL RNG to generate deterministic fur results

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

Intel MKL function **vslSkipAheadStream** is designed to jump to the desired random number without computing the prior random numbers in the sequence/stream. This particular feature in short helped add determinism to parallel fur generation.

Combining Intel® Inspector XE and code coverage tool to find thread safety issues

The DreamWorks Animation fur shader logic is very complicated with about 256 attributes for artists to tune. The fur shader itself is over 30k lines of code, and it depended on 10+ other DreamWorks Animation libraries. For example, the fur shader depended on geometry libraries to sample different types of geometry, such as mesh, NURBS surface, and subdivision surface. Some of these libraries were developed and maintained over a long period of time, and were not designed to run in parallel. Our next challenge was to make the fur shader and the code paths in legacy libraries exercised by fur shader to be thread safe.

Intel Inspector XE is a tool we depended on heavily to find thread safety issues in the parallelized fur shader. It is a dynamic memory and threading error checking tool for developing serial and multithreaded applications. Users must define a test case, and then run it with Inspector XE. Inspector XE will detect data race and deadlock issues in the application as well as record the call stack when they occur.

The example TBB `parallel_for` code in **Figure 7** illustrates some of the key features of Intel Inspector XE.

In this example, global variable `g` has a data race issue since multiple threads can read and write variable g without protection.

```
#include "tbb/parallel_for.h"
#include "tbb/blocked_range.h"
using namespace tbb;
float g    = 0.0f;
int   flag = 1;

class my_tbb_test {
private:
  float *a;
public:
  void operator() (const blocked_range<size_t> &r) const
  {
      for (size_t i=r.begin(); i<r.end(); i++)
      {
          g ++;
          if(flag) {a[i] += g;}
          else {a[i] -= g;}
      }
  }
  my_tbb_test(const my_tbb_test &other)
  {
      a = other.a;
  }
  my_tbb_test(float *a)
  {
      this->a = a;
  }
};

void func_foo()
{
  printf("Test function for code coverage\n");
}

int main()
{
  float data[10000];
  memset(data, 10000*sizeof(float), 0);
  parallel_for(blocked_range<size_t>(0, 10000),
               my_tbb_test(&data[0]));
  return 0;
}
```

Figure 7: `parallel_for` example to illustrate some of the key features of Intel® Inspector XE

"Intel® Inspector XE is a tool we depended on heavily to find thread safety issues in the parallelized fur shader. It is a dynamic memory and threading error checking tool for developing serial and multithreaded applications."

In order to ensure that Inspector XE can fully analyze TBB applications, an Intel® compiler macro "TBB_USE_THREADING_TOOLS" should be defined for compiling the application. So the Intel® compiler command line should be as follows: (Figure 8)

```
icpc –o tbb_for tbb_for.cpp –tbb –DTBB_USE_THREADING_TOOLS –O2 –g
```

Figure 8

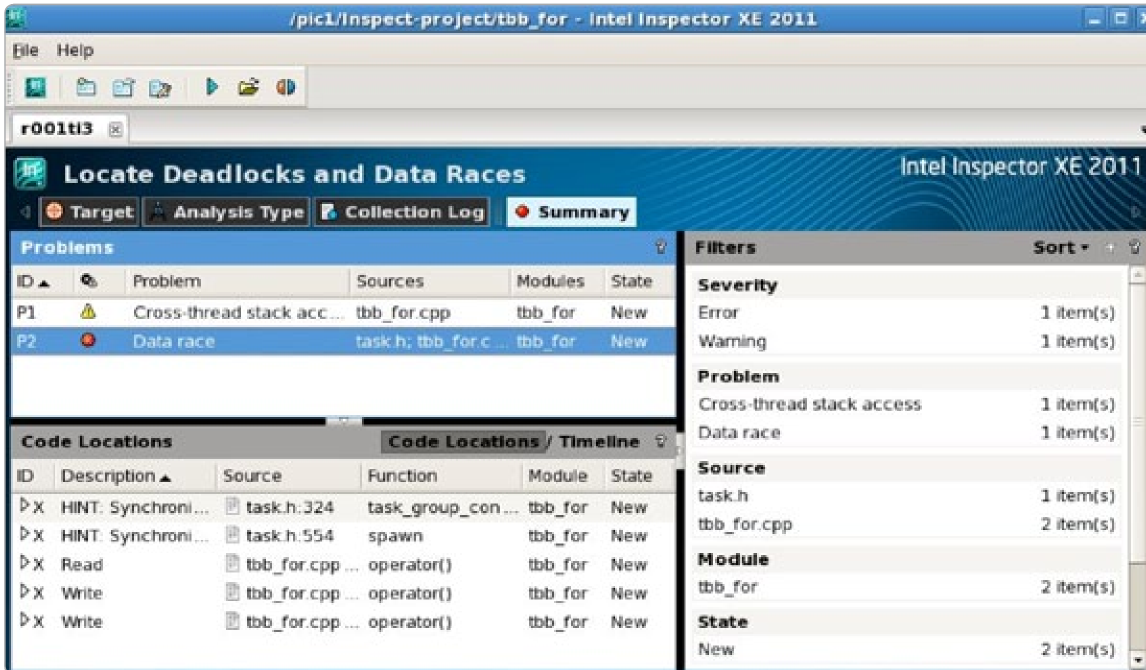


Figure 9: Intel® Inspector XE showing a data race issue in a test program

Then launch Inspector XE to run the application in order to locate deadlocks and data races. Figure 9 shows the screen shot of Intel Inspector XE after its analysis is completed.

In this example, one can see Inspector XE reporting a data race issue. If the row with data race is selected, Inspector XE will then show the call stack where this data race occurred (see Figure 10).

With the call stack for the data race, developers can usually figure out why the data race occurred quite easily.

Since Inspector XE only performs runtime analysis, the issues it can find are completely dependent on the test cases defined by developers. If the test cases do not exercise the code paths that have thread safety issues, Inspector XE won't be able to find them. It would not be too difficult for the author of the code to create appropriate test cases. But for a very large code base, such as the fur shader, developed over time and maintained by different authors, it can be quite challenging to define comprehensive test cases.

Interestingly, Intel® Composer C++ compiler provides a code coverage tool to help developers to analyze extent of code paths that are covered by test cases. Let us use the simple TBB parallel for code as an example. The following Intel® compiler options are needed to enable code coverage analysis:

```
icpc –o tbb_for tbb_for.cpp –tbb –g –prof-gen=srcpos –prof-dir ./code_coverage
```

Figure 11

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

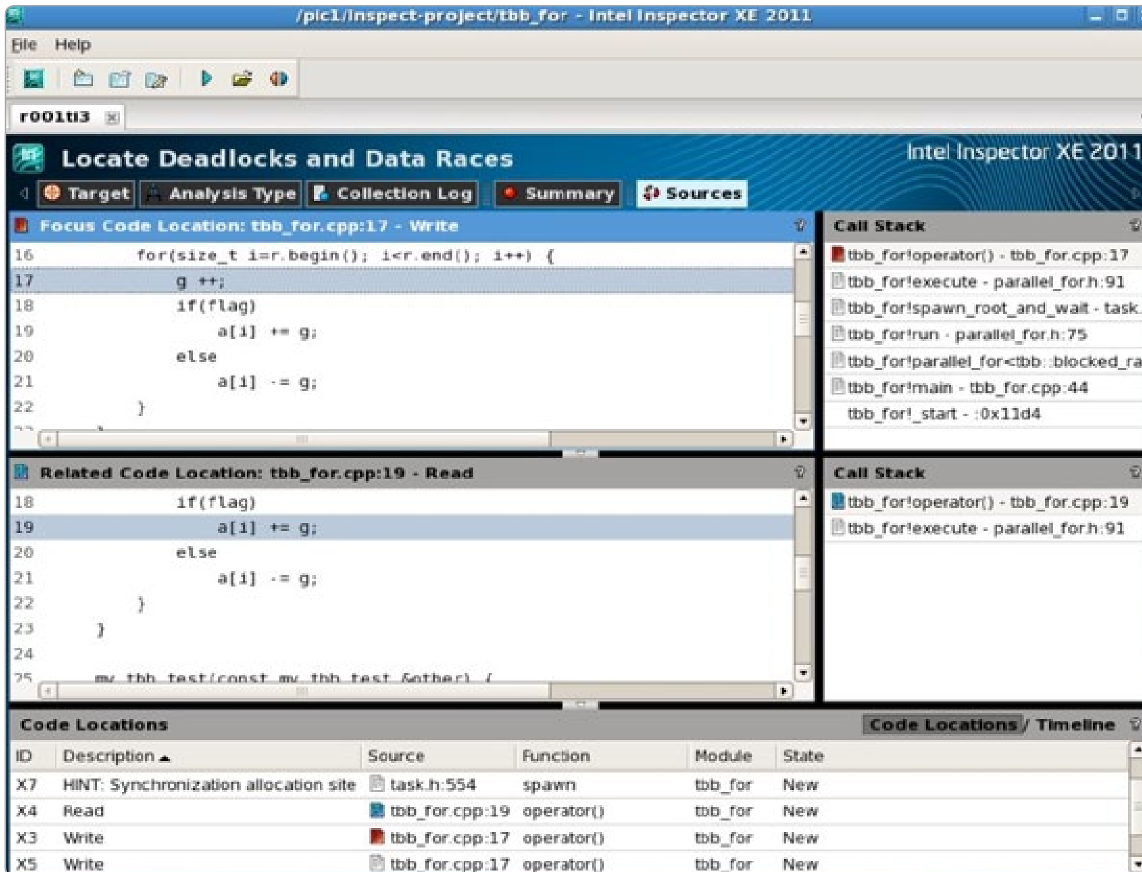


Figure 10: The call stack for a data race issue

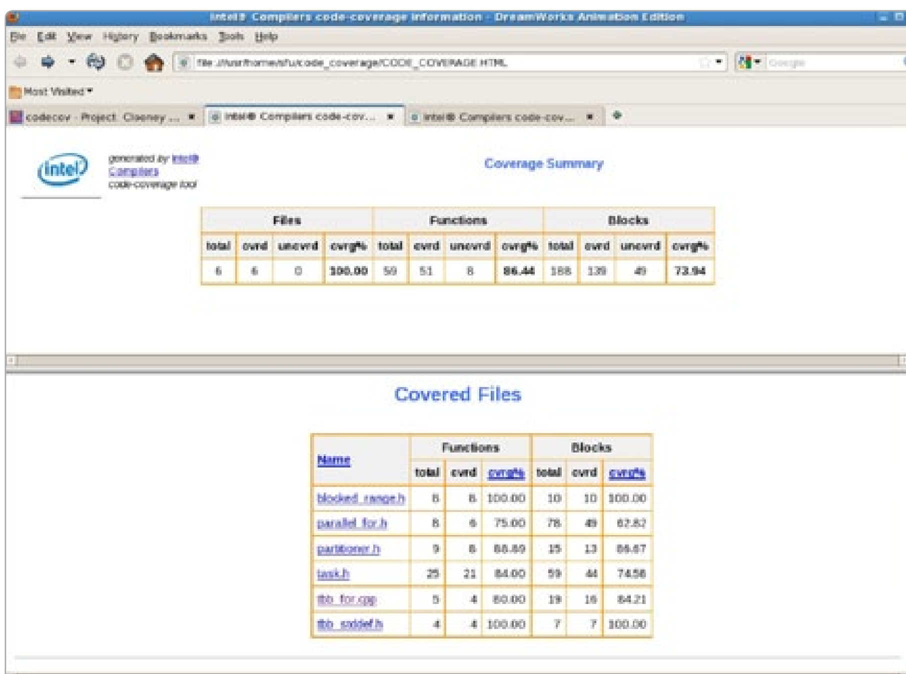


Figure 12: Intel® code coverage tool's analysis result

"In general, for large legacy codes, in order to introduce threading quickly, one may have to use mutexes to protect entire functions."

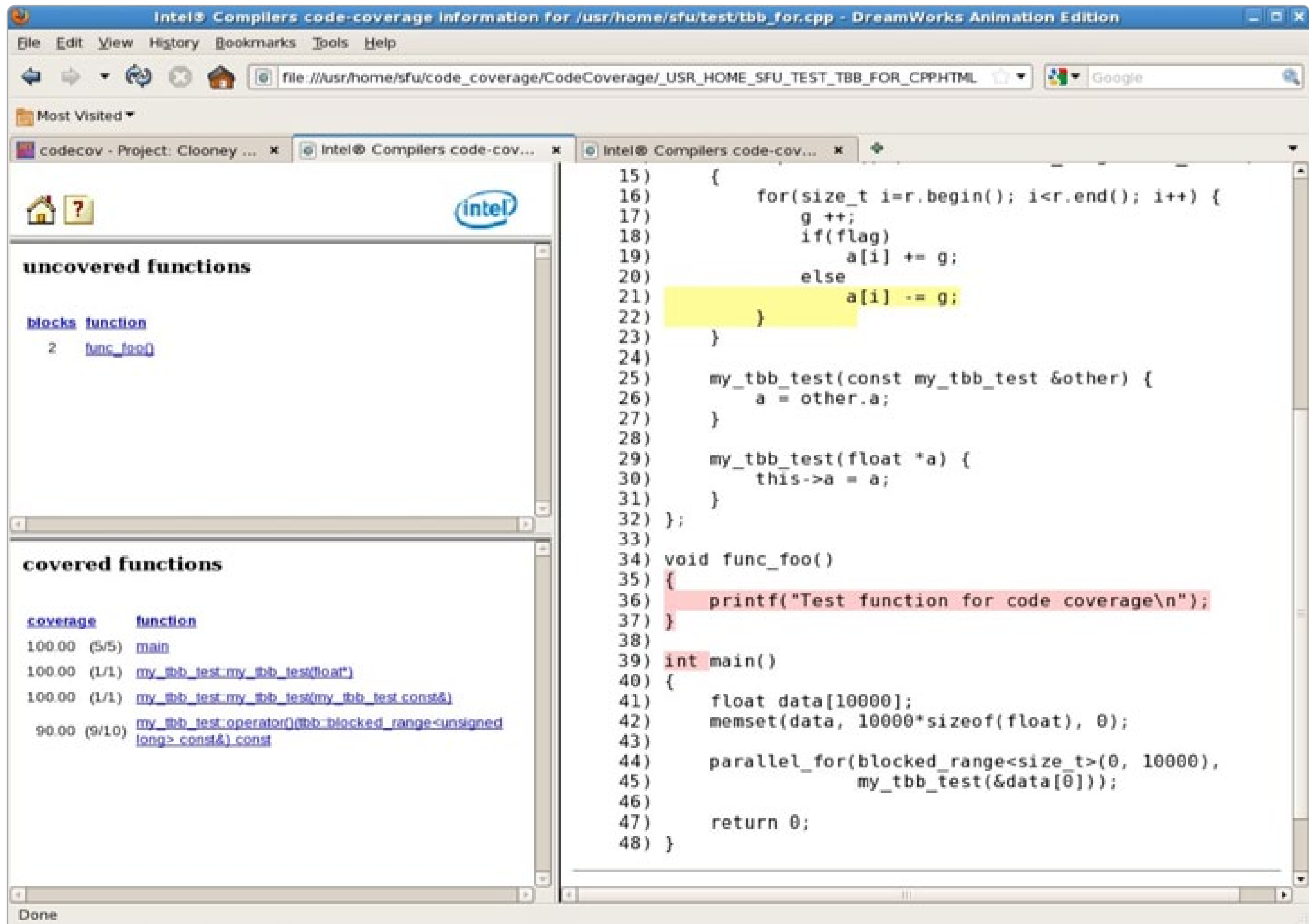


Figure 13: Intel® code coverage analysis result for file "tbb_for.cpp"

Option "prof-gen" enables code coverage analysis, and option "prof-dir" sets the directory to save code coverage results.

Now, the application built with above compiler options is run with several tests. With different runs, multiple files with various profile information are created in the ./code_coverage directory. Next, change directory to ./code_coverage, and run "profmerge". This command supplied with Intel® compiler will accumulates multiple *.dyn files into a single "pgopti.dpi" file. Finally, run "codecov –dpi pgopti.dpi" to generate an html page "CODE_COVERAGE.HTML." The following screen shot (**Figure 12**) shows what the code coverage analysis page looks like.

If the link for "tbb_for.cpp" is selected, a detailed code analysis result for that file is shown. (**Figure 13**)

On the left-hand side, the functions that are covered and uncovered by the tests are reported. On the right side, the code with white background color are fully covered while the codes highlighted with pink and yellow are uncovered by the tests (pink highlights point to uncovered functions, while yellow highlights point to uncovered basic blocks). From this simple example, one can see how Intel® compiler code coverage tool helps understand what execution code paths are covered by various test cases.

> "Intel® tools were critical in every step of this parallelization effort with DreamWorks Animation's fur shader."

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

Continuing on our DreamWorks Animation fur shader discussion, we used the Intel® code coverage tool to define about 100 test cases for the fur shader to get a good coverage of code execution paths. Then, running Inspector XE on these test cases helped us find a few more data race issues that would have been very hard to find with code inspection alone.

A caution on running large applications through Intel Inspector XE: since Inspector XE instruments the code heavily, applications usually run much slower than the un-instrumented regular version. So it is critical to design test cases that run for a very short time. In the DreamWorks Animation fur shader case, we reduced the number of hairs in fur generation loop since the hairs in the same fur generation loop usually go through similar code paths.

Different approaches to fixing thread safety issues

All thread safety issues found in the fur shader with Inspector XE were data races. The member functions in class `Fur` were designed to run with single thread, and these functions read/write the member variables of class `Fur`. However, with parallelization of the fur generation loop, these functions will be called from multiple threads. Thus, accessing the member variables of class `Fur` would be no longer thread safe. There are a few different approaches to fix such data race issues based on the different usages of the variables:

1. If a member variable is used inside a member function loop and there is no dependency of this member variable across loop iterations, then such a variable may be made as a local variable just in the scope of the loop statement. For example, in the fur shader, member variable mOutputHair, which saves the final result of the generated hair to pass on to tessellation, can be converted to a loop local variable since each thread needs a copy of that variable which is not shared across multiple threads. In addition, if a member function inside fur generation loop accessed mOutputHair, then the member function should be modified to accept mOutputHair as a function parameter.
2. If a member variable is shared by multiple threads, but it is an integral type, enumeration type, or pointer type, and only require limited arithmetic operations, such as "++", "- -", it may be converted to a TBB atomic variable. Atomic variables allow atomic operations, which have low overhead compared to locks. Atomic operations do not suffer race conditions or deadlocks. For example, mTotoalHairs, counts the total number of generated hairs. The only operation for that variable is "++" and it can be converted to an atomic variable as follows: (**Figure 14**)

```
tbb::atomic<int> mTotalHairs;
```

Figure 14

3. For all other shared member variables, a mutex is needed to protect them to avoid race conditions. If a mutex is in a high thread contention area, it may become quite expensive. In case of the fur shader there was no need for such a mutex since logically each hair was processed independently.

As mentioned earlier, the DreamWorks Animation fur shader depends on various other libraries, such as the geometry evaluation library. Some of these libraries contained static variables. For example, in the NURBS surface evaluation function, a static variable fPrecision controls the precision of evaluation. One could pass fPrecision as a function argument to avoid race conditions; however, such libraries are usually widely used by different applications and it is very time consuming to change the function interfaces to include a new parameter. In such cases, use of a thread local storage variable may be a good choice. Converting to thread local variables is straightforward—just add a prefix "__thread" to the declaration of that variable. For example:

```
__thread float fPrecision;
```

Figure 15

Thread-local storage variables can be accessed just like regular variables, and the compiler will maintain a separate copy of this variable for each thread. However, caution must be exercised when using thread local variables. First, there may be subtle hidden performance costs associated with the use of thread local variables (see blog: http://software.intel.com/en-us/blogs/2011/05/02/the-hidden-performance-cost-of-accessing-thread-local-variables/). Second, using large arrays as thread-local variables should be avoided since these would be allocated on the thread stack by the compiler, and could reduce available stack size.

Improving scalability with Intel® VTune™ Amplifier XE

Once the threaded fur shader ran correctly after data races and thread safety issues were fixed, the next step was to evaluate the performance. In particular, it is important to study how performance scaled with number of threads/cores. With the TBB task scheduler, it is very easy to control the number of threads to run the application. For example, one could code as follows:

```
tbb::task_scheduler_init init(nThread);
```

Figure 16

This scaling experimentation with the fur shader revealed a surprise—it did not scale well with the number of CPU cores at all. For example, the fur shader reached ~2x scaling on 2 cores but with 8 cores/threads the scaling did not improve beyond 2x.

Intel VTune Amplifier XE is a great tool to help find such thread scaling performance issues. In particular, it has a "locks and waits" analysis, which can identify an important cause for ineffective processor core utilization. One common performance problem is for threads to

/usr/pic1/AmplifierXE_Projects/texture_lock - Intel VTune Amplifier XE 2011

File View Help

r007lw

Locks and Waits - Locks and Waits

Intel VTune Amplifier XE 2011

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Top-down Tree

Grouping: Function / Call Stack

Function / Call Stack	Wait Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Wait Count	Spin Time	Module
FurShader::Fur::evaluateMaps	1058.409s	219,970	2427.596s	
FurShader::Fur::generateMSFurLoop	488.259s	89,649	1011.185s	
rml::internal::thread_monitor::commit_wait	52.734s	3,535	0s	
[TBB Dispatch Loop]	6.000s	1,208	6.130s	
	1.738s	331	1.391s	
	0.055s	1,968	0.071s	
	0.031s	3,517	0.021s	
	0.023s	419	0.010s	
	0.017s	66	0.010s	
	0.014s	238	0.010s	
	0.004s	1	0s	
	0.004s	87	0s	
	0.002s	5	0s	
Selected 0 row(s):	[Unknown]	[Unknown]	[Unknown]	[Unknown]

0 stack(s) selected. Viewing
Current stack is 0.0% of selection
No stack information

Thread (0x7e75)
TBB Worker Thread
TBB Worker Thread
TBB Worker Thread
TBB Worker Thread
TBB Worker Thread
TBB Worker Thread
TBB Worker Thread
Thread Concurre...

50s 100s 150s 200s 250s 300s 350s 400s 450s 500s

Thread
Running
Waits
Transitions
Thread Conc...
Concurre...

No filters are applied. Module: [All] Thread: [All] Call Stack Mode: Only user functions

Figure 17: Intel® VTune™ Amplifier XE showing thread contention in texture sampling function evaluateMaps (yellow lines showing lots of transitions between threads)

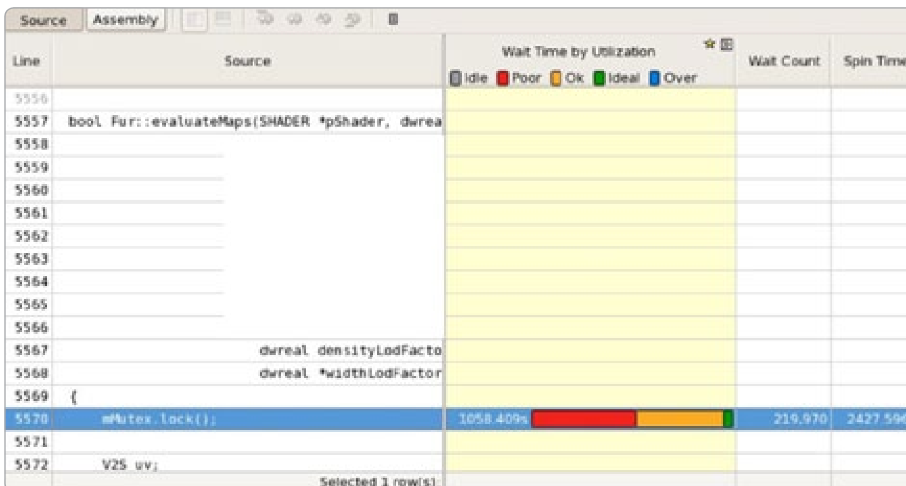


Figure 18: Drilling down on function evaluateMaps in Figure 17 shows the source code and the lock location where contention occurred

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

wait on synchronization objects such as locks or for threads to contend for access to a protected/locked resource. In the case of the fur shader, "locks and waits" analysis (see **Figures 17** and **18**) showed that the mutex in the texture sampling library was the most contentious and expensive one.

In DreamWorks Animation's texture sampling library, there is a cache to save the results from last texture sampling call. If the texture coordinates from current texture sampling call are the same as the last call, the sampling function simply returns the texture values saved in the cache. This cache is protected with a mutex to make it thread safe. However, in the workload tested, the texture sampling function was called very frequently, resulting in a high thread contention for texture cache access. And, this thread contention became the hotspot. In this particular instance, it was easy to eliminate the contention by removing the mutex and making the texture cache a thread-local storage variable.

In general, for large legacy codes, in order to introduce threading quickly, one may have to use mutexes to protect entire functions if it is not clear whether the functions are thread safe or not. The assumption would be that these legacy functions are not the performance bottlenecks. If Intel VTune Amplifier XE profiling shows that a mutex is a hotspot, then a further investigation would be needed to design a better solution. Following this approach, developers are made to always focus on performance-critical areas in parallelizing a large legacy application.

Conclusion

This paper highlights different ideas that enabled parallelization of DreamWorks Animation's fur generation for multicore processors while still continuing to use various legacy libraries needed in production. Performance evaluation of this parallel code on DreamWorks Animation's production shots on an 8-core Intel® workstation produced an average of 5x speedup on the fur generation loop. Overall speedup for the fur shader was 4x due to other serial regions in the execution path.

Intel® tools were critical in every step of this parallelization effort with DreamWorks Animation's fur shader: Intel Threading Building Block's (TBB) `parallel_for` construct was used to parallelize the fur generation loop, Intel Math Kernel Library's (MKL) random number generation functions were used to ensure that the result from parallelized fur generation was deterministic and independent of the number of threads used, Intel Inspector XE and Intel compiler's code coverage tools helped find all relevant thread safety issues, and Intel VTune Amplifier XE helped find/fix thread scalability issues. Without these Intel tools, parallelizing DreamWorks Animation's fur shader would have been a mission impossible. □

BLOG highlights

Doctor Fortran Gets Explicit—Again!

STEVE LIONEL, Developer Products Division

Nearly 11 years ago (!) I wrote an item for the *Visual Fortran Newsletter* on explicit interfaces in Fortran. In recent weeks, I have had to refer quite a few customers to this article, suggesting that many Fortran programmers don't understand the role and rules of explicit interfaces. However, when I reread the item, I realized that things had changed a bit since Fortran 95, so I figured it was time to revisit the issue.

In Fortran speak, an "interface" is information about a callable procedure. Fortran 77 had only "implicit interface" where the only thing you could say about a procedure was the datatype of a function result. While the language said that arguments in a call must match in number, order, type, and rank (number of dimensions), there was no way to describe the arguments so that the compiler could check them. Furthermore, a compiler did not need to know these things because it didn't affect how arguments were passed.

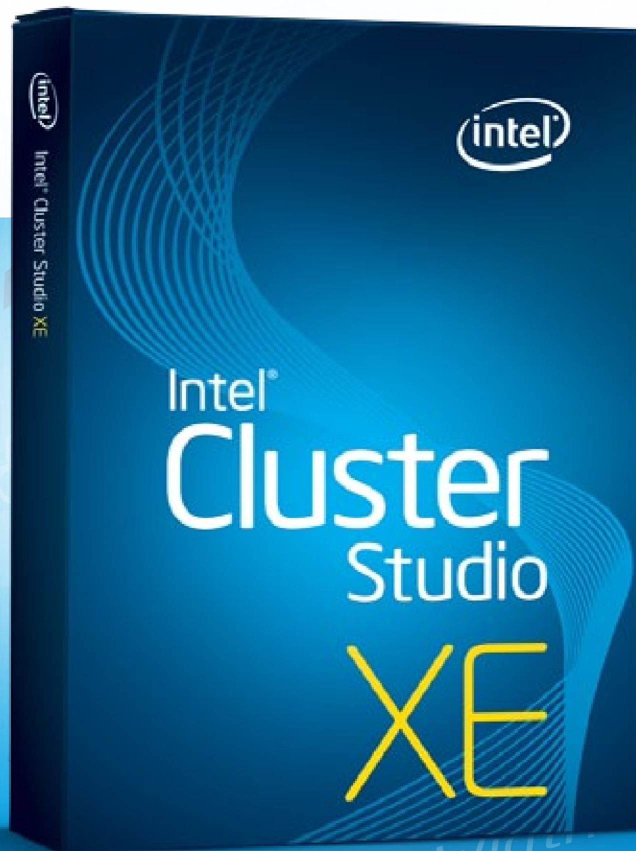
Enter Fortran 90. Suddenly, things get a lot more complicated. For example, a dummy argument could be an assumed-shape array, requiring the call to supply information about the array bounds....

SEE THE REST OF STEVE'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 Analysis
Intel® Cluster



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

Tools in Studio XE

by David Mackay, Ph.D. and
Krishna Ramkumar,
Software and Services Group, Intel Corporation

Intel® Cluster Studio XE offers new analysis tools—Intel® Inspector XE and Intel® VTune™ Amplifier XE—as an XE bonus, in addition to updates to tools previously found in Intel® Cluster Studio. As many software applications adopt a hybrid distributed/shared memory programming model, the addition of node software analysis capabilities is particularly useful. Now you can analyze what is happening in the shared memory section of your code, in addition to the internode interaction capabilities that have been supported for years.

VTune Amplifier XE and Intel Inspector XE provide detailed information, respectively, about software performance and correctness of a system. You can run these tools across a cluster to analyze your hybrid parallel application for correctness and performance, and efficiently understand what is going on in each node.

This article provides a brief sample analysis of an Intel® MPI program with VTune Amplifier XE. It also explores typical usage scenarios designed to give developers a sense of the tools' benefits in a cluster environment.

Sign up for future issues | Share with a friend

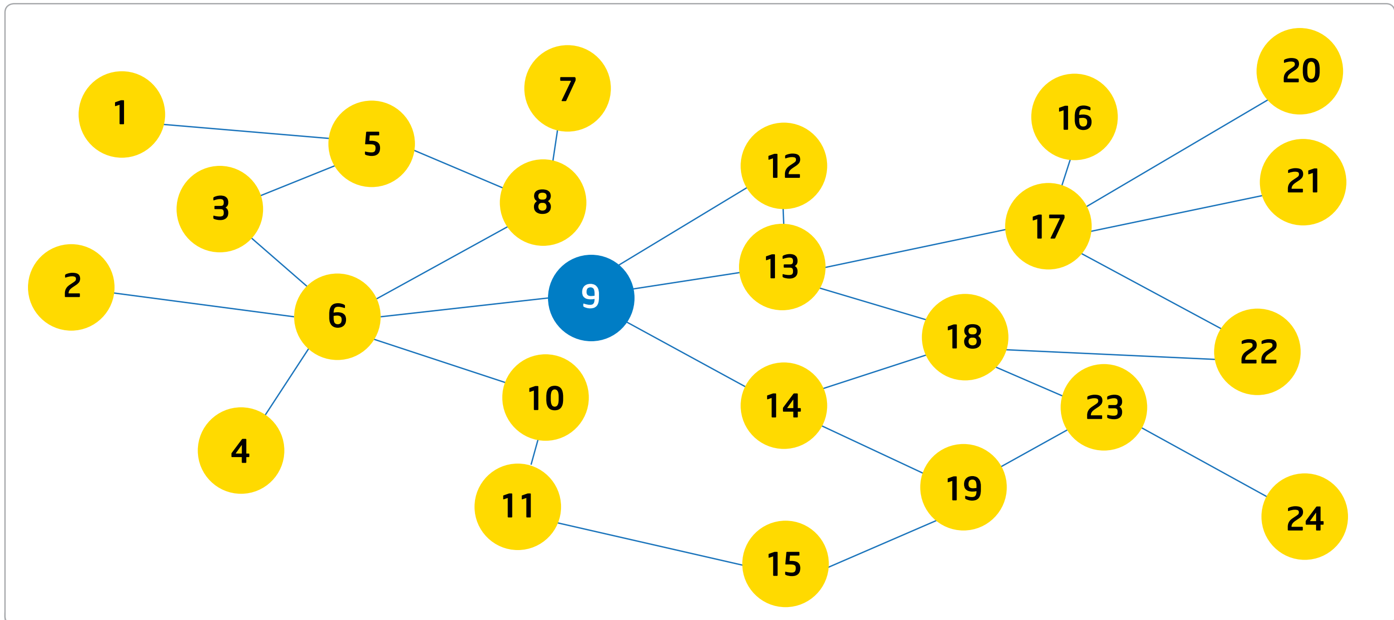


Figure 1. Sample graph abstraction

Intel Cluster Studio XE

The new Intel Cluster Studio XE adds the popular analysis tools Intel Inspector XE and the VTune Amplifier XE to the successful Intel Cluster Studio (containing Intel® MPI library, Intel® compilers, Intel® Threading Building Blocks (Intel® TBB), Intel® Math Kernel Library (Intel®MKL), and Intel® Trace Analyzer and Collector). In this paper, we illustrate the use of VTune Amplifier XE on a hybrid distributed/shared memory cluster application for tuning performance. An article in a future edition of *The Parallel Universe* will cover the usage of Intel Inspector XE for finding programming defects.

Intel MPI has long been a popular method for solving large, technically difficult problems on distributed memory compute clusters. As the industry shifted to multicore processors, many developers initially just ran one MPI process per MPI node. However, hybrid parallel programming is becoming more popular for performance-critical applications, especially as core count increases. In this model, distributed/shared memory constructs are used to implement two levels of parallelism, because the computation can now be distributed across the available MPI nodes. In addition, there can be threading per MPI node to make full use of all the available cores per MPI node. This has several advantages. First, many applications have significant amounts of data that each process needs to read or access for computations. As the number of cores per node on the compute cluster increases, each MPI process ends up with its own copy of the memory. This is an inefficient use of system resources, such as memory. Hybrid parallel programming shares memory on the node, and the threads share a common memory heap and can easily share a common memory data set. A second advantage is the ability to execute dynamic task stealing or task balancing on the node. It is notoriously difficult to shift tasks or work across MPI processes once they are already assigned. Intel's shared memory programming models, such as Intel TBB and Intel® Cilk™ Plus, offer task stealing under the hood and can automatically help provide a good workload balance among threads on a compute node.

Here, we illustrate the use of VTune Amplifier XE on a hybrid distributed/shared memory cluster application. The sample application is an algorithm that computes the ***Betweenness Centrality*** metric on large, real-world datasets. Betweenness Centrality of a vertex is a shortest path-based metric, which is defined as the ratio of the number of shortest paths (between all vertex pairs in a graph) passing through the vertex to the total number of shortest paths (again between all vertex pairs in a graph). For example, a high value of Betweenness Centrality for a vertex indicates that the vertex is important in a communications network (as generally communication is desired through the shortest path). The algorithm computes shortest paths from every vertex to all other vertices in the graph using breadth first traversals (we only consider undirected, unweighted graphs in this paper). For more details about the algorithm including pseudo code, please refer to Algorithm 1 in in *Parallel Algorithms for Evaluating Centrality Indices in Real-world Networks.*[1]

In **Figure 1**, a sample graph network is depicted with vertices labeled. In this case, vertex nine has the highest Betweenness Centrality value. This might be useful to know if each of the above vertices were to represent different cities in a transportation network or routers in a telecommunication network.

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

"MPI programs can now be tuned more precisely and easily with the insights available from Intel® Cluster Studio XE."

We begin with distributed parallelism using Intel MPI to implement this algorithm. As a next step, we add shared memory parallelism using Intel TBB. Specifically, we use Intel TBB's `parallel_for` framework to parallelize the main loop in the algorithm to improve performance. Inside the `parallel_for` we use an Intel TBB scoped spin mutex to ensure data coherency and avoid data races (look for a future article on using Intel Inspector XE to find data races in multithreaded programs).

The source code showing the Intel TBB `spin_mutex` is shown below (**Figure 2**).

```
if(w != node)
{
// scoped_lock is used here that protects code within this'if' block
spin_mutex::scoped_lock lock(bwMutex);
//betweennessCentrality is the global data structure
betweennessCentrality[w] = betweennessCentrality[w] + dependency[w];
}
```

Figure 2

"Intel's shared memory programming models, such as Intel® Threading Building Blocks and Intel® Cilk™ Plus, offer task stealing under the hood and can automatically help provide a good workload balance among threads on a compute node."

"Hybrid parallel programming is becoming more popular for performance-critical applications, especially as core count increases."

Sync Object / Function / Call Stack	Wait Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Wait Count	Spin Time
tbb::spin_mutex 0x9187e748	350.712s	85,186	1097.028s
Condition Variable 0x8161ca88	34.858s	23	0s
TBB Scheduler	0.764s	6	0.760s
Socket 0x9a1d6107	0.041s	12	0s
Stream CA-GrQc.txt 0x484fce50	0.001s	2	0s
Signal	0.000s	1	0s

Figure 4. Screenshot showing wait time on sync objects reported by Intel® VTune™ Amplifier XE

Sync Object / Function / Call Stack	Wait Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Wait Count	Spin Time
tbb::spin_mutex 0xe492abb1	328.116s	74,396	902.361s
computeBetweennessCentrality	328.080s	72,895	887.346s
ApplyAlgorithm::operator() ← [TBB parallel_for o	328.080s	72,895	887.346s
operator new	0.036s	1,501	15.015s
tbb::parallel_for<tbb::blocked_range<int>, Apply	0.036s	1,501	15.015s
Condition Variable 0x8161ca88	101.582s	23	0s
TBB Scheduler	4.653s	4	4.640s
Socket 0xdeb62f75	0.041s	12	0s
Stream CA-GrQc.txt 0xff615f84	0.001s	3	0s

Figure 5. Screenshot showing expansion of top sync objects

Line	Source	Wait Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Wait Count	Spin Time
236				
237	`if(w != node)`			
238	`{`			
239	`spin_mutex::scoped_lock lock(betweennessMutex);`	328.080s	72,895	887.346s
240	`betweennessCentrality[w] = betweennessCentrality[w] + dependency[w];`			
241	`}`			
242	`}`			
243	`}`			

Figure 6. Screenshot showing expansion of top sync objects

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

When we build this application we use the –DTBB_USE_THREADING_TOOLS option so that the analysis tools will recognize all of the Intel TBB constructs, and display more meaningful information when displaying the analysis results. To analyze this application for performance we use VTune Amplifier XE. The simple "hotspots" analysis shows us which routines consume the most compute time. The "locks and waits" analysis shows us where threads are stalled waiting for shared resources. In this case, we are going to conduct a locks and waits analysis so we invoke VTune Amplifier XE collection with the following command line:

```
mpirun -f mpd.hosts -perhost 1 -n 1 am-
plxe-cl -collect locksandwaits -r gmet-
rics_lw_r1 -- ./graph_metrics CA-GrQc.txt
```

Figure 3

In this mode, VTune Amplifier XE will produce a results data for each node in the MPI cluster and appends the node rank number to the results data, so we can distinguish results output from each node. For our screenshots we use data from node 0. The node rank comes from the MPI runtime and VTune Amplifier XE takes the values from the environment settings PMI_RANK or PMI_ID (these are automatically set by Intel MPI). In this analysis, we chose a workload with 5,242 vertices and 28,980 edges). When we open results in VTune Amplifier XE, we have the following display in **Figure 4**.

This shows a significant spinning on the Intel TBB spin_mutex . We click on the + box on the `tbb::spin_mutex` line to expand our display and we see the results shown in **Figure 5**.

Figure 5 shows us that it is `the Intel TBB spin` mutex around the routine `computeBetweennessCentrality` that is the bottleneck. This shows us where to concentrate our attention. Now when we double-click on this line or right-click and select a source view we obtain the view in **Figure 6**.

The results in Figure 6 show us exactly where the thread contention is occurring. This tells us we need to think of a better solution to minimize the thread contention. We recognize that it is a global variable update operation and that an Intel TBB atomic variable will be a more elegant solution. So, we remove the Intel TBB spin_mutex and create an Intel TBB atomic variable that will not require a spin_mutex, but preserve the thread safety required (**Figure 7**).

```
map<int, atomic<double> >
betweennessCentrality;
```

Figure 7

BLOG highlights

"Award-Winning" Intel® Parallel Studio XE

JAMES REINDERS,
Director of Software Development Products

Intel® Parallel Studio XE, in the category of "Best HPC software product or technology," was honored in the annual HPCwire Readers Choice Awards. The awards are an annual feature of the publication and constitute prestigious recognition from the high-performance computing community. The awards were announced and presented during the 2011 Supercomputing Conference, held in Seattle, WA. As HPCwire proclaimed, "The annual awards are highly coveted as prestigious recognition of achievement by the HPC community."

Intel Parallel Studio XE 2011 combines enhanced optimizing compilers, libraries, error checkers, and performance analyzers in a single integrated suite that enables developers to write faster, more reliable and secure code on Windows* and Linux*.

I'm very pleased to see this additional recognition for our Intel Parallel Studio XE, which is used by a very large number of developers around the world. We just recently offered an extended version of Intel Parallel Studio XE, for developers on cluster computers (characterized in part by their use of MPI), called Intel® Cluster Studio XE....

SEE THE REST OF JAMES' BLOG:

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

After this change, we compared the two algorithms (one with a scoped mutex and the other with an atomic variable on a 16 node Linux* cluster. Each node is a 64-bit 2-socket Intel® Xeon® CPU x5680 machine running at 3.33 GHz with 6 physical cores per socket with hyperthreading enabled (384 TBB threads). We ran a data set with 36,692 vertices and 367,662 edges. The program execution time with the spin_mutex was 284 seconds; when we adopted the atomic variable execution time dropped to 222 seconds—**a 22 percent performance improvement**.

The performance analysis tool VTune Amplifier XE can help analyze the performance of hybrid parallel applications. In general, we recommend tuning software on a single node: a) Make sure the software is vectorized using SIMD instructions such as SSE4 or AVX, b) Add the MPI constructs and use Intel® Trace Analyzer and Collector to improve the message passing performance, c) Add shared memory parallelism for improved performance on each node and, d) Run Intel Inspector XE to remove possible non-deterministic results, and then use VTune Amplifier XE to look for performance bottlenecks.

Summary

The ability to analyze the detailed activities on the nodes of a distributed program with VTune Amplifier XE offers much needed insights for the world of hybrid programming. VTune Amplifier XE can help with hybrid application regardless of the shared-memory programming model utilized in the hybrid—Intel TBB, Intel Cilk Plus, OpenMP*, Coarray Fortran, pthreads or Windows* threads. MPI programs can now be tuned more precisely and easily with the new tools available in Intel Cluster Studio XE. ☐

References

1. D.A.Bader, K. Madduri, "Parallel Algorithms for Evaluating Centrality Indices in Real-world Networks," The 35th International Conference on Parallel Processing (ICPP 2006), Columbus, OH, August 14–18, 2006.
2. Stanford Large Network Dataset Collection - http://snap.stanford.edu/data/index.html
3. http://software.intel.com/en-us/articles/intel-cluster-studio/
4. http://software.intel.com/en-us/articles/intel-parallel-studio-xe/

"Intel® VTune™ Amplifier XE provides detailed information about software performance on a system."

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

RESOURCES AND SITES OF INTEREST

Go Parallel

The mission of Dr. Dobb's 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

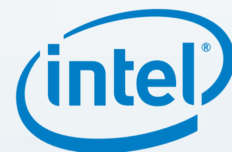
Free updates and fast downloads on even more new software technologies, tools, and best practices for smart coding and innovative user experiences.

> Join Intel® Software Dispatch.

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.

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



Intel® Learning Lab

The ultimate training resource for serial and parallel programming for multicore.
Demos, videos, articles, quick tips, and more.

Application Tuning

Assess the recommended methodology for using **Intel® Vtune™ Amplifier XE** to tune software on Intel® microarchitecture in this video.

© 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.

Software

Stay pulsed on the latest parallel techniques, tools and technologies with Intel® Software Insight.

Innovate at the forefront of software design, coding, and deployment. Join thousands of developers and IT pros worldwide who subscribe to Intel® Software Insight.

- Get the latest news in parallel programming, mobile technologies, and Open Source, as well as virtualization, security, and the cloud.
- Receive free tech downloads to help rev up your code.

Free updates. Free downloads.

From software experts directly to you.

Join Intel® Software Insight today!

softwaredispatch.intel.com

Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries. *Other names and brands may be claimed as the property of others. Copyright ©2012 Intel Corporation. All rights reserved. 0212/BC/RHM3/WQ