THE PARALLEL UNIVERSE

Issue 8
September 2011

HPC Study:

Biophysicists and Mathematicians Embrace Parallelism with Intel® Parallel Advisor
by Zakhar A. Matveev

Intel® Parallel Studio XE SP1 by Michael D'Mello

The Intel® Threading Building Blocks Flow Graph by Michael J. Voss, Ph.D.

"Intel® Parallel Advisor is the tool needed to answer your parallelism 'What If' questions. It will assist you in understanding how to maximize the benefits of your parallelism efforts."

KIRK HALLER, *Director of Research at Dassault Systems, SolidWorks Corporation*

Speed and simplify parallel development

Intel® Parallel Advisor pinpoints where your application can benefit most from threading—before you've wasted time, effort, and money.

The innovative threading assistant is the only step-by-step guide available for Microsoft Visual Studio* C developers and C++ developers who want to add threading to existing serial or parallel applications.

Rock your code. **Rock your world.**

LEARN MORE ABOUT INTEL PARALLEL ADVISOR

CONTENTS

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.

PARALLELISM PROGRAMMING:

WHO SIGNED ME UP FOR WRITING A BOOK?

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.

We have three very different articles for your consideration in this issue, and all of them are quite useful, but still compact.

One article is an overview of the capabilities of Intel® Parallel Studio XE SP1. If you do not already use all our tools, this may pique your interest. If you do use them all, this may highlight capabilities you will want to learn more about.

It is hard to choose a favorite article, especially in this issue, but I would vote for the feature article "HPC Study: Biophysicists and Mathematicians Embrace Parallelism with Intel® Parallel Advisor," simply because it is a real customer example. I'm a pushover for any article that shows the actual customer experience, step-by-step. Not only that, this article covers a pretty typical experience with Intel Parallel Advisor.

Around Intel, some people know me as "the guy who did not believe in Intel Parallel Advisor." True enough. For decades, I've seen tools try to make parallelism easy. It has hardened me into a skeptic down to the bone. In a way, I was right. I remain convinced no tool can magically look at your code and make it parallel. Today, parallelism still requires the skills of a human programmer.

But I'm actually no longer skeptical about Intel Parallel Advisor. So, what changed my mind? Answer: Intel Parallel Advisor works!

Intel Parallel Advisor changed the game. "Kobayashi Maru,[1]" you might say. Instead of trying to build a tool that magically did everything, we created a tool that assists you in your analysis of possibilities—like nothing before it could. It turns out this is magical because that is where we waste time!

Instead of trying to build a tool that magically did everything, we created a tool that assists you in your analysis of possibilities—like nothing before it could.

I admit that I've been there. I've had a bright idea on how to code something in parallel, only to code it up, debug it, and figure out that I missed a key reason it was not going to scale well. Of course, I wish I'd known that fact many weeks earlier. I'm left to decide if it is "good enough" or whether I wasted weeks of programming time. Intel Parallel Advisor makes that "good enough" question a thing of the past, because it lets us find those issues without weeks of programming and debugging. The HPC article shows you how that worked for one customer. I hope you give it a try when you need it, and see how well it can help you avoid dead ends. Give Intel Parallel Advisor a day of hard work on a real problem you have, and it just might save you weeks of unproductive dead ends while letting you really find the ways to scale an application. I've seen it help expert and novice alike. Fortunately, I did not make any serious bets with the development team about their ability to help experts and novice both, or I'd be out a lot of money now! I'm impressed with Intel Parallel Advisor because of what I've seen it do for our customers.

Finally, the article "The Intel® Threading Building Blocks Flow Graph" is about my favorite single new feature. My appreciation for Intel Threading Building Blocks (Intel® TBB) is hardly a secret, and this latest feature will give you some idea of why I continue to be a huge fan of Intel TBB and the team behind it. For some time now, users have asked for more interfaces to Intel TBB. Game developers, in particular, have asked for event-based interfaces to schedule tasks. Other developers have wanted interfaces for coordinating multiple dependent parts of an application. Others did not ask for anything, but proceeded to make our support team members amazed at what some developer would do with the pipeline functionality of Intel TBB, despite how convoluted it seemed to us. The Intel TBB design team outdid themselves, finding a unifying solution to all three of these interface needs with a general purpose "graph" interface. Internally, they called this tbb::graph for a long time including during early beta testing. Late in the game, before product release, they renamed it "flow graph." I've been told it is a better description of the feature and they've had positive feedback on the name change.

The concept of a group of tasks being organized with a dependency graph is very common in many applications. I first used it in compiler design, but have used it many times in signal processing applications such as radar system designs. Intel TBB's new flow graph is the right solution for such programs.

Oh, did I mention a book? I'm working with a couple of experts on a book about parallel programming. We have finally reached the point where we think we see the light at the end of the tunnel—we have written more than 75 percent of the text we need before we send it out to reviewers and the publisher. Will it be a groundbreaking book? We hope so. Time will tell. I'll try to update you next time on what our reviewers think, and tease you more with what will be inside it. For now, suffice it to say we are trying to put on paper what we have learned from working with our customers about how to best succeed at teaching parallelism. One way to explain what we think we have learned is this: computation is not everything because communication matters and you can learn a lot from knowing common patterns and having seen enough examples.

Our tools will help you succeed; I hope this issue gives you a few more ways to understand how they fit your needs.

Please do the authors the favor of reading their articles and keep the feedback coming! Your feedback and requests help guide future articles. I hope you enjoy this issue, and I look forward to hearing your thoughts.

Enjoy!

James Reinders
September 2011

P.S., As we go to press with this issue, news is out that support for Intel® Cilk™ Plus is going into a branch of gcc for evaluation by the gcc community. We have contributed our implementation to open source to help this effort. We are very excited about that. Of course, we will continue to support Intel Cilk Plus in our compiler and tools; having gcc support will just make Intel Cilk Plus better supported and a better way to go with coding. I'll blog on that more, and talk about it in a future issue of this magazine. Just like the popular success Intel TBB enjoys because it is supported widely, we look forward to growing support for Intel Cilk Plus.

1. The no-win Starfleet* training scenario, as seen in the Star Trek* films.

Sign up for future issues | Share with a friend

HPC Study: Biophysicists and Mathematicians Embrace Parallelism with Intel® Parallel Advisor

by Zakhar A. Matveev
Software Development Engineer

Learn how a group of research scientists in Russia parallelized their applications in response to the growing data from biological experiences and increasing complexity of simulation requirements.

Biophysics explorations are increasingly associated with high-performance computing (HPC). The use of computer modeling and simulations is becoming biology's "next microscope." The huge amount of heterogeneous data that biophysics applications must process requires high-performance computing techniques. The complex interaction of biological "units" increases the complexity of the model, and eventually makes it practically impossible to organize efficient computational processing in this area without using parallelism.

A group of research scientists at Nizhny Novgorod National Research University in Russia, including Professor G. Osipov Ph.D., V. Petrov Ph.D., and M. Komarov Ph.D., have been developing biophysics compute-intensive simulation applications for years. Because of the growing data from biological experiments and increasing complexity of simulation requirements, research scientists have been faced with significant computational challenges in this area. To improve the performance of the algorithms, they decided to parallelize their applications using Intel® Parallel Advisor 2011 (now available with the purchase of Windows* version of Intel® Parallel Studio XE).

Originally there were three different serial C++ applications to be parallelized:

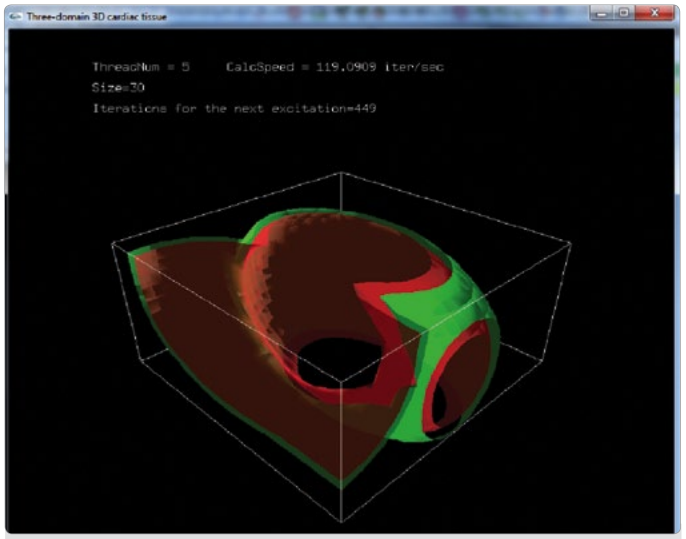
> CARDIAC: Three-domain cardiac 3-D simulation
> NEURAL: Brain neural ensembles dynamic analysis
> RATE: Phenomenological rate model to organize virtual mobile robot control

In this article we will discuss each of them to see how research scientists parallelized their code with the help of Intel® software tools.

Three-domain cardiac simulation serial application

Electromechanical cardiac simulation models are widely used to interpret medical data and test hypotheses about arrhythmia mechanisms. In the CARDIAC application, the Nizhny Novgorod scientists used a newly proposed three-domain model that takes into account not only cardiac muscle cells (cardiac myocytes) as usual, but also extracellular space and small passive cardiac cells called fibroblasts. More complex real-world modeling means more complex numerical computations.

The output of CARDIAC is a spatiotemporal distribution of myocyte and fibroblast voltage levels V (x,y,z,t) over a 3-D volume and time interval [0, T], as shown on the CARDIAC benchmark GUI in **Figure 1**.



```
ThreacNum = 5     CalcSpeed = 119.0909 iter/sec
Size=30
Iterations for the next excitation=449
```

Figure 1: CARDIAC benchmark application GUI. Myocytes voltage level is given in red color, while fibroblasts voltage is given in green.

```
#1    for ($t_i$) // time ([0, T]) integration outer loop
#2        for ($P_i$ = {x,y,z}) // space (3D volume) iteration inner loop
              //calculates total extracellular current in $P_i$

          //Poisson solver:
          while (err > err0)
#3            for ($P_i$ = {x,y,z}) // space iteration inner loop
                      //calculates total extracellular current in $P_i$

#4        for ($P_i$ = {x,y,z}) // space iteration inner loop
                  //calculates ionic currents in $P_i$

#5        for ($P_i$ = {x,y,z}) // space iteration inner loop
                  //calculates myocyte voltage in $P_i$

#6        for ($P_i$ = {x,y,z}) // space iteration inner loop
                  //calculates fibroblast voltage in $P_i$
```

Figure 2: CARDIAC application program structure

The algorithm involves four key steps:

1. Find the total extracellular current (requires numerical calculation of discrete Laplace operator using five-point central approximation).
2. Solve the Poisson equation using an iterative scheme.
3. Calculate ionic currents through the cell using the biologically-relevant Luo-Rudy model for myocytes and recently proposed Sachse model for fibroblasts. In terms of numerical calculations it implies solving 15 nonlinear ordinary differential equations describing the dynamics of a single cardiac cell.
4. Solve the partial differential equations to find the fibroblast and the myocyte voltage levels.

Sign up for future issues | Share with a friend

The spatiotemporal nature of the problem suggested this nesting of loops in the serial program structure, as seen in **Figure 2**.

In the next section, we will look how this program has been transformed to execute calculations in parallel.

CARDIAC parallelization case study

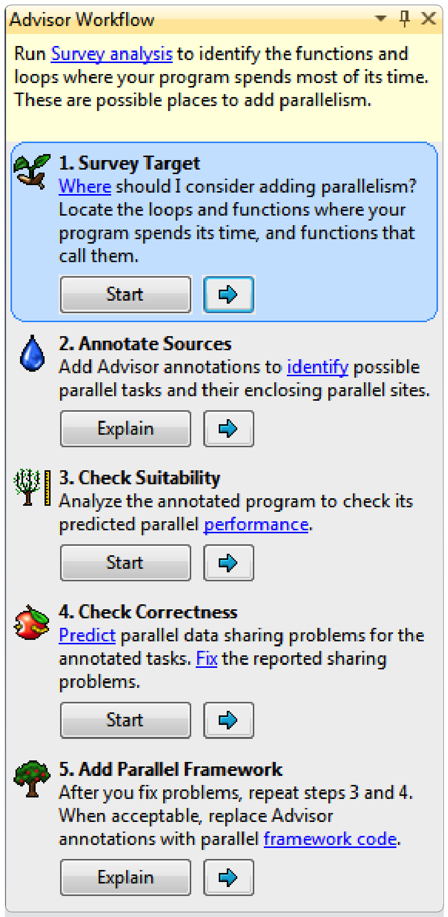
Conceptually, Intel® Parallel Advisor can be considered as a combination of several interrelated analysis tools and a proven methodology for adding parallelism to applications. (Methodology tends to be quite a natural thing for scientists.) The methodology is explicitly exposed by Intel Parallel Advisor in the workflow seen in **Figure 3**.

CARDIAC developers followed the workflow "step one" prompt and used the survey tool, which runs the application and profiles it, as seen in **Figure 4.** As expected, the survey reported that the outer time integration loop took approximately 100 percent of total execution time. However, executing this kind of loop in parallel is typically impossible because of the strong dependency between iterations (this loop is a refinement process, where the `i+1` iteration is a function of iteration `i`). Therefore, inner loops with iteration over space were of the most interest.

As the survey tree automatically expanded node accentuates in **Figure 4**, the myocyte integration loop #5 looked like the best candidate amongst the inner loops. Developers marked the corresponding code region (site) by inserting an Intel Parallel Advisor macro (annotation) for future analysis by other Intel Parallel Advisor tools.

(Note: In terms of Intel Parallel Advisor annotations terminology, "site" is a code region to parallelize, whereas "task" is a program extent to be executed in parallel with other task instances. One example is a typical loop parallelization where single-loop iteration corresponds to the task instance, while the overall loop is considered to be a site.)

But is it enough to parallelize only one 96 percent total time loop? Amdahl's law says that even relatively small pieces of serial execution might affect your performance for a large number of threads. Thus, the scientists decided to try several more places for parallelization. This is easy to do because the Intel Parallel Advisor annotations, as opposed to real parallel framework code, enable you to experiment with different parallelization approaches in a simple and safe way. To identify other parallel "hotspots," they used the survey source feature to provide loop time metrics (**Figure 5**)—this clearly suggested the second and third candidates


Advisor Workflow

Run Survey analysis to identify the functions and loops where your program spends most of its time. These are possible places to add parallelism.

1. Survey Target
Where should I consider adding parallelism? Locate the loops and functions where your program spends its time, and functions that call them.

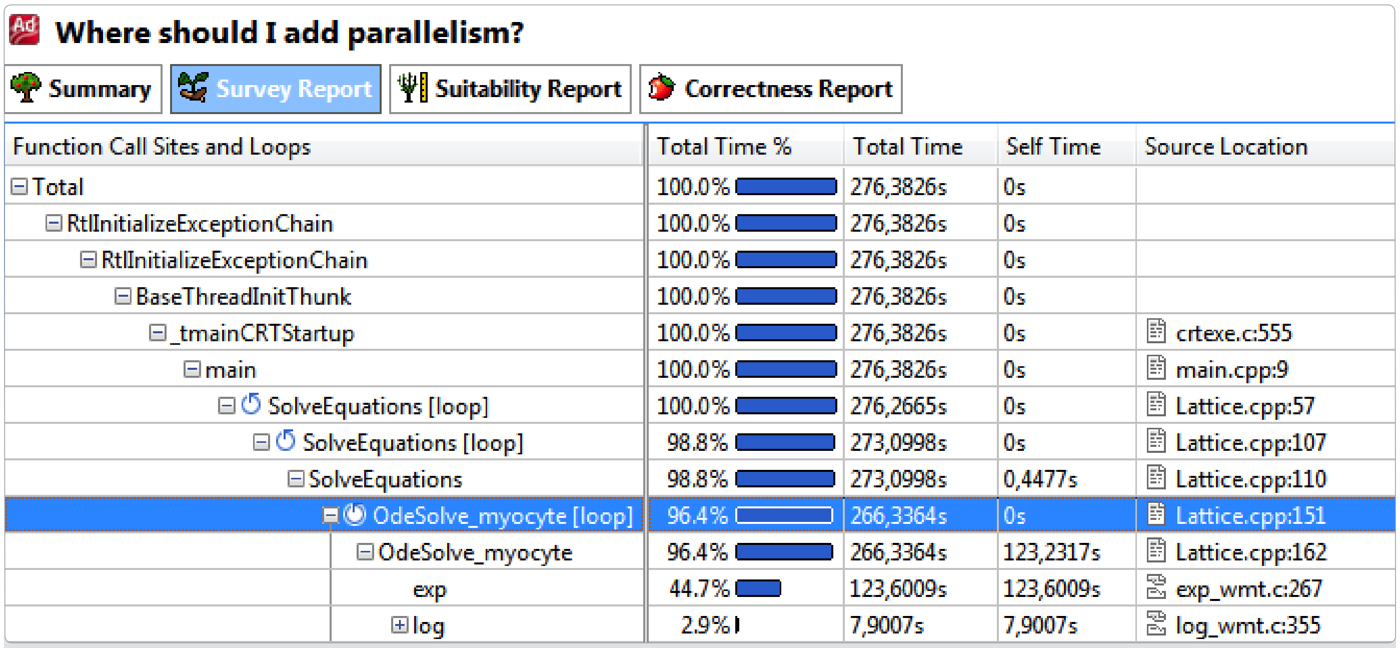
2. Annotate Sources
Add Advisor annotations to identify possible parallel tasks and their enclosing parallel sites.

3. Check Suitability
Analyze the annotated program to check its predicted parallel performance.

4. Check Correctness
Predict parallel data sharing problems for the annotated tasks. Fix the reported sharing problems.

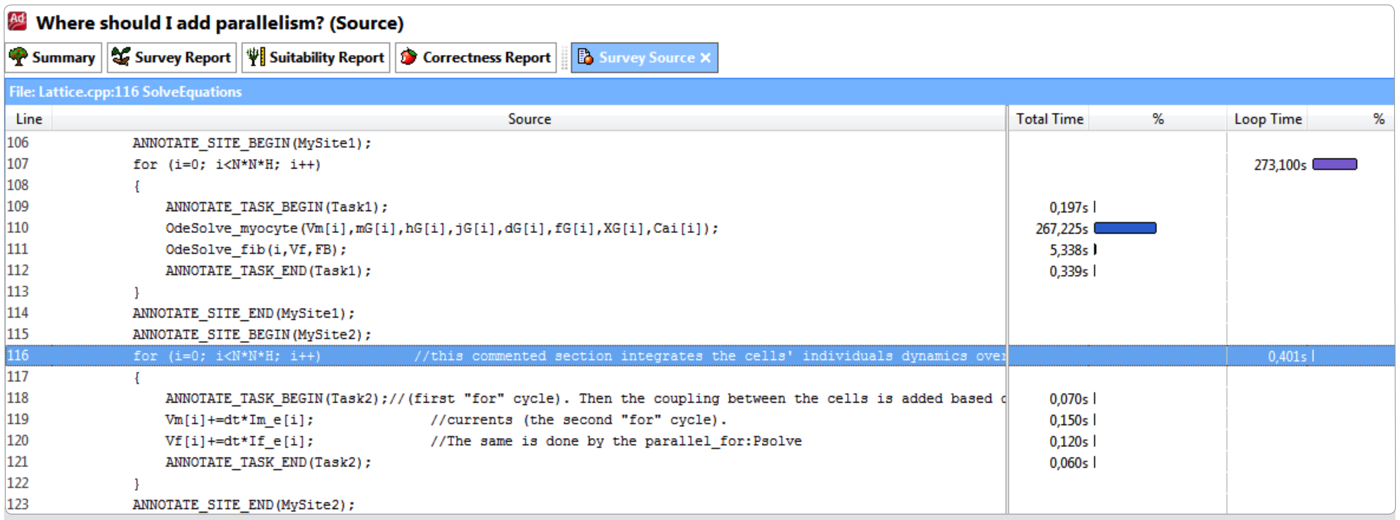
5. Add Parallel Framework
After you fix problems, repeat steps 3 and 4. When acceptable, replace Advisor annotations with parallel framework code.

Figure 3: Intel® Parallel Advisor workflow GUI guides you through parallelization steps.


Where should I add parallelism?

Function Call Sites and Loops	Total Time %	Total Time	Self Time	Source Location
⊟Total	100.0%	276,3826s	0s	
⊟RtlInitializeExceptionChain	100.0%	276,3826s	0s	
⊟RtlInitializeExceptionChain	100.0%	276,3826s	0s	
⊟BaseThreadInitThunk	100.0%	276,3826s	0s	
⊟_tmainCRTStartup	100.0%	276,3826s	0s	crtexe.c:555
⊟main	100.0%	276,3826s	0s	main.cpp:9
⊟SolveEquations [loop]	100.0%	276,2665s	0s	Lattice.cpp:57
⊟SolveEquations [loop]	98.8%	273,0998s	0s	Lattice.cpp:107
⊟SolveEquations	98.8%	273,0998s	0,4477s	Lattice.cpp:110
⊟OdeSolve_myocyte [loop]	96.4%	266,3364s	0s	Lattice.cpp:151
⊟OdeSolve_myocyte	96.4%	266,3364s	123,2317s	Lattice.cpp:162
exp	44.7%	123,6009s	123,6009s	exp_wmt.c:267
⊞log	2.9%	7,9007s	7,9007s	log_wmt.c:355

Figure 4: Intel® Parallel Advisor survey report for CARDIAC application



File: Lattice.cpp:116 SolveEquations

```
Line  Source                                                                                                         Total Time  %   Loop Time  %
106   ANNOTATE_SITE_BEGIN(MySite1);
107   for (i=0; i<N*N*H; i++)                                                                                                          273,100s
108   {
109       ANNOTATE_TASK_BEGIN(Task1);                                                                                0,197s
110       OdeSolve_myocyte(Vm[i],mG[i],hG[i],jG[i],dG[i],fG[i],XG[i],Cai[i]);                                      267,225s
111       OdeSolve_fib(i,Vf,FB);                                                                                     5,338s
112       ANNOTATE_TASK_END(Task1);                                                                                  0,339s
113   }
114   ANNOTATE_SITE_END(MySite1);
115   ANNOTATE_SITE_BEGIN(MySite2);
116   for (i=0; i<N*N*H; i++)         //this commented section integrates the cells' individuals dynamics over                        0,401s
117   {
118       ANNOTATE_TASK_BEGIN(Task2);//(first "for" cycle). Then the coupling between the cells is added based o     0,070s
119       Vm[i]+=dt*Im_e[i];              //currents (the second "for" cycle).                                       0,150s
120       Vf[i]+=dt*If_e[i];              //The same is done by the parallel_for:Psolve                              0,120s
121       ANNOTATE_TASK_END(Task2);                                                                                  0,060s
122   }
123   ANNOTATE_SITE_END(MySite2);
```

Figure 5: Intel® Parallel Advisor survey source view and loop time feature

(Poisson solver loop #3 and <time step * voltage> multiplication loop #6 in **Figure 2**). There was an additional, smaller spatial iteration loop inside of the Poisson solver. The scientists decided to merge it with the main solver loop to increase the granularity of loop body.

But how scalable is this parallelization idea? Is this really a good way to invest time and effort? The purpose of the Intel Parallel Advisor suitability tool is to provide quick answers to such questions by predicting approximate parallel performance for the code marked by annotations. Therefore, on the next step, the scientists ran the suitability tool; its forecast was encouraging for the most computationally loaded myocyte loop (with 22.79x expected speed-up on 32 cores, as seen in **Figure 6**) and quite cautious on other loops (with "bad" scalability near 2x for any target platform).

The Intel Parallel Advisor suitability tool also provided a strong recommendation to switch "Task Chunking" on when implementing the target parallel framework code. Fortunately, the CARDIAC scientists were planning to use Intel® Threading Building Blocks (Intel® TBB), which provides good support for task chunking.

The program was rebuilt using a debug configuration so it could be examined by the Intel Parallel Advisor correctness tool, which predicts possible parallel data sharing problems. Surprisingly, the correctness analysis identified data race and memory reuse problems and also pointed out the particular places in the code where a read/write communication problem occurs for

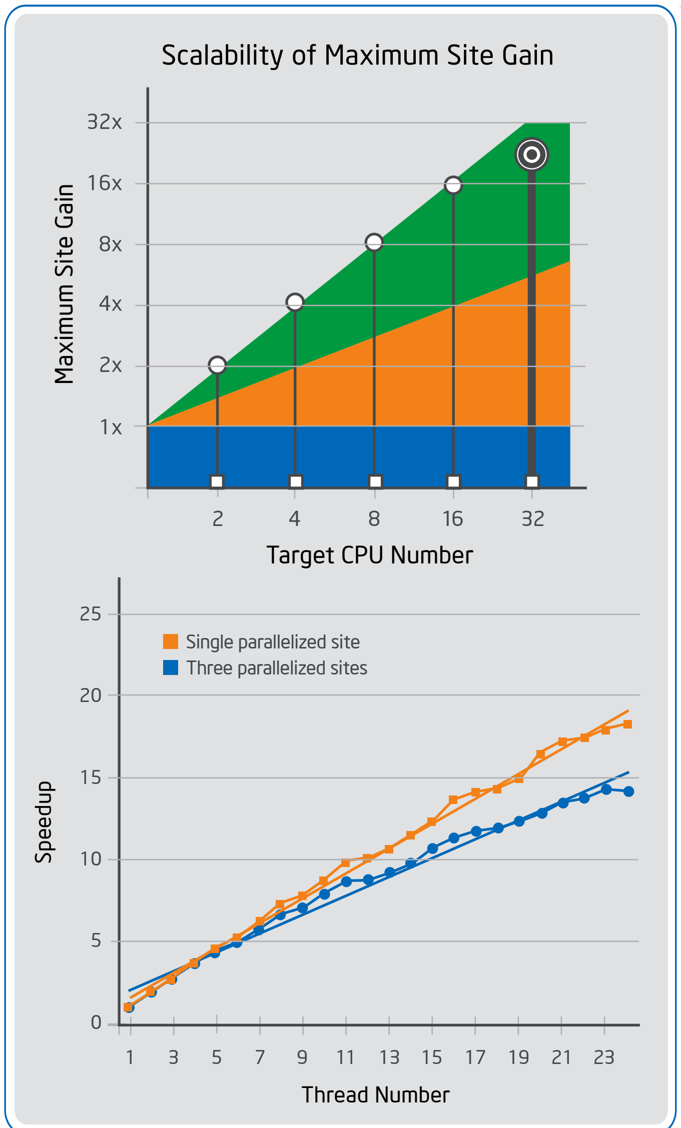
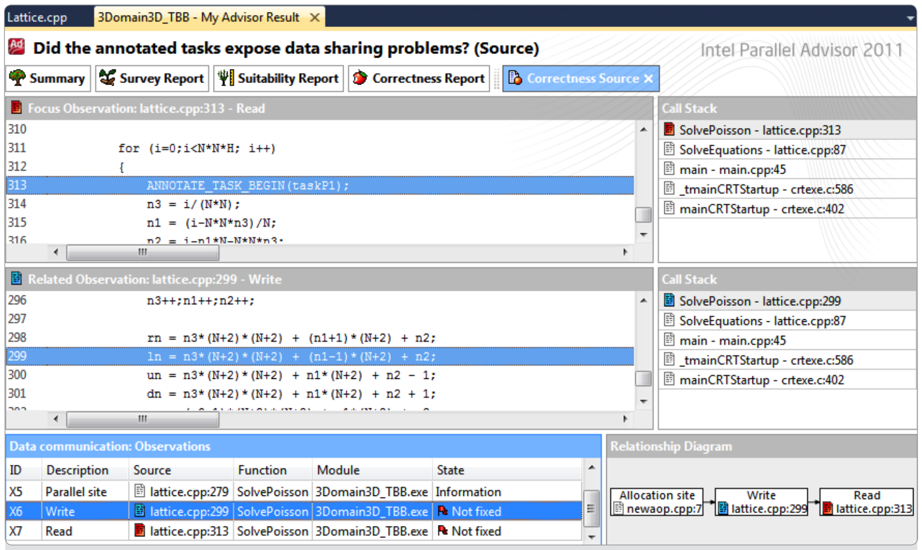


Figure 6: Intel® Parallel Advisor scalability forecast and experimental data for CARDIAC application (actual parallelization done with Intel® TBB)

Sign up for future issues | Share with a friend



Lattice.cpp | 3Domain3D_TBB - My Advisor Result

Did the annotated tasks expose data sharing problems? (Source) Intel Parallel Advisor 2011

Summary | Survey Report | Suitability Report | Correctness Report | Correctness Source

Focus Observation: lattice.cpp:313 - Read

```
310
311         for (i=0;i<N*N*H; i++)
312         {
313             ANNOTATE_TASK_BEGIN(taskP1);
314             n3 = i/(N*N);
315             n1 = (i-N*N*n3)/N;
316             n2 = i-n1*N-N*N*n3;
```

Call Stack

- SolvePoisson - lattice.cpp:313
- SolveEquations - lattice.cpp:87
- main - main.cpp:45
- _tmainCRTStartup - crtexe.c:586
- mainCRTStartup - crtexe.c:402

Related Observation: lattice.cpp:299 - Write

```
296             n3++;n1++;n2++;
297
298             rn = n3*(N+2)*(N+2) + (n1+1)*(N+2) + n2;
299             ln = n3*(N+2)*(N+2) + (n1-1)*(N+2) + n2;
300             un = n3*(N+2)*(N+2) + n1*(N+2) + n2 - 1;
301             dn = n3*(N+2)*(N+2) + n1*(N+2) + n2 + 1;
```

Call Stack

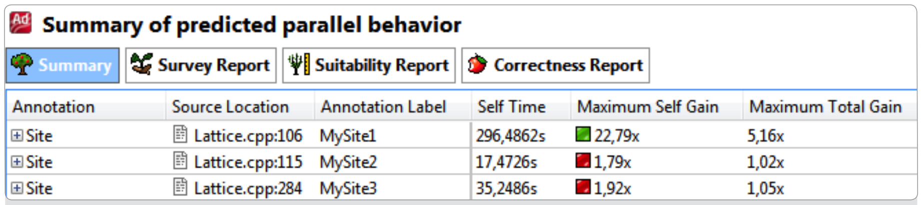
- SolvePoisson - lattice.cpp:299
- SolveEquations - lattice.cpp:87
- main - main.cpp:45
- _tmainCRTStartup - crtexe.c:586
- mainCRTStartup - crtexe.c:402

Data communication: Observations

ID	Description	Source	Function	Module	State
X5	Parallel site	lattice.cpp:279	SolvePoisson	3Domain3D_TBB.exe	Information
X6	Write	lattice.cpp:299	SolvePoisson	3Domain3D_TBB.exe	Not fixed
X7	Read	lattice.cpp:313	SolvePoisson	3Domain3D_TBB.exe	Not fixed

Relationship Diagram

Figure 7: Intel® Parallel Advisor correctness report for CARDIAC application



Summary of predicted parallel behavior

Summary | Survey Report | Suitability Report | Correctness Report

Annotation	Source Location	Annotation Label	Self Time	Maximum Self Gain	Maximum Total Gain
⊞ Site	Lattice.cpp:106	MySite1	296,4862s	🟩 22,79x	5,16x
⊞ Site	Lattice.cpp:115	MySite2	17,4726s	🟥 1,79x	1,02x
⊞ Site	Lattice.cpp:284	MySite3	35,2486s	🟥 1,92x	1,05x

Figure 8: Intel® Parallel Advisor annotations summary report

```
#1    for (ti) // time integration outer loop (tÎ[0, T]))

#2        for (ni) // iteration over N neuron network vertexes

#3            for (ni) // iteration over N neuron network vertexes
              //Calculates interaction with other neurons

          //Calculate given neuron characteristics
```

Figure 9: NEURAL application program structure

```
#1    for (ti) // time iteration outer loop

               // get sensor data from robot (x, y, angle, etc)

#2    for (ni) // iteration over N neuron network vertexes
               // calculate neuron states

#3    for (ni) // iteration over N neuron network vertexes
               // calculate neuron interactions

#4    for (ni) // iteration over N neuron network vertexes
               // update variables

      // send data to robot motor (x, y, angle, etc)
```

Figure 10: RATE application program structure

recently merged loops inside of the Poisson solver (**Figure 7**). It was quite easy to fix this problem by splitting loops back (i.e., avoiding execution of two interrelated loops in parallel). This adjustment did not affect the suitability performance forecast much, while the correctness analysis showed no remaining problems after it was rerun.

The last step was to replace Intel Parallel Advisor annotations with Intel TBB parallel framework code and measure the parallel application performance. The Advisor Summary GUI feature provided a convenient way to take a look at the job done on previous steps, compare possible parallelization approaches, and quickly locate the source code regions actually requiring the conversion into the parallel framework instructions (**Figure 8**).

Eventually the code was parallelized and the measurement of the parallel application was done on a quad-socket Intel® Xeon® 24-core server, code-named Dunnington. Experimental data confirmed the optimistic suitability forecast, demonstrating, for example, 11x total improvement on 16 cores (**Figure 6**). It also confirmed that the model with several parallelized loops looks more promising, because in accordance with Amdahl's law, when the most intensive loop takes less time by using a greater number of cores, even lightweight code region parallelization provides a visible benefit in performance as well.

Neural networks serial application

NEURAL algorithms belong to the adaptive systems area, which has numerous applications in biology, artificial intelligence, and artificial neural networks. The Nizhny Novgorod scientists' study attempted to reduce the existing gap between biological and artificial systems by introducing deterministic chaos into the model.

Neural ensemble modeling is a complex "all-to-all" network graph analysis across N vertexes (neurons). Each vertex of the graph can be described by a nonlinear system of differential equations (Hodgkin-Huxley model). Interaction between the neurons is also described by a nonlinear differential equation. **Figure 9** is a simplified representation of the algorithm.

Rate phenomenological model serial application

Another broad class of neural models takes the form of the simpler phenomenological rate models typically used for learning and perception purposes. This modeling methodology was used in the current project for real-time virtual robot control.

The algorithm takes the basic form seen in **Figure 10**.

NEURAL and RATE parallelization case study

Given the similarity of loop structure over all three applications, the method of parallelization used in CARDIAC seemed a likely approach. In the case of NEURAL, it implied a quite straightforward solution as shown in **Figure 9**: there is only one inner loop iterating over the network vertexes (it took 98.9 percent of the total serial application time according to the survey analysis). Thus, developers decided to annotate it, though they had the choice between loop #2 and the innermost loop #3. First they tried loop #3. After adding annotations inside the `F_v` function, responsible for cross-neuron interaction calculations, suitability analysis was run. But the forecast was discouraging, predicting only a 1.01x improvement maximum.

Just adding another pair of annotation macros let developers to test loop #2. The second attempt was more successful: suitability run showed quite an interesting forecast with a scalability local maximum between 4 and 8 cores (**Figure 11**). This data seemed a bit strange to the research scientists after seeing the scalability gains for CARDIAC; they were curious whether or not the experimental data would confirm the Intel Parallel Advisor forecast. As you can see from **Figure 11**, real data (measured on Intel TBB-parallelized application) absolutely confirmed the tool prediction, demonstrating local maximum scaling at 6 cores.

What is behind the different scaling behaviors of CARDIAC and NEURAL? One answer can be found by looking through the suitability report statistics and comparing site "average instance time" metric (**Figure 12**). With CARDIAC, we deal with quite a heavy computational load each time-step (i.e., with "heavy" site), so we don't have to start up and shut down parallel execution too often. On the contrary, NEURAL has very little work for the thread pool to do each time-step, which is executed in very big time loop, resulting in significant threading overheads. That's why the first attempt of parallelization at loop #3 was disappointing. Even with loop #2 parallelized, the time iteration

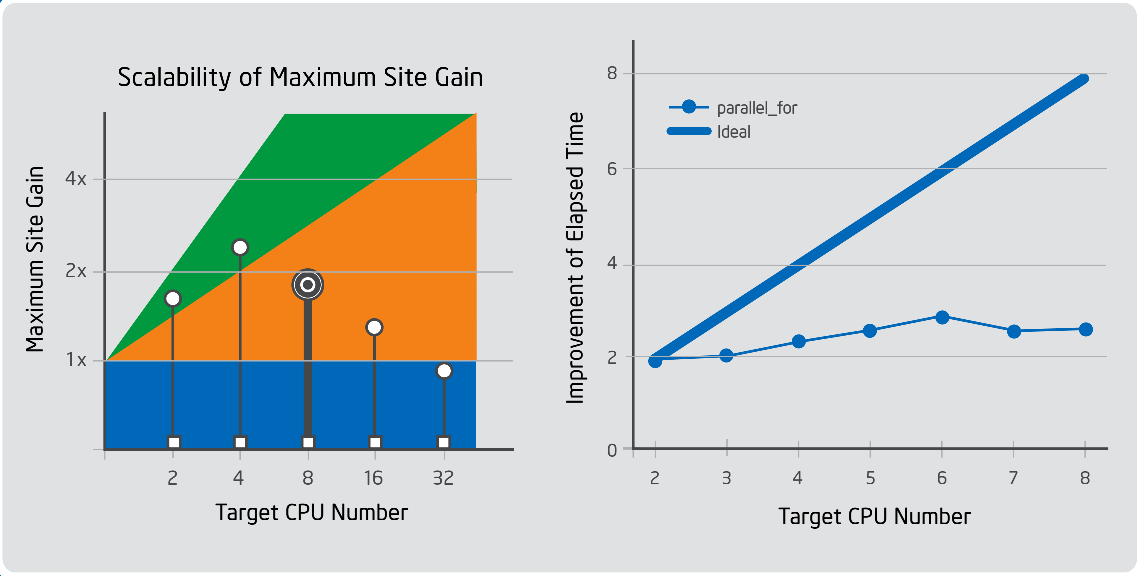
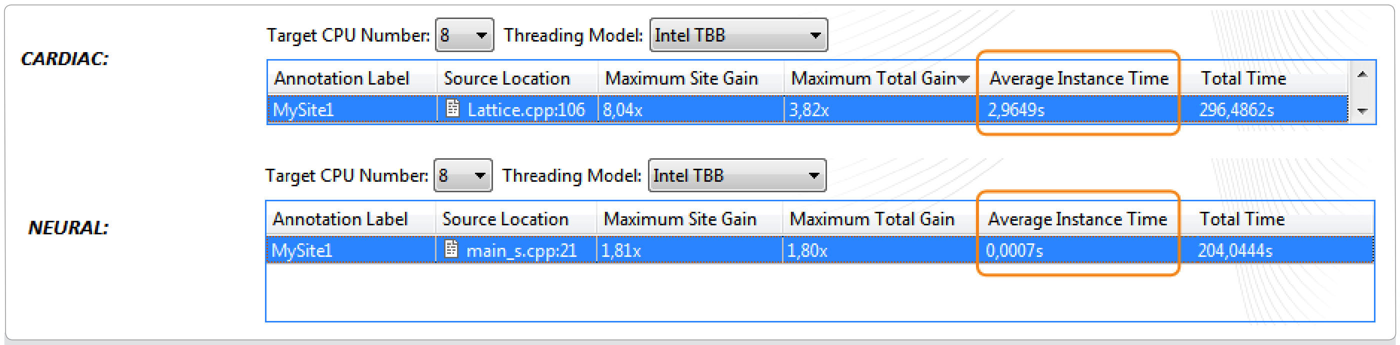


Figure 11: Intel® Parallel Advisor scalability forecast and experimental data for NEURAL application (actual parallelization done with Intel® TBB)

Amdahl's law says that even relatively small pieces of serial execution might affect your performance for a large number of threads.



CARDIAC:

Target CPU Number: 8 Threading Model: Intel TBB

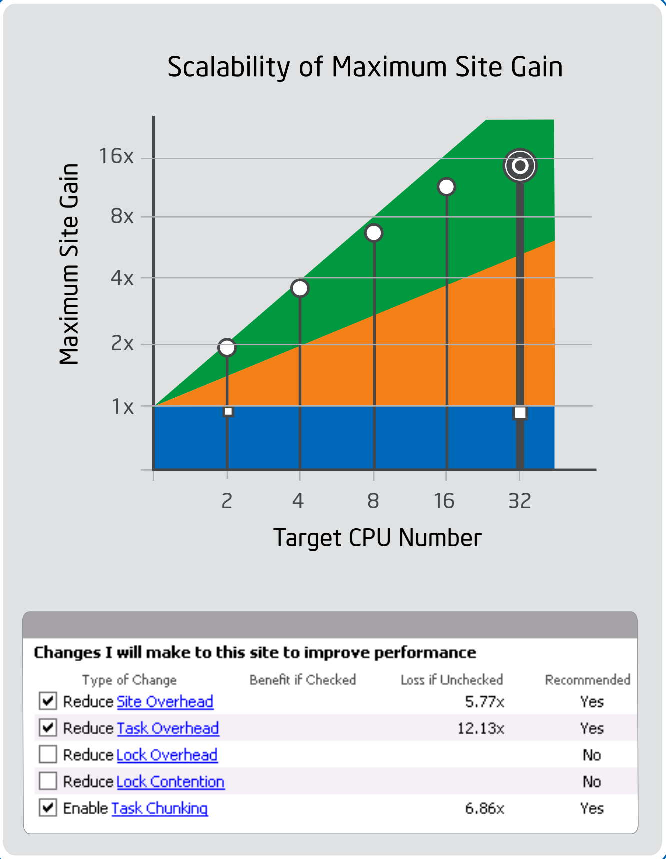
Annotation Label	Source Location	Maximum Site Gain	Maximum Total Gain	Average Instance Time	Total Time
MySite1	Lattice.cpp:106	8,04x	3,82x	2,9649s	296,4862s

NEURAL:

Target CPU Number: 8 Threading Model: Intel TBB

Annotation Label	Source Location	Maximum Site Gain	Maximum Total Gain	Average Instance Time	Total Time
MySite1	main_s.cpp:21	1,81x	1,80x	0,0007s	204,0444s

Figure 12: Intel® Parallel Advisor suitability site metrics for CARDIAC and NEURAL applications



Changes I will make to this site to improve performance

Type of Change	Benefit if Checked	Loss if Unchecked	Recommended
☑ Reduce Site Overhead		5.77x	Yes
☑ Reduce Task Overhead		12.13x	Yes
☐ Reduce Lock Overhead			No
☐ Reduce Lock Contention			No
☑ Enable Task Chunking		6.86x	Yes

Figure 13: RATE model suitability forecast and recommendations

In accordance with Amdahl's law, when the most intensive loop takes less time by using a greater number of cores, even lightweight code region parallelization provides a visible benefit in performance as well.

granularity was not big enough to provide nice scalability. But the lesson learned on this NEURAL example helped in the next model parallelization.

The next study was done on the RATE model. Survey and survey source reports indicated that all three inner loops looked promising, although the first one took about half the time. Based on this, three parallel sites were annotated and a suitability analysis was run. The scalability forecast was only a little bit better than in NEURAL case. Despite the fact that a maximum 5-6x gain is better than nothing, they desired to look for a better approach. So the Intel Parallel Advisor forecast edged the developers into restructuring the serial code a bit.

One obvious idea was to increase the amount of work in the inner loop (i.e., site "average instance time") by fusing all three loops together to decrease total overhead as previous studies suggested. After minimal refactoring, the application was restructured to have a single major parallel site and suitability was rerun. The new result demonstrated much better scalability and opportunities with a potential 14x gain, which appeared for 32 cores (**Figure 13**).

Lessons learned

Despite the fact that the three mathematical models are different, it's easy to see that the parallelization schemes in all three applications look similar. They all consist of an outer time iteration loop, which was not a subject for direct parallelization, and multiple inner loops, which looked quite promising to be parallelized. They found that the right target for parallelization is the inner loop, whose available work should be made as big as possible.

This tends to be a parallelization pattern for various differential equation solvers whose solution is based on an iterative time integration (i.e., solution refinement) basis. This parallelization pattern was quickly recognized by the Nizhny Novgorod research scientists because:

> Intel Parallel Advisor could quickly model different parallelization approaches without any significant application modifications (using annotations).

> Intel Parallel Advisor methodology suggested an efficient method to step through the necessary parallelization stages, providing recommendations and metrics to simplify decision making at every step.

> Even in the case when the Intel Parallel Advisor forecast was negative, its suitability and correctness tools nudged developers in the right direction to restructure the original code.

> The fact that the same group of scientists participated in several parallelization experiments simultaneously, helping share, reuse, and summarize the experience obtained during these different parallelization efforts.

Professor G. Osipov's parallelization studies have proven the value of Intel Parallel Advisor tools and methodology when parallelizing three different HPC-relevant applications. Intel Parallel Advisor tools helped to find regions that will benefit from running in parallel, to identify and fix a data sharing problem, and to model quickly parallel program structure and potential benefits. Intel Parallel Advisor prediction accuracy and efficiency were confirmed by real experimental data.

At the moment, the research activities to parallelize numerical modeling algorithms are underway in Nizhny Novgorod, helping research scientists to improve and analyze their application performance, while helping Intel Parallel Studio developers confirm the strengths and identify possible weaknesses of Intel Parallel Advisor. □

BLOG highlights

Introducing Intel® Fortran Studio XE 2011

STEVE LIONEL, Developer Products Division

Let us return to those thrilling days of yesteryear. Yes, I mean November 2010, when Intel® Parallel Studio XE was first released. This suite of high-performance computing development tools included new versions of the Intel C++ and Fortran compiler products, (now renamed "Composer XE"), and two new analysis tools: Intel® VTune™ Amplifier XE and Intel® Inspector XE. The analysis tools were significantly upgraded versions of the Intel® Parallel Amplifier and Intel® Parallel Inspector that had been launched in 2009 for C/C++ on Windows* only. The new "XE" tools not only supported Fortran as well, but were now available on Linux* for the first time.

Fortran programmers loved the new features of the compiler, but there was some muted grumbling in the background. You see, while it was possible to buy a subset containing C++ and the analysis tools, called Intel® C++ Studio XE, there was no corresponding subset for Fortran-only programmers. Fortran users who also wanted the analysis tools either needed to buy them separately, or purchase the larger Parallel Studio XE product containing a C++ compiler, which, while excellent, might go unused in Fortran-only shops. "Where," you cried, "is our Fortran Studio XE?" Ok, ok. You can put down your pitchforks and Arithmetic IFs—Intel® Fortran Studio XE 2011 is now here for both Linux and Windows!

SEE THE REST OF STEVE'S BLOG:

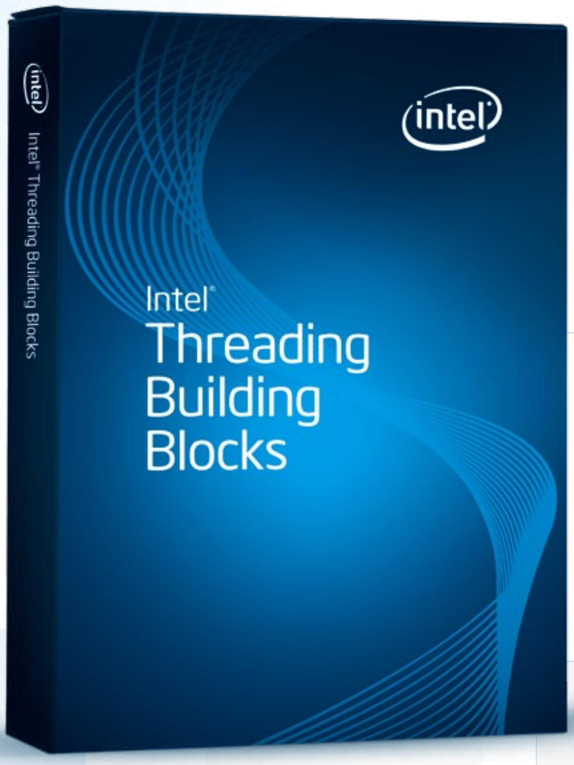
Visit **Go-Parallel.com**

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

THE INTEL® THREADING BUILDING BLOCKS FLOW GRAPH

by Michael J. Voss, Ph.D.
Software Architect

User feedback inspired the flow graph feature in Intel® Threading Building Blocks, which allows programmers to express static and dynamic dependency graphs, as well as reactive or event-based graphs.



MICHAEL J. VOSS
Senior Software Engineer

Sign up for future issues | Share with a friend

Intel® Threading Building Blocks (Intel® TBB) 4.0 includes flow graph as a fully supported feature. The flow graph supports both static and dynamic dependency graphs, as well as reactive graphs that respond to and pass data messages. Introduced as a Community Preview feature[1] in Intel® TBB 3.0 Update 5, the flow graph interface has been refined and improved based on several months of user feedback.

In fact, numerous development teams across the media, gaming, financial services, and technical computing segments have been evaluating the flow graph as an Intel TBB 3.0 Community Preview feature. Before the flow graph was available, some event-based and reactive programs were simply impractical to implement using Intel TBB. In other cases, users were either writing complex code that used the low-level Intel TBB tasking interface directly, or were over-constraining their parallelism to use an Intel TBB pipeline. The flow graph provides a more natural fit for many applications, while maintaining or improving performance over other Intel TBB-based solutions.

An overview of the flow graph interface

An Intel TBB flow graph consists of three primary components: a `graph` object, nodes, and edges. The `graph` object provides methods to run tasks in the context of the graph and to wait for the graph to complete. Nodes generate, transform, or buffer messages. Edges wire the graph together, connecting the nodes that send messages to the nodes that should receive them. There are several types of nodes, as shown in **Figure 1**. There are functional nodes that execute user code, buffering nodes, nodes that join and split messages, and several other miscellaneous node types.

A dependence graph example

Figure 2 shows an approach to implementing a wave-front computation using a set of `continue_node` objects. In this example, each computation must wait for the computation above it and the computation to its left to complete before it can start executing. A `continue_node` starts executing when it receives a `continue_msg` from each of its predecessors.

In **Figure 3**, this approach is used to implement a blocked wave-front calculation, where each computation updates a `BxB` block of the matrix `values`. The `for` loop in function `run_graph` creates the set of the `continue_node` objects. In the figure, the `continue_node` constructor is passed a reference to the `graph` object `g` and a function object (or in this case a lambda expression) that calls `update_block` on its block.

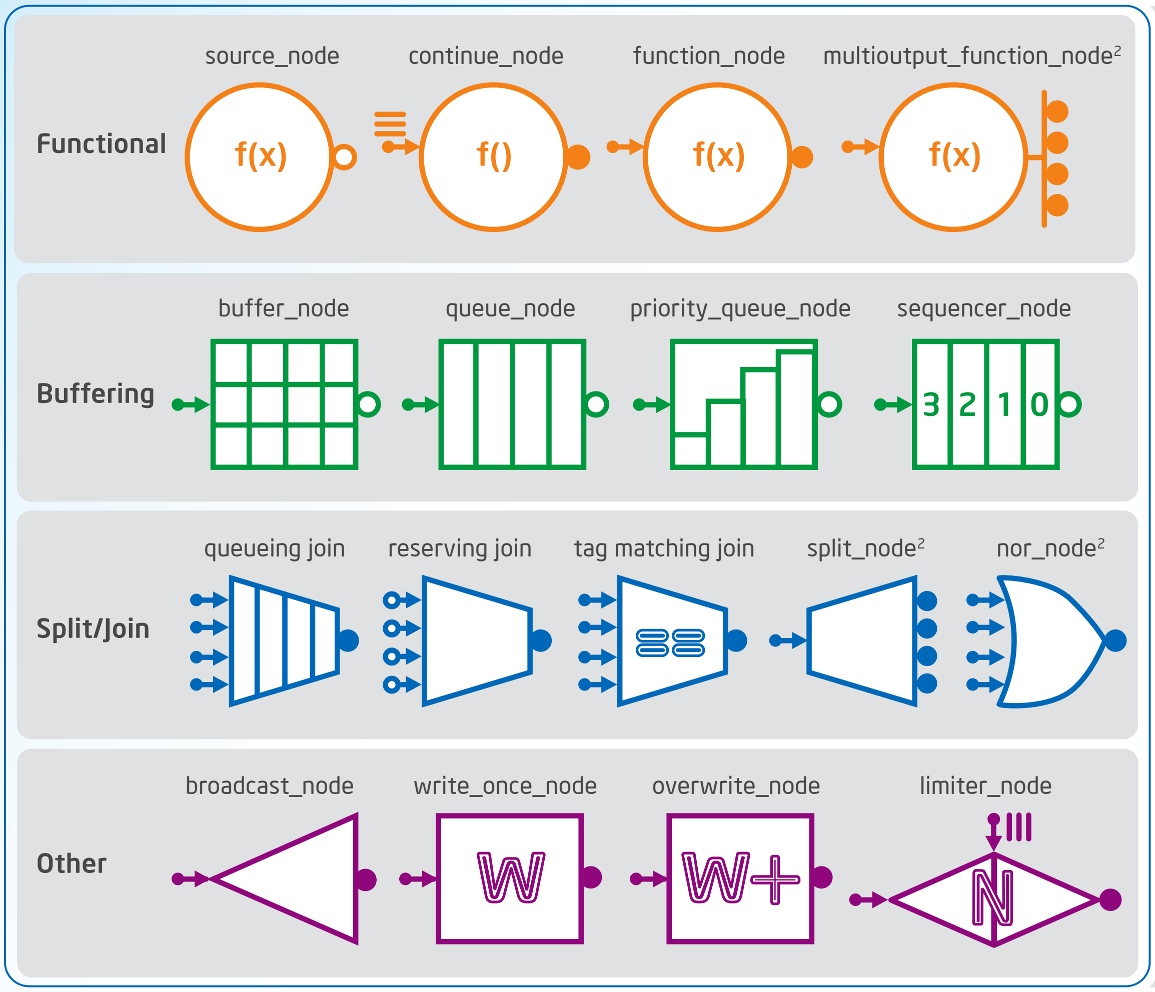


Figure 1: The node types supported by the Intel® Threading Building Blocks flow graph

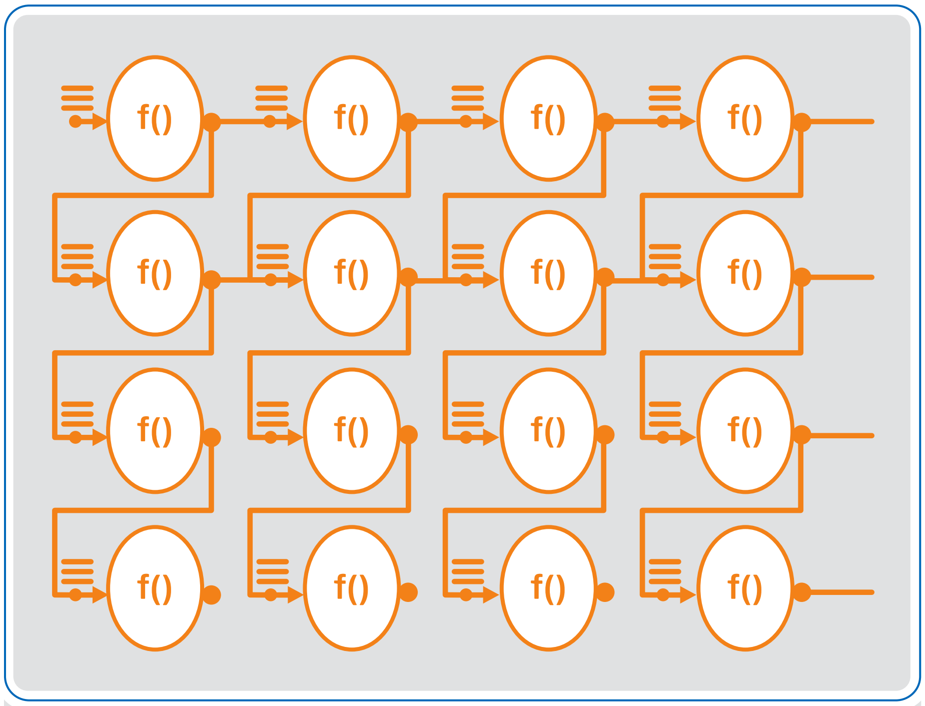


Figure 2: Using an Intel® TBB flow graph to express a wave-front calculation

```
// M and N are the number of rows and columns in the matrix
// MB and NB are the number of blocks in the rows and columns
// B is the block size (BxB squares)

using namespace tbb;
using namespace tbb::flow;

double value[M][N];

graph g;
continue_node<continue_msg> *node[MB][NB];

double run_graph( ) {
  value[M-1][N-1] = 0;
    for( int i=MB; --i>=0; ) {
     for( int j=NB; --j>=0; ) {
      node[i][j] =
       new continue_node<continue_msg>( g,
         [=]( const continue_msg& ) { update_block( i, j ); } );
       if ( i + 1 < MB ) make_edge( *node[i][j], *node[i+1][j] );
       if ( j + 1 < NB ) make_edge( *node[i][j], *node[i][j+1] );
    }
  }
  node[0][0]->try_put(continue_msg());
  g.wait_for_all();
  for( int i=0; i<MB; ++i )
     for( int j=0; j<NB; ++j )
          delete node[i][j];
  return value[M-1][N-1];
}
```

Figure 3: An implementation of a blocked wave-front calculation

Once the flow graph is set up in the example, a `continue_msg` is put to the node in the upper left corner, `node[0][0]`, to start the wave front through the graph. The call to `g.wait_for_all()` blocks until the entire wave-front computation completes.

A complete description of this example and complete source code can be found in the blog article, "Implementing a wave-front computation using the Intel® Threading Building Blocks flow graph" found at http://software.intel.com/en-us/blogs/tag/flow_graph.

A message graph example

Figure 4 shows an Intel Threading Building Blocks flow graph that implements a simple feature detection application. A number of images will enter the graph and two alternative feature detection algorithms will be applied to each one. If either algorithm detects a feature of interest, the image will be stored for later inspection.

In the figure, a `source_node`, `src` supplies images to a reserving join node, `resource_join`. The second input of `resource_join` is connected to a queue of image buffers, `buffers`. A `source_node` only generates new items after its current output has been consumed. A reserving join node does not consume incoming items until it can reserve an input at each of its incoming ports. The front of this graph is therefore nicely constructed to control memory use. New images will only be generated by `src` if an image buffer is available in `buffers` to pair with it.

Once an incoming image is paired with a buffer, it is forwarded to the `function_node`, `preprocess_function`, which preprocesses the image, placing the results in the associated buffer. The `preprocess_function` may be created with unlimited concurrency, allowing it to process multiple images concurrently. In a feature-detection application, this preprocessing might, for example, include algorithms for blurring the image.

The output of `preprocess_function` is connected by an edge to `detect_A` and `detect_B`, which implement two alternative algorithms for detecting the feature of interest in the images. Again, these nodes

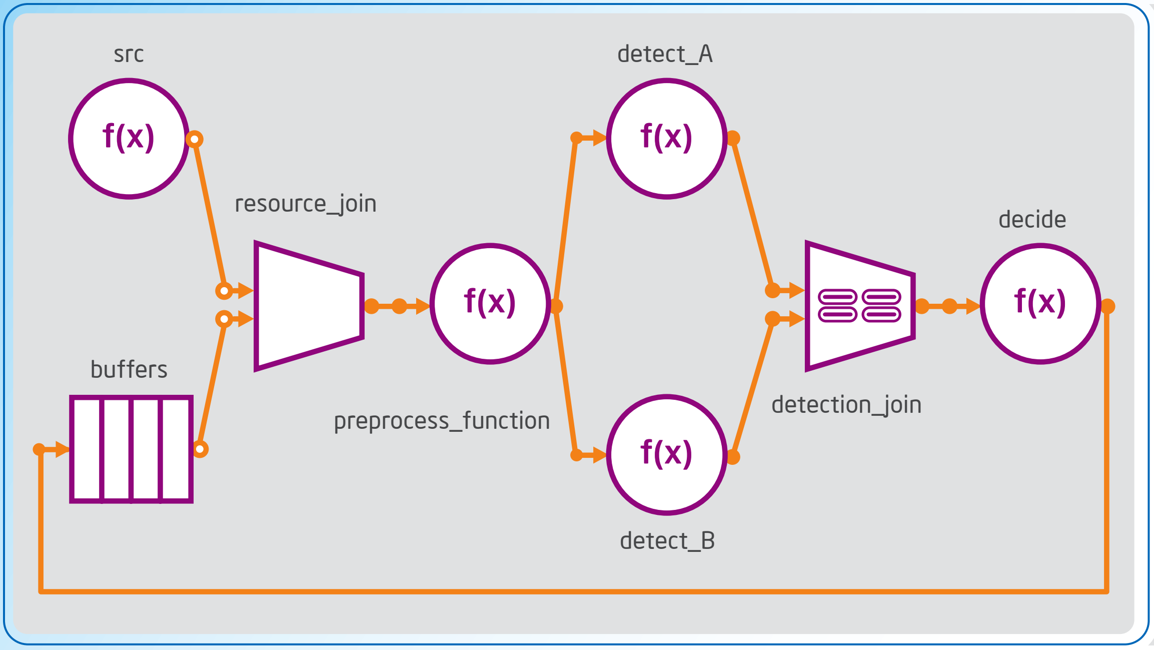


Figure 4: A feature detection example using an Intel® TBB flow graph

	Task graphs	Pipeline / `parallel_pipeline`	Flow graphs
Expressiveness	Can express acyclic dependency graphs	Can express linear pipelines	Can express acyclic dependency graphs as well as acyclic and cyclic messaging graphs
Ease-of-use	Requires low-level bookkeeping code and explicit spawning of tasks	A concise, type-safe interface	More verbose than `parallel_pipeline`, but does not require explicit bookkeeping or task spawning
Persistence	Graphs are executed destructively; cannot be re-executed	Can be executed multiple times (applies to pipeline only)	Can be executed multiple times
Performance[3]	Very low overhead since it is built directly on tasks and is executed destructively	Uses last-in, first-out (LIFO) task scheduling to optimize for cache locality. Overhead is comparable to flow graph.	Uses first-in, first-out (FIFO) task scheduling to keep messages flowing through the graph. Overhead is comparable to pipeline and `parallel_pipeline`.

Table 1: Comparing flow graph, `parallel_pipeline`, and graphs of tasks

may be created with unlimited concurrency, allowing multiple images to be processed by each node concurrently. The outputs of these detection nodes are forwarded to a tag matching join, `detection_join`. A tag matching join pairs items together based on matching tags; in this case, it will pair the outputs of `detect_A` and `detect_B` based on the image they were processing. Use of a tag matching join is important here because multiple images may be in flight in the graph simultaneously, and it's important to match the proper results together.

Finally, a pair of results reaches the `function_node decide`. It inspects the results from each algorithm to see if the feature might be present in the image. If so, it stores the image for later inspection. When `decide` is complete, it returns the buffer to `buffers` so it can be paired with another incoming image.

A more complete description of this example and complete source code can be found in the blog article, "A feature-detection example using the Intel® Threading Building Blocks flow graph" found at http://software.intel.com/en-us/blogs/tag/flow_graph.

Choosing between a flow graph, pipeline, or an acyclic graph of tasks

While the flow graph adds significant functionality to Intel Threading Building Blocks 4.0, some applications suited to the flow graph can be implemented using the existing low-level support for acyclic graphs of tasks and the generic `parallel_pipeline` algorithm. **Table 1** compares these different features and provides characteristics that may help in selecting the most appropriate model to use.

Summary

Intel Threading Building Blocks (Intel TBB) 4.0 includes flow graph as a fully supported feature. A flow graph can be used to express static and dynamic dependency graphs, as well as reactive or event-based graphs that respond to and pass messages between computations. You can learn more about this feature and download the Intel TBB 4.0 library at www.threadingbuildingblocks.org. ☐

1. As a Community Preview feature, the *flow graph* was named *graph*. We now use the name *flow graph* to emphasize that this feature expresses the control flow in an application. The more generic name *graph* falsely implied a more data-structure centric approach and a collection of classical graph-based algorithms.
2. `multioutput_function_node`, `split_node`, and `or_node` are Community Preview features in Intel TBB 4.0
3. Refer to http://software.intel.com/en-us/articles/optimization-notice for more information regarding performance and optimization choices in Intel software products.

BLOG highlights

Understanding the Internals of tbb::graph: Balancing Push and Pull

MICHAEL J. VOSS, PH.D., Senior Software Engineer

In this post, I'm going to describe the hybrid push-pull protocol used by Intel® Threading Building Blocks graph Community Preview Feature.

The hybrid push-pull protocol used by tbb::graph biases communication to prevent polling and to reduce unnecessary retries. Understanding the details of this protocol is not necessary to use tbb::graph, but it makes understanding its performance easier.

Nodes in a graph are persistent and exist until a user explicitly destroys them. But unlike some actor systems, a thread is not assigned to each tbb::graph node. Tasks are created on-demand to execute node bodies and pass messages between nodes when there is activity in the graph. Consequently, a tbb::graph node does not spin in a loop waiting for messages to arrive. Instead when a message arrives, a task is created to apply the receiving node's body to the incoming message.

SEE THE REST OF MICHAEL'S BLOG:

Visit **Go-Parallel.com**

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

Sign up for future issues | Share with a friend

intel
Intel® Parallel Advisor
Intel®
Parallel
Advisor
2011

Intel® Parallel Studio XE SP1

Intel® Parallel Studio XE combines Intel's industry-leading C/C++ and Fortran compilers, high- performance parallel libraries, error checking, code robustness, and **performance profiling** technologies into a single suite offering. The SP1 release now adds functionality to simplify the transition from multicore to many-core hardware platforms.

by Michael D'Mello

Sign up for future issues | Share with a friend

Even the most masterful software developers are often left with nagging performance and correctness questions: Can I make my software run faster on the current or next generation hardware? What limits the performance of my code? How susceptible is my software to errors and security vulnerabilities?

For years, multitudes of developers used a variety of software tools from Intel to help them address these types of questions. In 2010, this community of developers began moving to a new generation of software tools called Intel® Parallel Studio XE. Designed to provide a wide range of functionality while maximizing the user experience, this bundle of sophisticated tools makes the identification, characterization, and "cure" of performance bottlenecks, memory and threading errors, and security issues as painless as possible. The toolset also provides developers with industry-leading C++ compilers and Fortran compilers and a set of parallel programming models to prepare and position professionals for future generations of multicore and many-core hardware.

Inside Intel Parallel Studio XE

It is worthwhile to examine what underlying technologies have led to the success of Intel Parallel Studio XE, and how the upcoming SP1 release of this toolkit will further add to the power of the suite.

The essential functionality of Intel Parallel Studio XE includes:

> Intel® C++ Compiler XE and Intel® Fortran Compiler XE[1]
- Intel® Math Kernel Library, Intel® Integrated Performance Primitives
- Intel® Threading Building Blocks

> Intel® VTune™ Amplifier XE performance profiler

> Intel® Inspector XE dynamic memory and threading checker

> Intel® Static Security Analysis static error and security checker

The need for excellent compilers is obvious, and Intel compilers lead in providing performance for best single-core performance and multi-core scalability. The compilers are maintained on a path that targets and accommodates all the latest hardware platform innovations. A very relevant current example of this is support for Intel® Advanced Vector Extensions (Intel® AVX). This vector register technology is available in the latest processors based on the Intel® microarchitecture known by its codename, Sandy Bridge. The technology is capable of up to 2x performance speedup over the existing Streaming SIMD Extensions (SSE) format. The SP1 release tunes support for AVX and the Sandy Bridge platform in general, and like the previous release of Intel® Parallel Studio XE, it will provide the same set of tools on Windows*, Linux*, and Mac OS* X operating systems.

Guided auto-parallelization

Beyond extensive support for the very latest in multicore and many-core hardware, there are elements of the Intel compiler that users often overlook. These are optimization modes beyond the traditional -O1, -O2 switches that practitioners are familiar with. One such optimization technology is guided auto-parallelization (GAP). This is a workflow-oriented approach that provides compiler-generated guidance to change source code so that it can be compiled for greater performance through vectorization, parallelization, and/or data transformations. Besides advice on source code changes and the addition of compiler directives (pragmas, for example), GAP also gives advice on compilation options. GAP is flexible and can be combined with interprocedural optimization (IPO)[2] and profile guided optimization (PGO)[3], two other modes of optimization also provided by the Intel compiler.

Multicore-ready libraries

Next on the list are the Intel Math Kernel Library (Intel® MKL) and the Intel Integrated Performance Primitives (Intel® IPP) libraries. These multicore-ready libraries provide some of the easiest and most direct mechanisms for code parallelization and performance gain possible. Heavily used in scientific and engineering applications, Intel MKL is a staple in the energy, healthcare, financial analytics, and high-performance computing (HPC) domains. The Intel IPP library plays a similar role for optimizing software in the multimedia, data processing, and communications domains. These libraries attempt to maximize the use of vectorization and threading for best single-core and multicore performance, respectively. With the SP1 release, these libraries extend their seamless support of the benefits of AVX, among many other optimizations specific to the Sandy Bridge microarchitecture.

Guided auto-parallelization is a workflow-oriented approach that provides compiler-generated guidance to change source code so that it can be compiled for greater performance through vectorization, parallelization, and/or data transformations.

The C/C++ optimizing compiler includes Intel Threading Building Blocks (Intel® TBB) and Intel® Cilk™ Plus. Intel TBB provides C++ language support for task-based parallelism while letting the compiler do vectorization. Intel Cilk Plus provides capabilities for task, vector, and data parallelism. Both are highly relevant as hardware evolves to include different types of cores (i.e., heterogeneity) on the same platform. For Fortran developers, Intel Parallel Studio XE offers Co-array Fortran and, with the SP1 release, industry-leading support for the Fortran 2008 standard as well.

The Intel TBB C++ template library made its debut in 2006. Since then, it has enjoyed widespread adoption among C++ developers. This task-based parallel library internally maintains a thread pool and a task scheduler. The task scheduler maps user-created tasks to the library managed pool of threads. The scheduler accommodates affinitization of tasks to threads, and this feature allows
for some remarkable optimizations to be included directly into the functionality of the library. For example, similar tasks often address and consume similar data, and by affinitizing these tasks to a certain thread, the library can, to some degree, ensure the data required by these tasks remains available (i.e., "hot") in cache. The SP1 release offers a major enhancement to the Intel TBB library—Intel TBB flow graph (see article by Michael Voss on page 14). The concept here is to enable the developer to introduce parallelism by focusing on the graph representing the functionality sought after. This higher-level perspective is expected to reduce implementation time significantly while leveraging all the performance benefits built into the library.

Performance profilers

Intel® VTune™ Amplifier XE represents one of the most complete profilers available in the industry today. The functionality of this tool is conveniently separated into two parts—time-based profiling and event-based profiling. The time-based functionality covers traditional profiling of code, including call stacks, as well as "Concurrency" and "Locks and Waits" type analyses for multithreaded code. To complete the picture provided in the time-based profiles, the tool enables the user to leverage hardware counters and thereby track numerous processor events (e.g., instruction retirement, cache misses, TLB misses, etc.) generated as a code executes. This functionality is referred to as event-based sampling (EBS), and it comes with very low overhead because of the direct support by Intel® processors. The user therefore gets a highly detailed characterization of how a given piece of software drives the underlying hardware. The information is invaluable in understanding not only how a program runs, but also how well it is written.

Another feature of Intel VTune Amplifier XE worth highlighting here is "Frame Analysis." This is a feature that applies to marking the timeline view of the profile of a code. Basically, the timeline view is a picture of the time evolution of threads and the interactions between them. Threads "interact" via operating system objects (mutexes, locks, etc.), and interactions are indicated on the timeline by lines drawn between threads. The timeline view is fundamental to understanding load balance; it is an integral part of the "concurrency" and "locks and waits" analyses mentioned above. However, for continuously running codes, a user may be interested in only a portion of the timeline. Frame Analysis enables the user to mark the timeline to identify the region or regions of interest. A computer game is a good example of a continuously running code; so is a financial trading engine. Indeed, many codes, in a variety of application areas, may be categorized as such.

Sign up for future issues | Share with a friend

Perfecting your code

Finally, no code is perfect, and any comprehensive toolset has to provide some support for error checking and detection. The Intel® Inspector XE component provides memory-checking and thread-checking functionality. Memory leaks, race conditions, and deadlocks are some of the main types of errors tracked. Intel Inspector XE can also be used to visualize the results of the source-level error-checking functionality of the Intel C++ compiler. This functionality, referred to as Intel® Static Security Analysis, examines source code for errors and security vulnerabilities. Collectively, these components of the suite provide powerful mechanisms to enhance robustness and drive overall code quality.

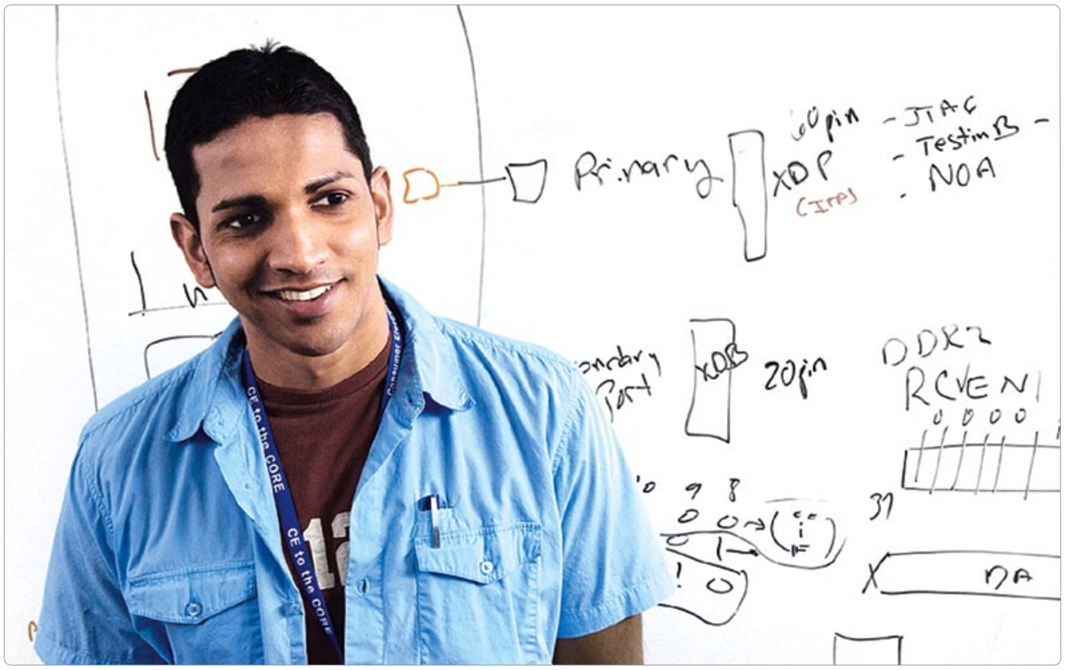
The timeline view is fundamental to understanding load balance; it is an integral part of the "Concurrency" and "Locks and Waits" analyses.

Summary

Intel® Parallel Studio XE offers a suite of tools to help software developers write better code on the latest available multicore and many-core platforms. The suite directly addresses questions of code quality, robustness, security, performance, and scalability. The design and responsiveness of the suite's components make for a positive and highly profitable end-user experience. The SP1 release furthers the proposition of a highly convenient mechanism to enable developers to enhance the value of their own software solutions and that of their enterprise. ☐

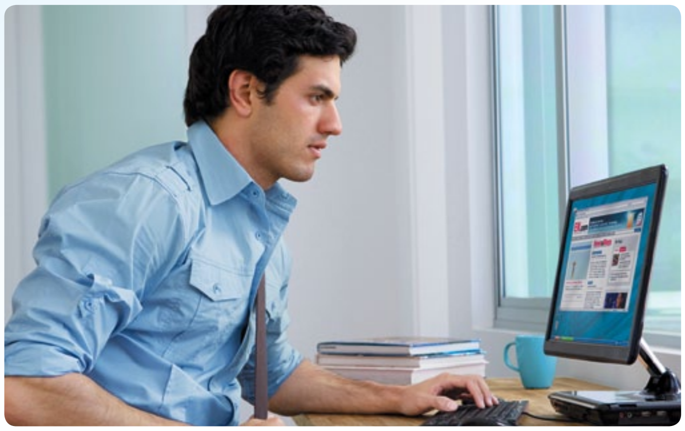
1. For convenience, several variations of this set are also offered. For example, the first three items along with Intel® Static Security Checker have been assembled into a single bundle called Intel® Composer XE. A version of this bundle, called Intel® C++ Composer XE, is also available. This is essentially the Intel® Composer XE bundle without the Fortran components. Intel® C++ Composer XE along with items four and five on the list forms Intel® C++ Studio XE. The analogous bundle for Fortran users, Intel® Fortran Studio XE, is available as well.
2. Interprocedural Optimization is a collection of compiler optimization techniques based on analyzing the entire program rather than a single function or code block, which is typical of other optimization techniques.
3. Profile Guided Optimization is a compiler technology that seeks to produce a more optimized executable from a given executable. The creation of the optimized version is guided by the results of one or more runs of the original executable with a representative dataset or workload. Using this information, the compiler tries to generate an optimized executable that runs faster than the original one.

Download a free trial of this software for a limited time at http://intel.com/software/products.



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.

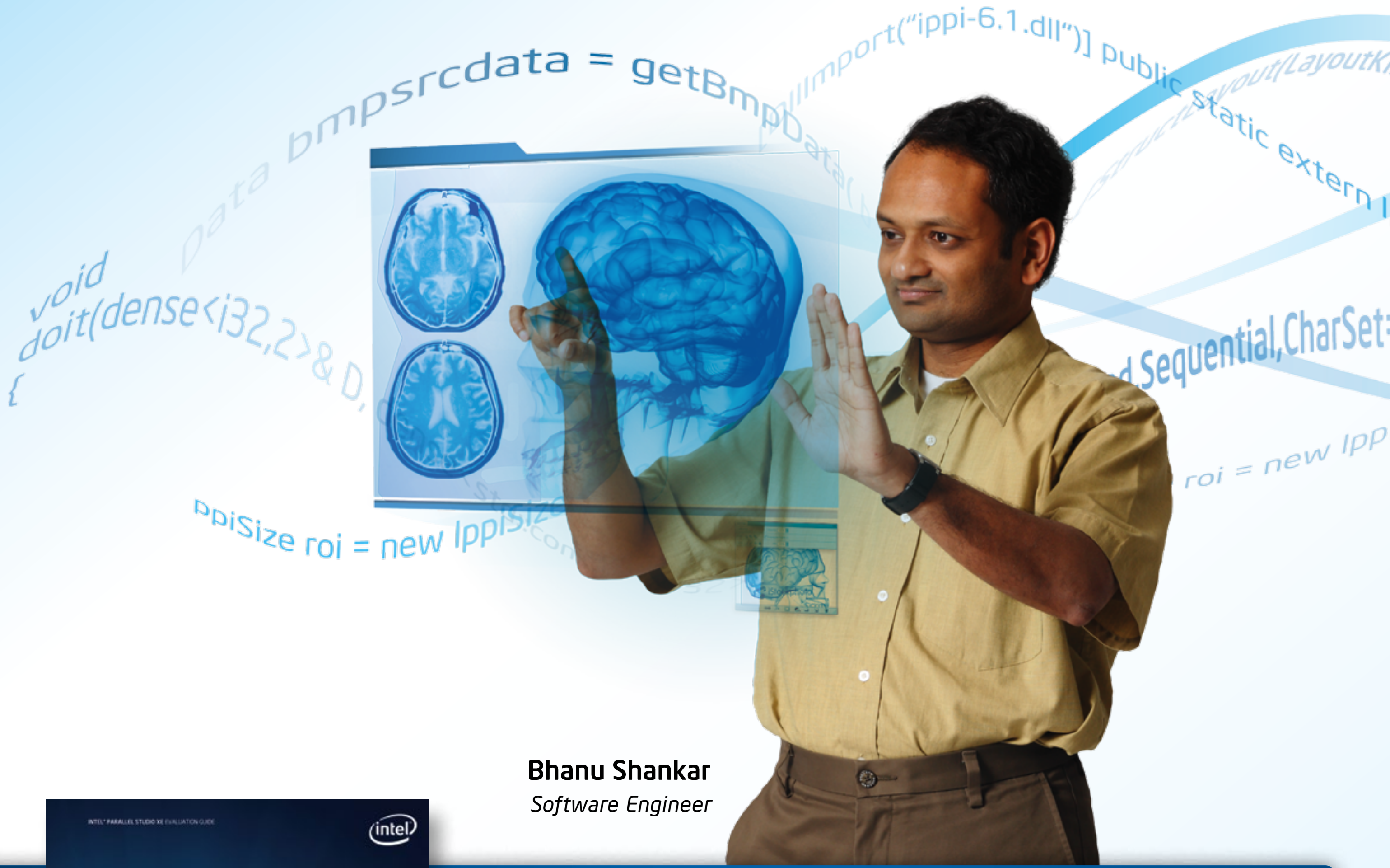
Intel® Inspector XE

Intel® VTune™ Amplifier XE

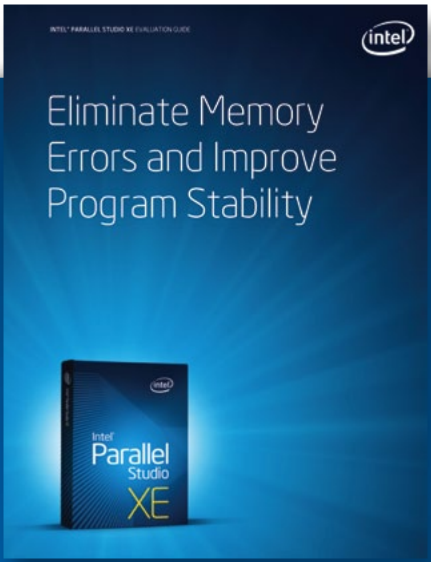
Intel® Software Evaluation Center

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.

Smooth the path to improved performance


void doit(dense<i32,2>& D,
{
data bmpsrcdata = getBmpData(
import("ippi-6.1.dll")] public static extern
Sequential,CharSet
IppiSize roi = new IppiSize
roi = new Ipp

Bhanu Shankar
Software Engineer



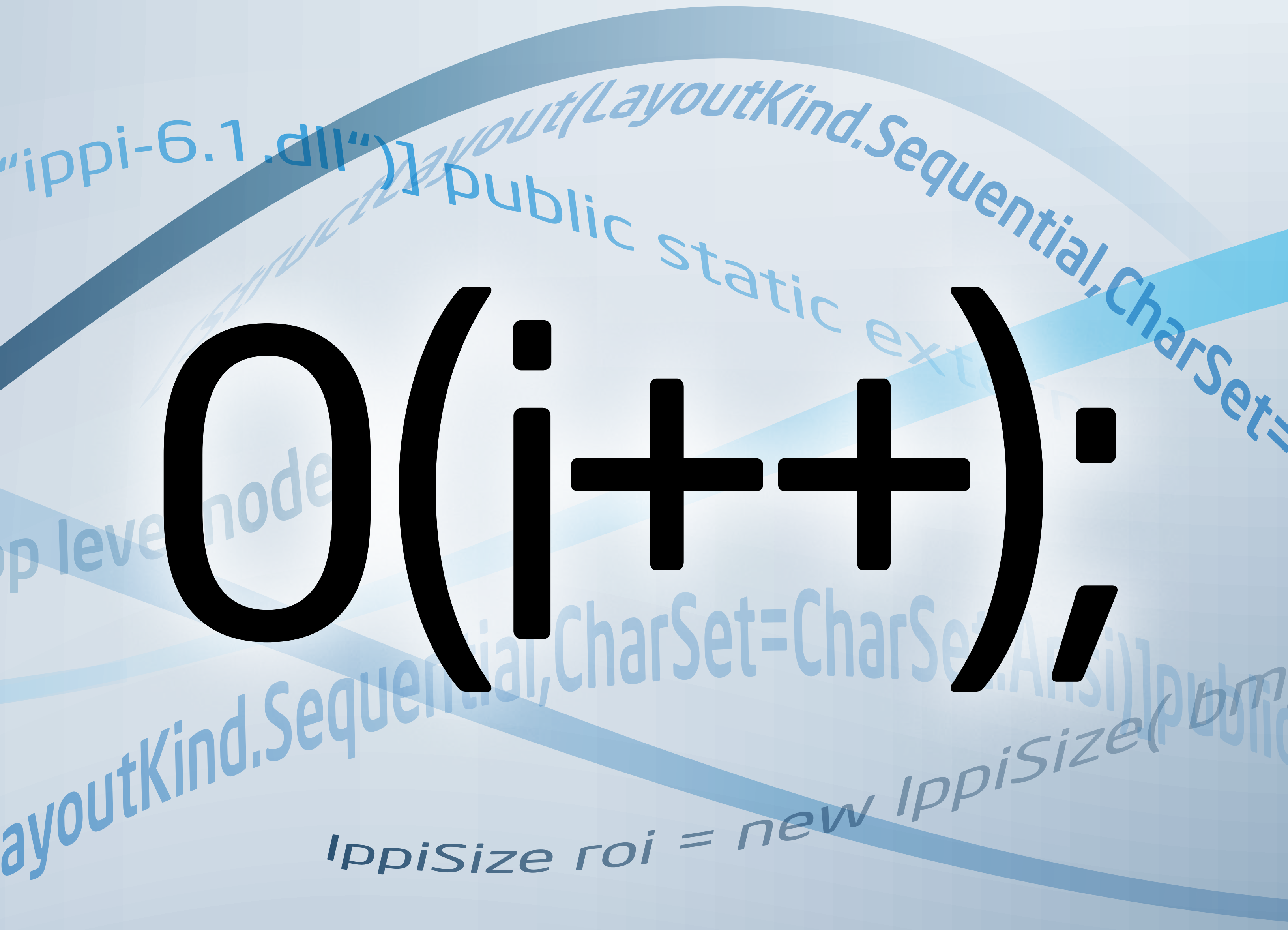
Pinpoint errors and help ensure application reliability and quality with "Eliminate Memory Errors and Improve Program Stability."

The free step-by-step evaluation guide walks you through how to:

> Identify, analyze, and resolve threading errors in parallel programs.
> Find bugs and get graphics to render correctly.
> Use a command-line interface to automate testing.
> Choose small, representative data sets.

Rock your code.
ROCK YOUR WORLD.

DOWNLOAD THE HANDS-ON GUIDE TODAY


O(i++);
"ippi-6.1.dll")] public static extern
[StructLayout(LayoutKind.Sequential,CharSet=
p level node
ayoutKind.Sequential,CharSet=CharSet.Ansi)]public
IppiSize roi = new IppiSize(bm

What will your compiler do with this C++ statement?

O(i++); will trigger different responses depending on if you are using g++ or Intel® C++ and Windows* or Linux*.

Find out how different—and learn how you can quickly improve application performance without rewriting a single line of code.

GET THE ANSWER TODAY