Intel® Software
Development Products

Software

Writing Parallel Programs That Work

Paul Petersen (Intel)

Abstract

Serial algorithms typically run inefficiently on parallel machines. This may sound like an obvious statement, but it is the root cause of why parallel programming is considered to be difficult. The current state of the computer industry is still that almost all programs in existence are serial.

This talk will describe the techniques used in the Intel Parallel Studio to provide a developer with the tools necessary to understand the behaviors and limitations of the existing serial programs. Once the limitations are known the developer can refactor the algorithms and reanalyze the resulting programs with the tools in the Intel Parallel Studio XE to create parallel programs that work.

Remember this slide from earlier?

- Basic approach
  - **Find compute intensive loops**
  - **Make the loop iterations independent … so they can safely execute in any order without loop-carried dependencies**
  - Place the appropriate OpenMP directive and test

This reminds me of a rather famous cartoon:

(just use Google search on: **then a miracle occurs cartoon**

Where should we begin?

- Do you start with a blank sheet of paper … or not?

1. You need to organize your ideas
   - Usually by expressing a serial algorithm
2. Otherwise you may be asked to improve software with demonstrated value
   - Usually expressed by a serial implementation
   - Or a process in your MPI program, another serial program

- Either way, you likely start serial

Do you really mean that?

1

```
for (int I = 0; I < N; ++I)
        A[I] = B[I] + C[I];
```

- This loop is equivalent to:

```
I = 0;
if (! (I < N)) goto done;

A[0] = B[0] + C[0]; // I = 0

++I;
if (! (I < N)) goto done;

A[1] = B[1] + C[1]; // I = 1
…
    done:
```

2

```
for (int I = 0; I < N; ++I)
        Work( &A[I] );
```

- This loop is equivalent to:

```
I = 0;
if (! (I < N)) goto done;

Work( &A[0] ); // I = 0

++I;
if (! (I < N)) goto done;

Work( &A[1] ); // I = 1
…
    done:
```

Or did you really mean this…

1

```
for (int I = 0; I < N; ++I)
        A[I] = B[I] + C[I];
```

```
A[0…N-1] = B[0…N-1] + C[0…N-1]
```

2

```
for (int I = 0; I < N; ++I)
        Work( &A[I] );
```

```
foreach X in A[0…N-1]
        Work( &X );
```

or even…

```
Work( &A[0…N-1] )
```

Digression: debugging

- What is still the #1 debugging tool in use today?
  - A "`print`" statement

- Inserting a "`print`" statement into your serial program typically does not change its behavior, but allows observation of what is happening

- Serial languages force you to specify the semantics of your algorithm by enforcing a specific serial execution

- Parallel languages force you to specify what is allowable to execute in concurrently

We can use this debugging technique to bridge the gap and observe potential parallelism

Bridging the gap

- Annotations are statements that *markup* existing algorithms to express the parallel model requested
- Intel® Advisor XE accomplishes this with a core set of modeling annotations:
  - **SITE** (where should I focus)
  - **TASK** (what should I do)
  - **LOCK** (it really is serial)
- Similar to concepts in OpenMP, TBB, or Cilk – but simplified

Annotations can be considered *as-if* they are "print" statements to a special trace file

Annotation: SITE (where should I focus)

Intel® Advisor XE uses a **SITE** to:

1. Define a name for a section of the application
2. Declare that all interesting things to analyze occur inside of this section
3. Declare that any parallelism that is declared will be finished before the section exits

Example: SITE

```
#include <advisor-annotate.h>
…
ANNOTATE_SITE_BEGIN(MySite);
Work( &A[0] );
Work( &A[1] );
ANNOTATE_SITE_END();
…
```

- Annotations are just statements with no visible side-effects

Annotation: TASK (what should I do)

Intel® Advisor XE uses a **TASK** to:

1. Define a name for a section of the application
2. Declare the statements in this section could be executed immediately or deferred
3. Declare the statements in this section can overlap (concurrent/parallel) execution with other statements in the enclosing **SITE**

Example: TASK

```
#include <advisor-annotate.h>
…
ANNOTATE_TASK_BEGIN(MyTask0);
Work( &A[0] );
ANNOTATE_TASK_END();

ANNOTATE_TASK_BEGIN(MyTask1);
Work( &A[1] );
ANNOTATE_TASK_END();
…
```

- Annotations can partition the serial execution to be a specification for parallel execution

Annotation: LOCK (it really is serial)

Intel® Advisor XE uses a **LOCK** to:

1. Define an (non-unique) name for a section of the application

2. Declare that sections with this name are not allowed to execute in parallel, they must be serialized

Example: LOCK

```
#include <advisor-annotate.h>
…
ANNOTATE_LOCK_ACQUIRE(0);
globalCounter = globalCounter + 1;
ANNOTATE_LOCK_RELEASE(0);
…
```

- A **LOCK** is a concept familiar to developers, but it really means a serialized section of code
- This annotation may be implemented with other non-lock based synchronization mechanisms, such as atomic variables

What about loops?

- The combination of a **SITE** + **TASK** using a serial looping construct can be used to model a parallel loop

```
#include <advisor-annotate.h>
…
ANNOTATE_SITE_BEGIN(MyLoopSite);
for (int I = 0; I < N; ++I) {
    ANNOTATE_TASK_BEGIN(MyIteration);
    Work( &A[I] );
    ANNOTATE_TASK_END();
}
ANNOTATE_SITE_END();
…
```

```
#include <omp.h>
…
#pragma omp parallel
#pragma omp single
for (int I = 0; I < N; ++I) {
#pragma omp task
        Work( &A[I] );
}
…
```

OpenMP equivalent

What about loops (continued)?

- The combination of a **SITE** + **TASK** can be optimized via the **ITERATION_TASK** annotation

```
#include <advisor-annotate.h>
…
ANNOTATE_SITE_BEGIN(MyLoopSite);
for (int I = 0; I < N; ++I) {
    ANNOTATE_ITERATION_TASK(MyIteration);
    Work( &A[I] );
}
ANNOTATE_SITE_END();
…
```

```
#include <omp.h>
…
#pragma omp parallel for
for (int I = 0; I < N; ++I) {
    Work( &A[I] );
}
…
```

OpenMP equivalent

Intel® Advisor XE

- A product in Intel® Parallel Studio 2013, which is a plug-in for Microsoft* Visual Studio, and also available on Linux
- A design tool that assists in making good decisions to transform a serial algorithm to use multi-core hardware
- A serial modeling tool using annotated serial code to calculate what might happen if that code were executed in parallel as specified by the annotations
- A methodology and workflow to help you learn where to use parallel programming

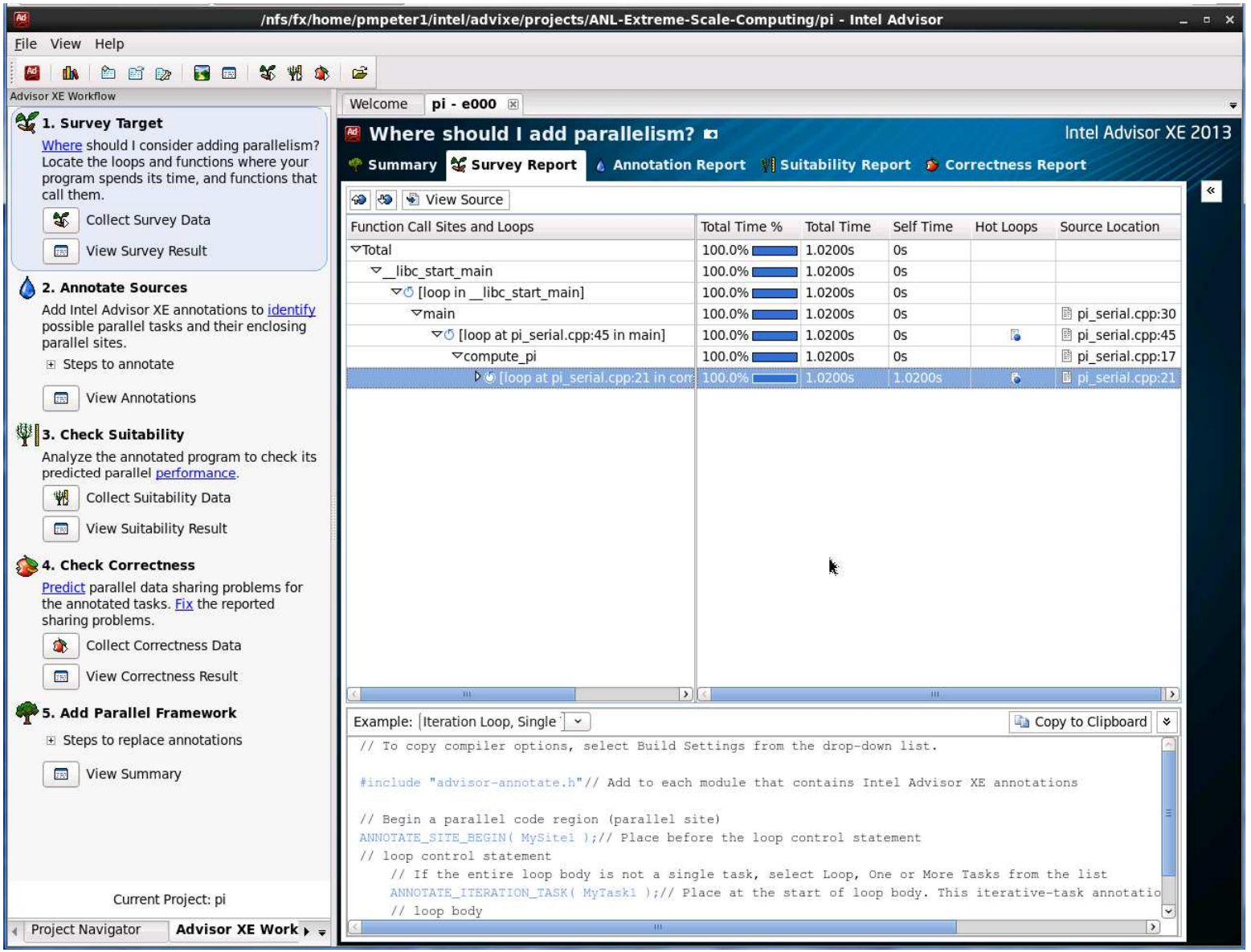
Why estimate performance first?

- If programs were trivial to parallelize we would have already finished converting them
- Amdahl's law states the benefit of parallelism is a based on the fraction of execution you parallelize
- Therefore, you must focus effort on the places that are valuable to parallelize, not the places that are easy to parallelize

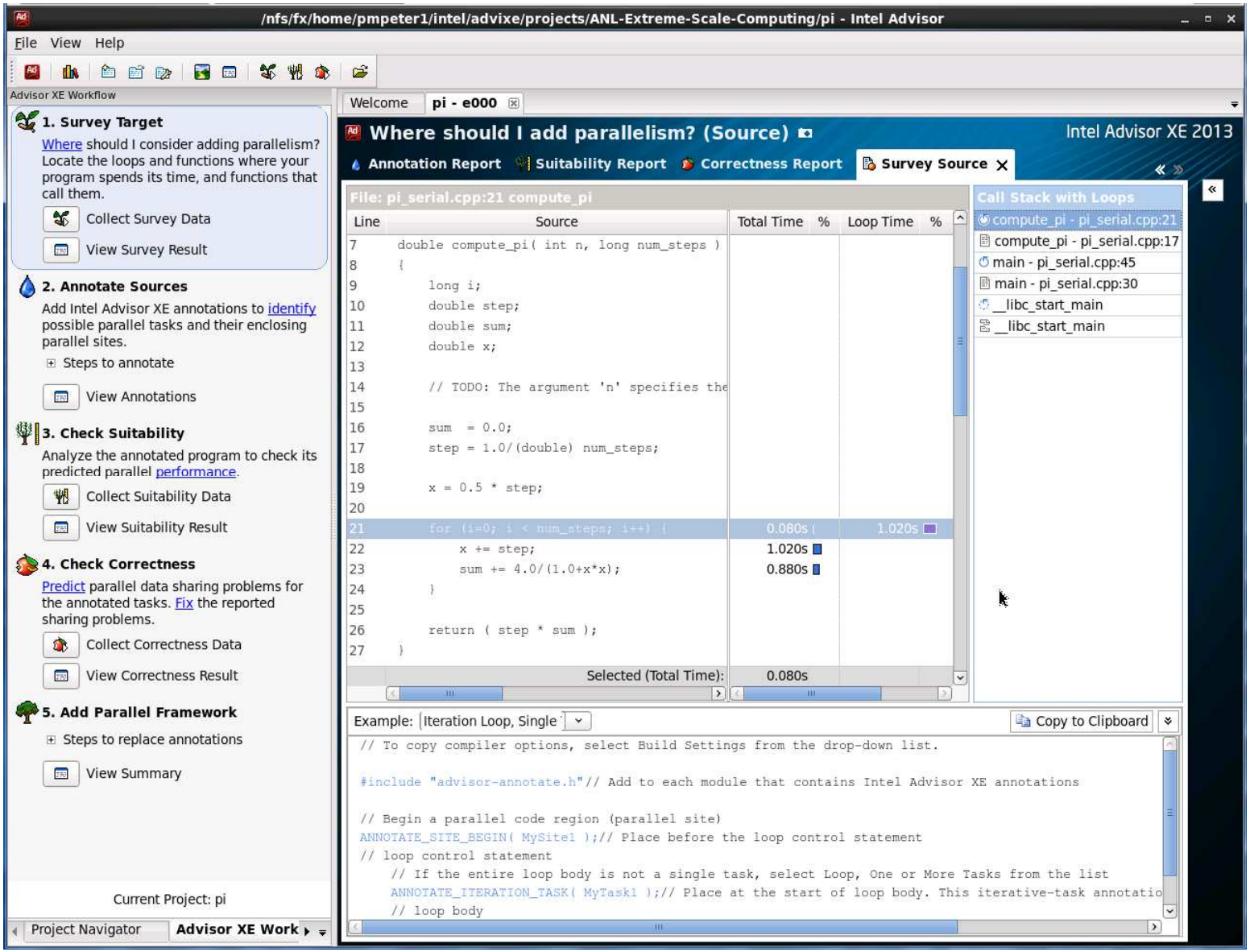
Until you have a plausible parallelism model that can achieve the benefits you want, it does not pay to check if it can be made correct

(on an ideal machine)

Step: Survey Target



Step: Survey Target – View Sources



Survey – How Does it Work?

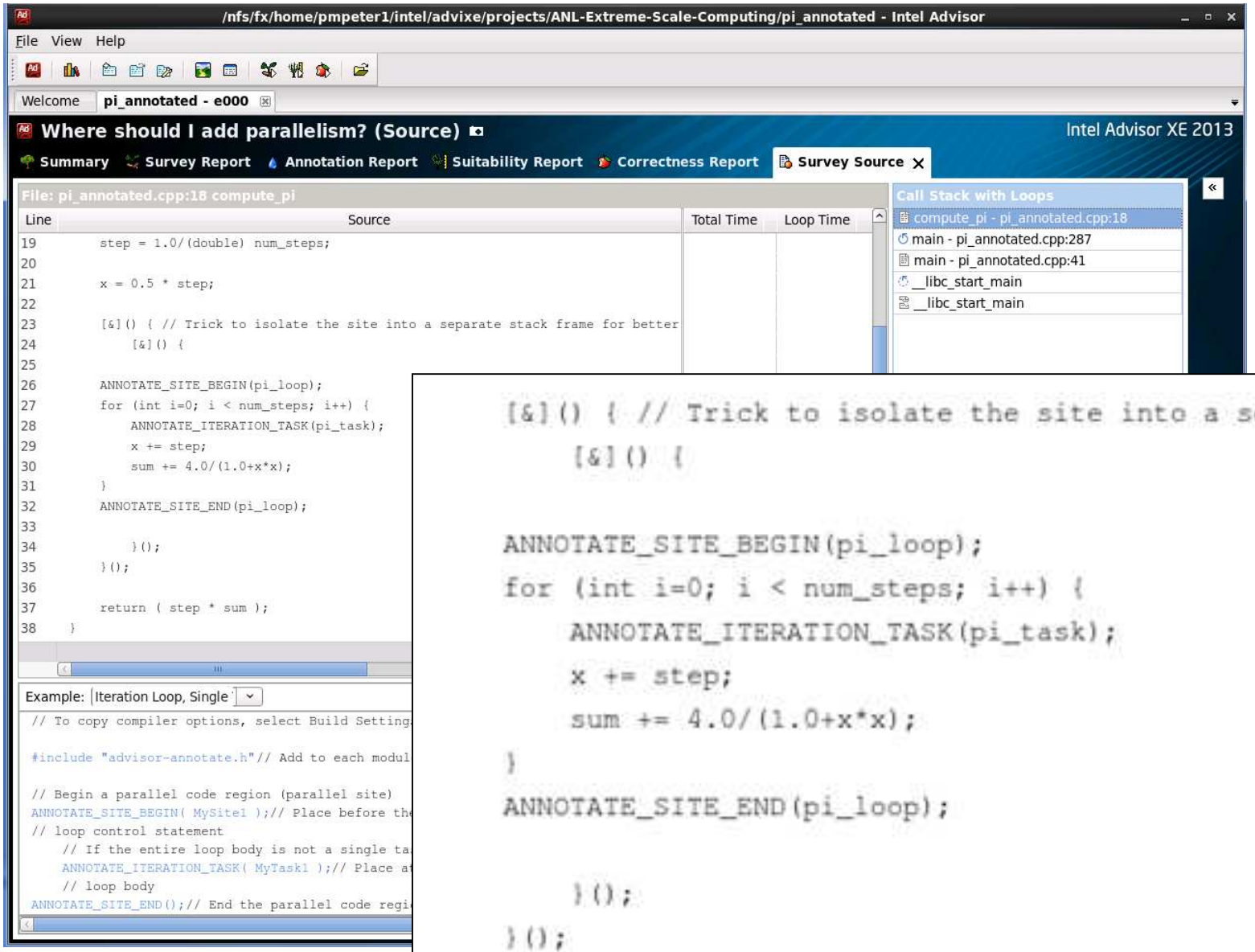
- Statistical Call Stack Sampling
- Focus on top-down inclusive execution time

- Define a periodic timer to sample IP addresses
- Unwind the call-stack at sample points

- Statically analyze the binary to detect loops

- Display the aggregate time a sample:
  - hits a basic block (IP or call stack-frame)
  - a call-stack frame intersects a loop

Step: Annotate Sources

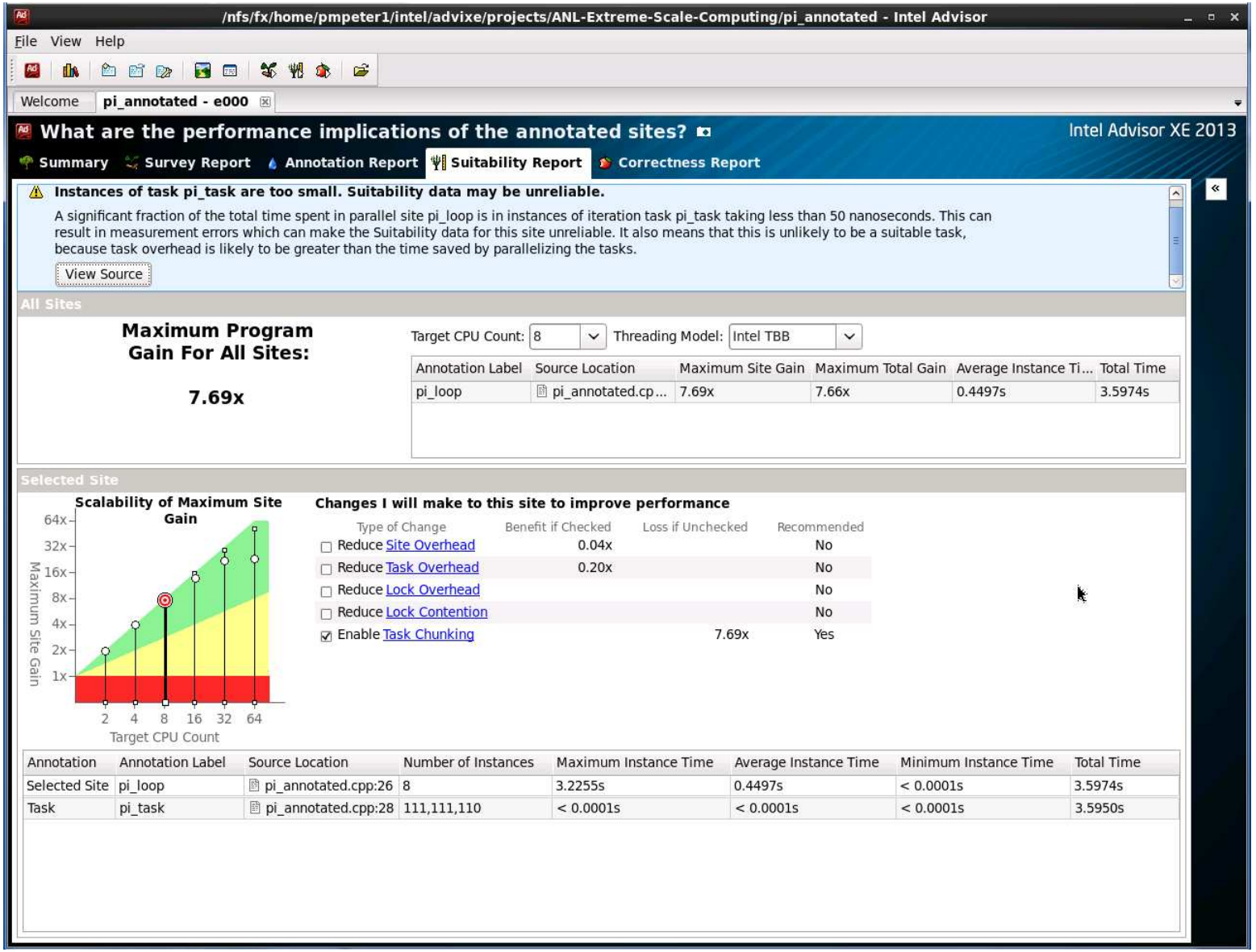



```
[&]() { // Trick to isolate the site into a separate
    [&]() {

ANNOTATE_SITE_BEGIN(pi_loop);
for (int i=0; i < num_steps; i++) {
    ANNOTATE_ITERATION_TASK(pi_task);
    x += step;
    sum += 4.0/(1.0+x*x);
}
ANNOTATE_SITE_END(pi_loop);

    }();
}();
```

Step: Check Suitability

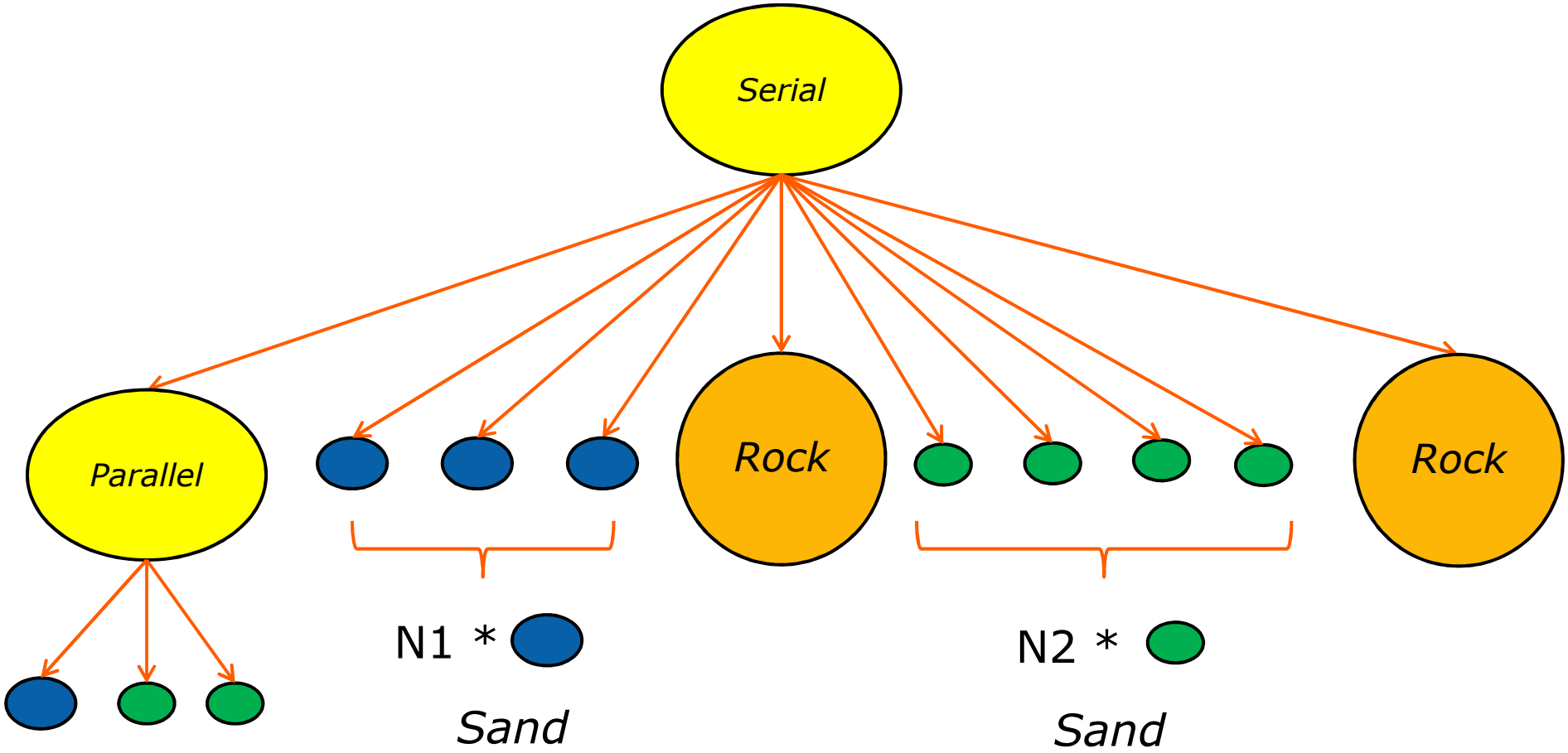


Suitability – How Does It Work?

- Performance Modeling
- Gather an event trace of modeling annotations at runtime
- Build a compressed execution tree
- Simulate the execution tree
  - On an ideal parallel machine
  - Under varying number of threads
  - With varying assumptions about overhead and scheduling
- Display the results and show how to improve the model

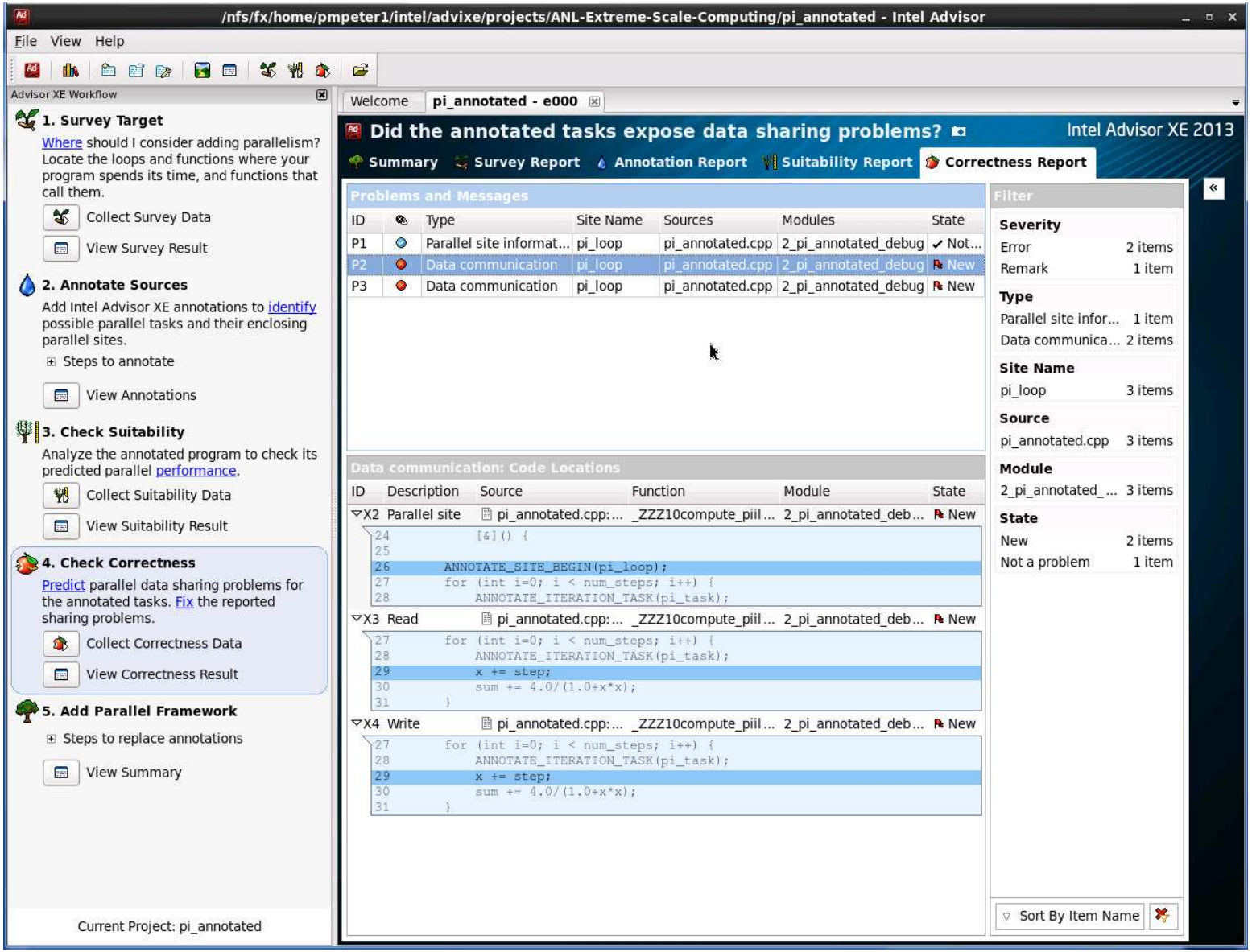
- The purpose is to check if the performance model is "suitable" as a starting point for parallelization
- It is not designed as an accurate performance prediction
  - Tree compression, Greedy Scheduling,
  - Ideal machine, No memory model, …

Suitability – *Rocks* and *Sand* Trace Compression



Count the *rocks*, but weigh the *sand*

Step: Check Correctness



Correctness – How Does It Work?

- Memory Trace Analysis
- Only instrument the program when inside a SITE

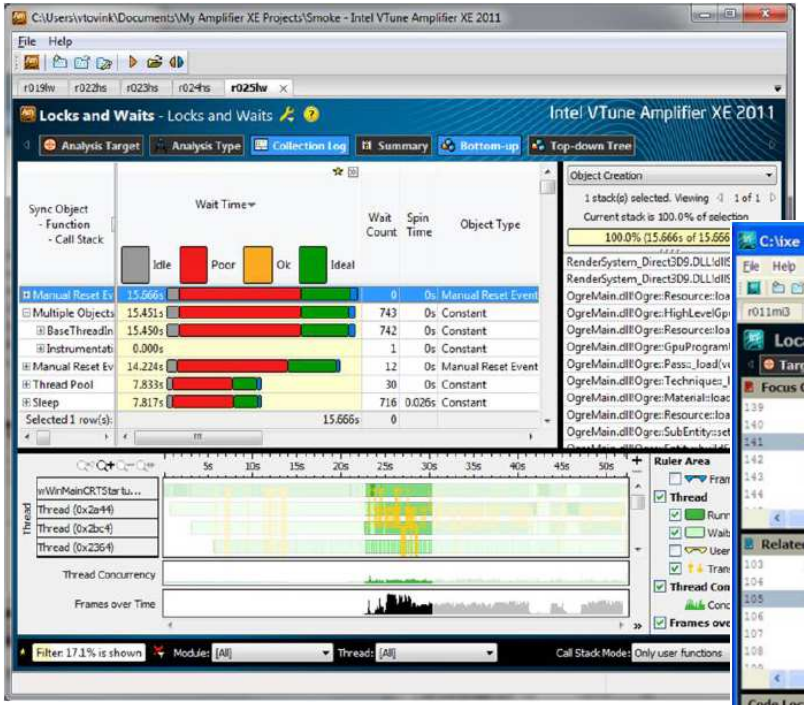
- Record a history of prior memory accesses
- Record the SITE/TASK structure

- When the next memory location is accessed, check:
  - If the prior access was in a concurrent TASK
  - If the prior access was in an equivalent LOCK context
  - Either this access or the prior access was a write

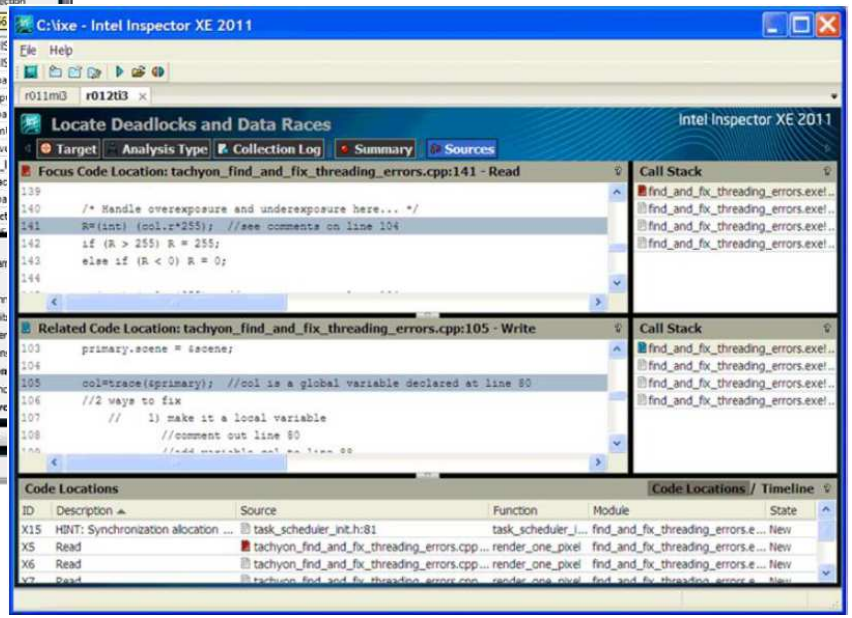
- Tell the user about
  - Data Communication - RAW dependence
  - Memory Reuse – WAR or WAW dependence
  - Inconsistent LOCK usage

What Is The Next Step?

- Great parallel performance requires real hardware on which to tune your implementations
- Parallelism is not just relaxation of serial algorithms



VTune Amplifier XE



Inspector XE

Writing Parallel Programs That Work

1. You need the ability to express your computation
   … usually serially

2. You need to understood the serial code
   … as a specification of what computations must happen

3. You should create a parallel model of the code
   … Intel® Advisor XE can be very helpful

4. You then express the parallel computation
   … with your favorite parallel framework

5. You should finish by tuning on parallel hardware
   … Intel® Parallel Studio XE can be very helpful

Intel® Parallel Studio 2013 XE

- More information about Parallel Studio XE and Advisor XE is available online, including a 30-day free trial

www.intel.com/go/parallel

- Supports Microsoft Visual Studio* 2005, 2008 and 2010.
- Support Linux

Optimization Notice

Optimization Notice

Intel compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the "Intel Compiler User and Reference Guides" under "Compiler Options." Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel compilers, associated libraries and associated development tools 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 Intel Streaming SIMD Extensions 2 (Intel SSE2), Intel Streaming SIMD Extensions 3 (Intel SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel 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.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20110228

Legal Disclaimer

http://intel.com/software/products

Step: Add Parallel Framework

- Final step: Convert annotations into uses of a parallel framework
- Examples in three parallel frameworks are used
  - Intel® Cilk™ Plus
  - Intel® Threading Building Blocks (Intel® TBB)
  - OpenMP*
- Converting a parallel model into explicitly parallel implementation is straightforward
- Each parallel framework you might use has concepts similar to a **SITE**, **TASK**, or **LOCK**
- Learning how common parallel framework features match annotations allows easy conversion to an explicitly parallel implementation

Parallel Framework: OpenMP

Specialized syntax can represent our parallel models efficiently

```
#include <advisor-annotate.h>
…
ANNOTATE_SITE_BEGIN(MyLoopSite);
for (int I = 0; I < N; ++I) {
   ANNOTATE_TASK_BEGIN(MyLoopIteration);
      Work( &A[I] );
   ANNOTATE_TASK_END();
}
ANNOTATE_SITE_END();
…
```

```
#pragma parallel for
   for (int I = 0; I < N; ++I) {
      Work( &A[I] );
   }
```

Parallel Framework: Intel® Cilk™ Plus

Specialized syntax can represent our parallel models efficiently

```
#include <advisor-annotate.h>
…
ANNOTATE_SITE_BEGIN(MyLoopSite);
for (int I = 0; I < N; ++I) {
   ANNOTATE_TASK_BEGIN(MyLoopIteration);
      Work( &A[I] );
   ANNOTATE_TASK_END();
}
ANNOTATE_SITE_END();
…
```

```
cilk_for (int I = 0; I < N; ++I)
        Work( &A[I] );
```

Parallel Framework:
Intel® Threading Building Blocks (Intel® TBB)

Libraries can also represent our parallel models efficiently

```
#include <advisor-annotate.h>
…
ANNOTATE_SITE_BEGIN(MyLoopSite);
for (int I = 0; I < N; ++I) {
   ANNOTATE_TASK_BEGIN(MyLoopIteration);
      Work( &A[I] );
   ANNOTATE_TASK_END();
}
ANNOTATE_SITE_END();
…
```

```
#include "tbb/parallel_for.h"
…
tbb::parallel_for (0, N, 1,
   [=](int I) { Work ( &A[I] ); } );
….
```

Parallel Framework: OpenMP

Exploit unique parallel framework features

```
#include <advisor-annotate.h>
…
ANNOTATE_ACQUIRE_BEGIN(0);
std::cout << A[I] << std::endl;
ANNOTATE_RELEASE_END(0);
…
```

```
#include <omp.h>
…
#pragma omp critical
    std::cout << A[I] << std::endl;
…
```

Parallel Framework: Intel® Cilk™ Plus (cont.)

Exploit unique parallel framework features

```
#include <advisor-annotate.h>
…
ANNOTATE_ACQUIRE_BEGIN(0);
std::cout << A[I] << std::endl;
ANNOTATE_RELEASE_END(0);
…
```

```
#include <cilk/reducer_ostream.h>
…
cilk::reducer_ostream cout_reducer(std::cout);
…
cout_reducer << A[I] << std::endl;
…
```

Parallel Framework:
OpenMP

Generic models can translate to specialized implementations

```
#include <advisor-annotate.h>
…
ANNOTATE_LOCK_ACQUIRE(0);
globalCounter = globalCounter + 1;
ANNOTATE_LOCK_RELEASE(0);
…
```

```
#include <omp.h>
…
#pragma atomic
    ++globalCounter;
…
```

Parallel Framework:
Intel® TBB (cont.)

Generic models can translate to specialized implementations

```
#include <advisor-annotate.h>
…
ANNOTATE_LOCK_ACQUIRE(0);
globalCounter = globalCounter + 1;
ANNOTATE_LOCK_RELEASE(0);
…
```

```
#include “tbb/atomic.h”
…
tbb::atomic<int> globalCounter;
…
++globalCounter;
…
```

About the speaker

Paul Petersen is a Sr. Principal Engineer in the Software and Solutions Group (SSG) at Intel. He received a Ph.D. degree in Computer Science from the University of Illinois in 1993.

After UIUC, he was employed at Kuck and Associates, Inc. (KAI) working on auto-parallelizing compiler (KAP), and was involved in the early definition and implementations of OpenMP. While at KAI, he developed the Assure line of parallelization/correctness products, for Fortran, C++ and Java.

In 2000, Intel Corporation acquired KAI, and he joined the software tools group. At Intel, he worked with the tools group to create the Thread Checker products, which evolved into the Inspector and Advisor components of the Intel® Parallel Studio. Inspector uses dynamic binary instrumentation to detect memory and concurrency bugs, and Advisor uses similar techniques along with performance measurement and modeling to assist developers in transforming existing serial applications to be ready for parallel execution.

- **Abstract:** Serial algorithms typically run very inefficiently on parallel machines. This may sound like an obvious statement, but it is the root cause of why parallel programming is considered to be difficult. The current state of the computer industry is still that almost all programs in existence are serial. To address this situation, Intel has created Parallel Studio XE, and in particular Advisor XE.
- This talk will describe the techniques used in Advisor XE to provide a developer with the tools necessary to understand the limitations of the existing serial algorithms. Once the limitations are known the developer can refactor the algorithms and reanalyze the resulting code to see if it could run effectively on parallel hardware. Almost all implementations of serial algorithms are serial for a reason, and the tools available in Advisor XE help the user expose these reasons so that appropriate rewrites can be done.

- **Bio:** Paul Petersen is a Sr. Principal Engineer in the Software and Solutions Group (SSG) at Intel. He received a Ph.D. degree in Computer Science from the University of Illinois in 1993. After UIUC, he was employed at Kuck and Associates, Inc. (KAI) working on auto-parallelizing compiler (KAP), and was involved in the early definition and implementations of OpenMP. While at KAI, he developed the Assure line of parallelization/correctness products, for Fortran, C++ and Java. In 2000, Intel Corporation acquired KAI, and he joined the software tools group. At Intel, he worked with the tools group to create the Thread Checker products, which evolved into the Inspector and Advisor components of the Intel® Parallel Studio. Inspector uses dynamic binary instrumentation to detect memory and concurrency bugs, and Advisor uses similar techniques along with performance measurement and modeling to assist developers in transforming existing serial applications to be ready for parallel execution.