



Tools for OpenMP Programming

PPCES 2017

Tim Cramer
Dirk Schmidl

■ Intel Inspector XE

→ Overview

→ Live Demo

■ Intel VTune Amplifier XE

→ Overview

→ Live Demo

Race Condition

- **Data Race: the typical OpenMP programming error, when:**
 - two or more threads access the same memory location, and
 - at least one of these accesses is a write, and
 - the accesses are not protected by locks or critical regions, and
 - the accesses are not synchronized, e.g. by a barrier.
- **Non-deterministic occurrence: e.g. the sequence of the execution of parallel loop iterations is non-deterministic and may change from run to run**
- **In many cases *private* clauses, *barriers* or *critical regions* are missing**
- **Data races are hard to find using a traditional debugger**
 - Use the *Intel Inspector XE*

■ Detection of

- Memory Errors
- Deadlocks
- Data Races

■ Support for

- Linux (32bit and 64bit) and Windows (32bit and 64bit)
- WIN32-Threads, Posix-Threads, Intel Threading Building Blocks and OpenMP

■ Features

- Binary instrumentation gives full functionality
- Independent stand-alone GUI for Windows and Linux
- Memory error detection

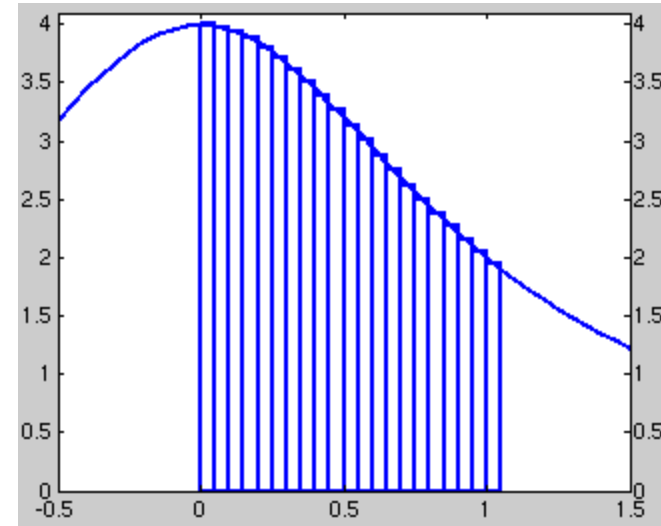
PI Example Code

```
double f(double x)
{
    return (4.0 / (1.0 + x*x));
}
```

$$\pi = \int_0^1 \frac{4}{1+x^2}$$

```
double CalcPi (int n)
{
    const double fH = 1.0 / (double) n;
    double fSum = 0.0;
    double fX;
    int i;

```



```
#pragma omp parallel for private(fX,i) reduction(+:fSum)
for (i = 0; i < n; i++)
{
    fX = fH * ((double)i + 0.5);
    fSum += f(fX);
}
return fH * fSum;
}
```

PI Example Code

```
double f(double x)
{
    return (4.0 / (1.0 + x*x));
}
```

```
double CalcPi (int n)
{
    const double fH = 1.0 / (double) n;
    double fSum = 0.0;
    double fX;
    int i;
```

```
#pragma omp parallel for private(fX,i) reduction(+:fSum)
for (i = 0; i < n; i++)
{
    fX = fH * ((double)i + 0.5);
    fSum += f(fX);
}
return fH * fSum;
}
```

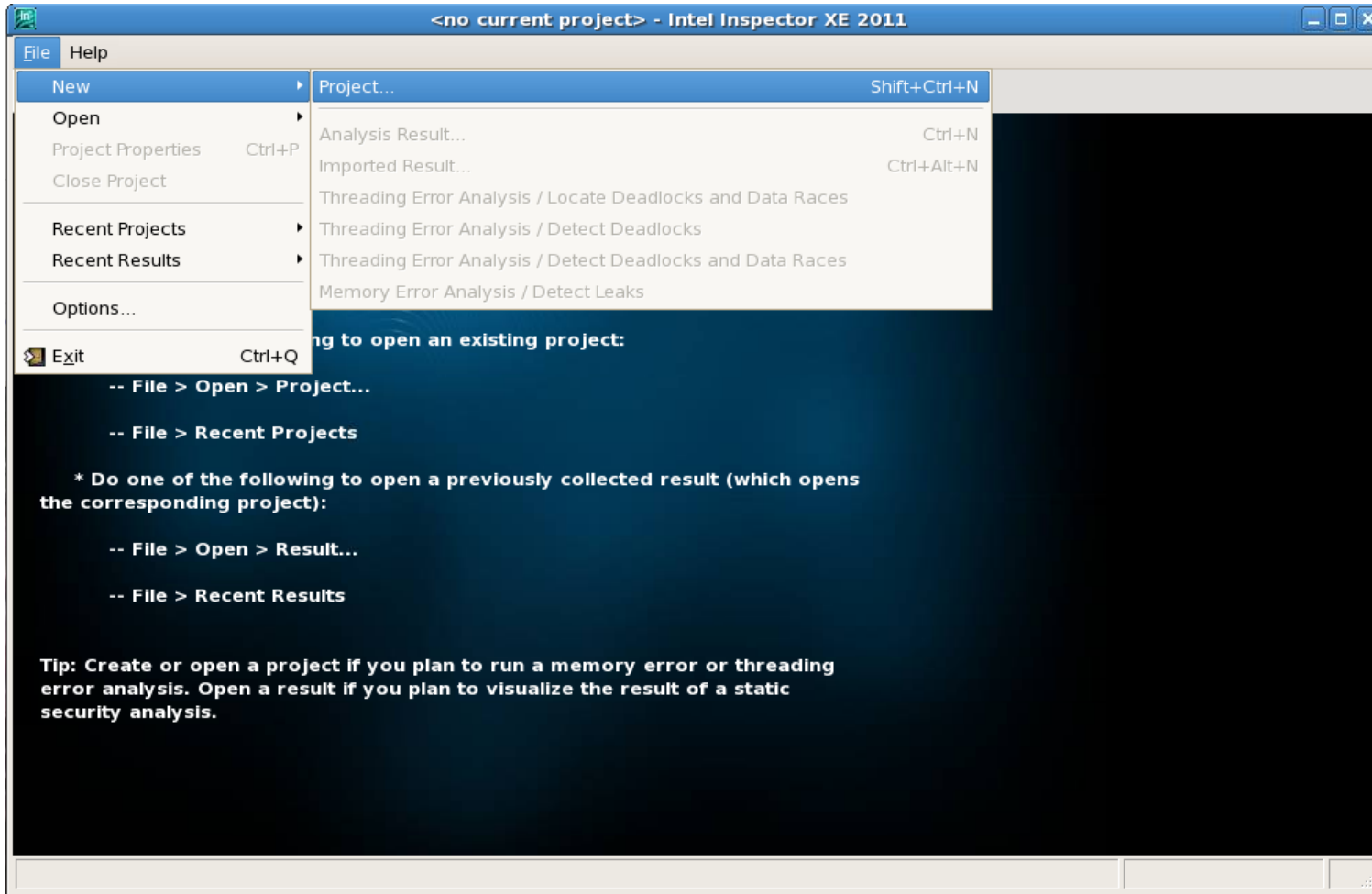
What if we
would have
forgotten
this?

Live Demo

Intel Inspector XE

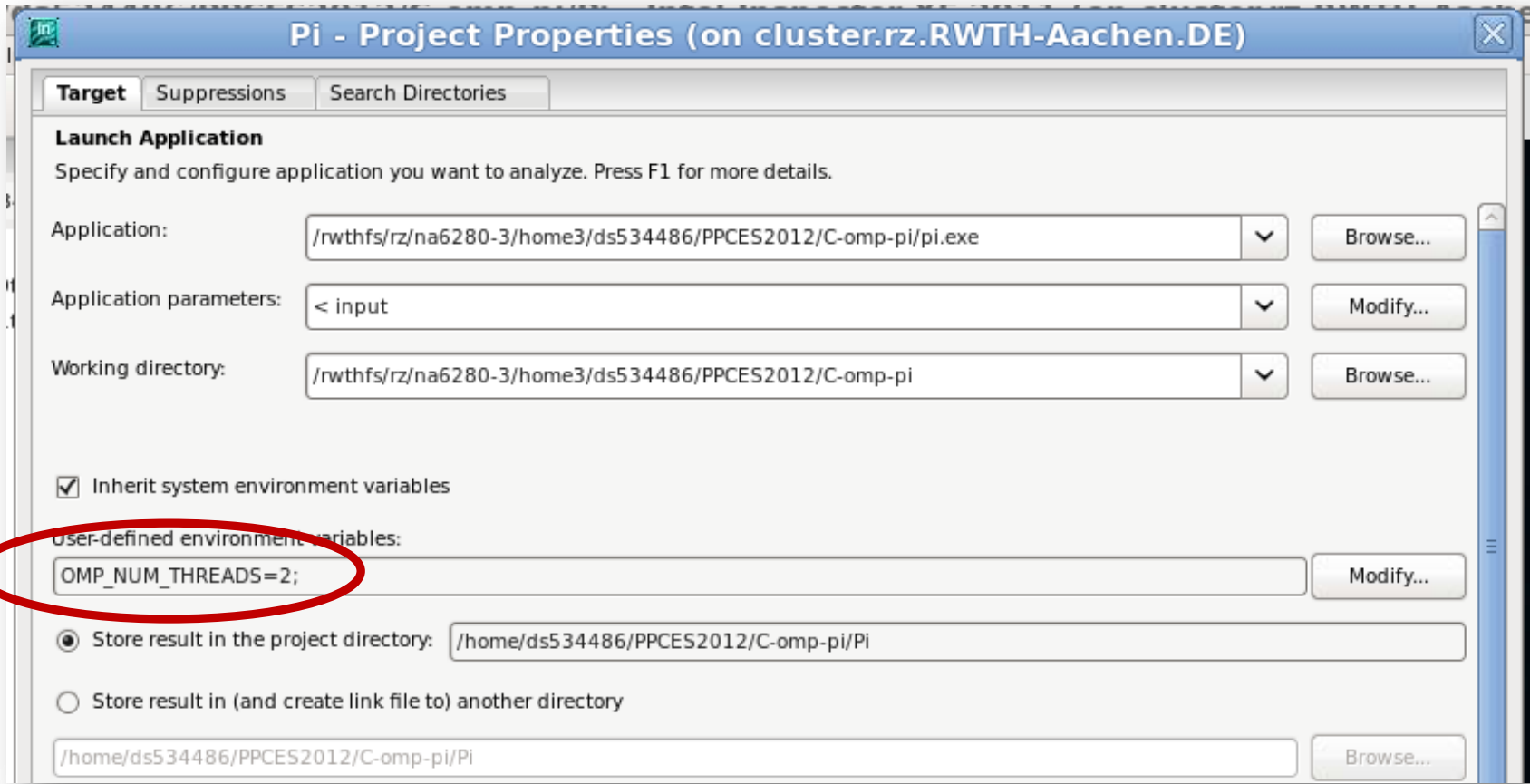
Inspector XE – Create Project

\$ module load intelixe ; inspxe-gui



Inspector XE – Create Project

- ensure that multiple threads are used
- choose a small dataset (really!), execution time can increase 10X – 1000X



Inspector XE – Configure Analysis

Threading Error Analysis Modes

1. Detect Deadlocks
2. Detect Deadlocks and Data Races
3. Locate Deadlocks and Data Races



more details,
more overhead

Configure Analysis Type

Analysis Type

- Memory Error Analysis
 - Detect Leaks
 - Detect Memory Problem
 - Locate Memory Problem
- Threading Error Analysis
 - Detect Deadlocks
 - Detect Deadlocks and Data Races
 - Locate Deadlocks and Data Races**
 - Custom Analysis Types

Locate Deadlocks and Data Races

Widest scope threading error analysis type. Maximizes the load on the system. Maximizes the time required to perform the analysis. Maximizes the chances the analysis will fail because the system may run out of resources. Press F1 for more details.

Terminate on deadlock

Stack frame depth: 16

Scope: Normal

Start

Stop

Take Snapshot

Close

Inspector XE – Results

- 1 detected problems
- 2 filters
- 3 code location

The screenshot displays the Intel Inspector XE 2011 interface. The main window title is "r001ti3". The top navigation bar includes "Locate Deadlocks and Data Races" and "Intel Inspector XE 2011". Below this, there are tabs for "Target", "Analysis Type", "Collection Log", and "Summary".

The "Problems" pane shows a single problem, P1, which is a "Data race" detected in "pi.c" within the "pi.exe" module, with a state of "New". A yellow circle with the number "1" is placed over this pane.

The "Code Locations" pane shows two locations for the data race, both in "pi.c:71" within the "CalcPi" function of "pi.exe". The code snippets are:

```
Code Locations / Timeline
ID Description Source Function Module
vX1 Read pi.c:71 CalcPi pi.exe
69 {
70 fX = fH * ((double)i + 0.5);
71 fSum += f(fX);
72 }
73 return fH * fSum;
vX2 Write pi.c:71 CalcPi pi.exe
69 {
70 fX = fH * ((double)i + 0.5);
71 fSum += f(fX);
72 }
73 return fH * fSum;
```

A yellow circle with the number "3" is placed over the code snippets in the "Code Locations" pane.

The "Filters" pane on the right shows a list of filters with their counts:

Severity	Count
Error	1 item(s)
Problem	Count
Data race	1 item(s)
Source	Count
pi.c	1 item(s)
Module	Count
pi.exe	1 item(s)
State	Count
New	1 item(s)
Suppressed	Count
Not suppressed	1 item(s)
Investigated	Count
Not investigated	1 item(s)

A yellow circle with the number "2" is placed over the "Filters" pane.

Inspector XE – Results

1 Timeline view

The screenshot displays the Intel Inspector XE 2011 interface. The main window title is "r001ti3". The application title is "Locate Deadlocks and Data Races". The top navigation bar includes "Target", "Analysis Type", "Collection Log", and "Summary".

The "Problems" pane shows a table with the following data:

ID	Problem	Sources	Modules	State
P1	Data race	pi.c	pi.exe	New

The "Filters" pane on the right shows the following counts:

Severity	Count
Error	1 item(s)

Problem	Count
Data race	1 item(s)

Source	Count
pi.c	1 item(s)

Module	Count
pi.exe	1 item(s)

State	Count
New	1 item(s)

Suppressed	Count
Not suppressed	1 item(s)

Investigated	Count
Not investigated	1 item(s)

The "Timeline" pane at the bottom shows a horizontal axis with percentage markers from 0.05% to 0.4%. Two threads are visible: "main (16217)" and "OMP Worker ...". Blue bars represent the execution of these threads, with diamond markers indicating the data race event. A yellow circle with the number "1" is overlaid on the bottom right corner of the timeline pane.

Inspector XE – Results

- 1 Source Code producing the issue – double click opens an editor
- 2 Corresponding Call Stack

The screenshot displays the Intel Inspector XE 2011 interface. The top bar shows the target 'r001ti3' and the title 'Locate Deadlocks and Data Races'. Below this are tabs for 'Target', 'Analysis Type', 'Collection Log', 'Summary', and 'Sources'. The main area is divided into four panes:

- Focus Code Location: pi.c:71 - Write:** Shows a C++ code snippet with a for loop. Line 71, `fSum += f(fX);`, is highlighted. A yellow circle with the number '1' is next to line 74.
- Call Stack:** Shows the call stack for the selected code location. The stack includes: `pi.exe!CalcPi - pi.c:71`, `pi.exe!CalcPi - pi.c:67`, `pi.exe!main - pi.c:40`, and `pi.exe!_start - pi.exe:2340`. A yellow circle with the number '2' is next to the top entry.
- Related Code Location: pi.c:71 - Write:** Shows the same code snippet as the focus pane. Line 71 is highlighted. A yellow circle with the number '1' is next to line 74.
- Call Stack:** Shows the same call stack as the focus pane. A yellow circle with the number '2' is next to the top entry.

At the bottom, the 'Code Locations' pane shows a table of code locations:

ID	Description	Source	Function	Module
X1	Read	pi.c:71	CalcPi	pi.exe
X2	Write	pi.c:71	CalcPi	pi.exe
X3	Write	pi.c:71	CalcPi	pi.exe

Inspector XE – Results

- 1 Source Code producing the issue – double click opens an e
- 2 Corresponding Call Stack

The missing reduction is detected.

The screenshot displays the Intel Inspector XE 2011 interface. The top bar shows the target 'r001ti3' and navigation tabs for 'Target', 'Analysis Type', 'Collection Log', 'Summary', and 'Sources'. The main area is divided into three panels:

- Focus Code Location: pi.c:71 - Write:** Shows a C code snippet with a for loop. Line 71, `fSum += f(fX);`, is highlighted with a yellow circle '1'. The corresponding call stack on the right (labeled '2') shows: `pi.exe!CalcPi - pi.c:71`, `pi.exe!CalcPi - pi.c:67`, `pi.exe!main - pi.c:40`, and `pi.exe!_start - pi.exe:2340`.
- Related Code Location: pi.c:71 - Write:** Shows the same code snippet. Line 71 is highlighted with a yellow circle '1'. The corresponding call stack on the right (labeled '2') is identical to the focus code location.
- Code Locations:** A table listing the detected code locations.

ID	Description	Source	Function	Module
X1	Read	pi.c:71	CalcPi	pi.exe
X2	Write	pi.c:71	CalcPi	pi.exe
X3	Write	pi.c:71	CalcPi	pi.exe

Command Line Tool – inspxe-cl

Threading Error Analysis Modes

1. Detect Deadlocks (ti1)
2. Detect Deadlocks and Data Races (ti2)
3. Locate Deadlocks and Data Races (ti3)

```
$ inspxe-cl -collect ti3 -- pi.exe < input
```

```
$ inspxe-cl -report problems ...
```

Data collection without GUI allows to use batch jobs.

Viewing results in text mode is helpful, when remote connections are slow.

Intel VTune Amplifier XE

- **Performance Analyses for**
 - Serial Applications
 - Shared Memory Parallel Applications
- **Sampling-based measurements**
- **Features:**
 - Hot Spot Analysis
 - Concurrency Analysis
 - Wait
 - Hardware Performance Counter Support

Prerequisites:

- ssh -Y login-t
- module load intelvtune

Stream

- Standard Benchmark to measure memory performance.
- Version is parallelized with OpenMP.

Measures Memory bandwidth for:

$y=x$ (copy)

$y=s*x$ (scale)

$y=x+z$ (add)

$y=x+s*z$ (triad)

```
#pragma omp parallel for
for (j=0; j<N; j++)
    b[j] = scalar*c[j];
```

for double vectors x,y,z and scalar double value s

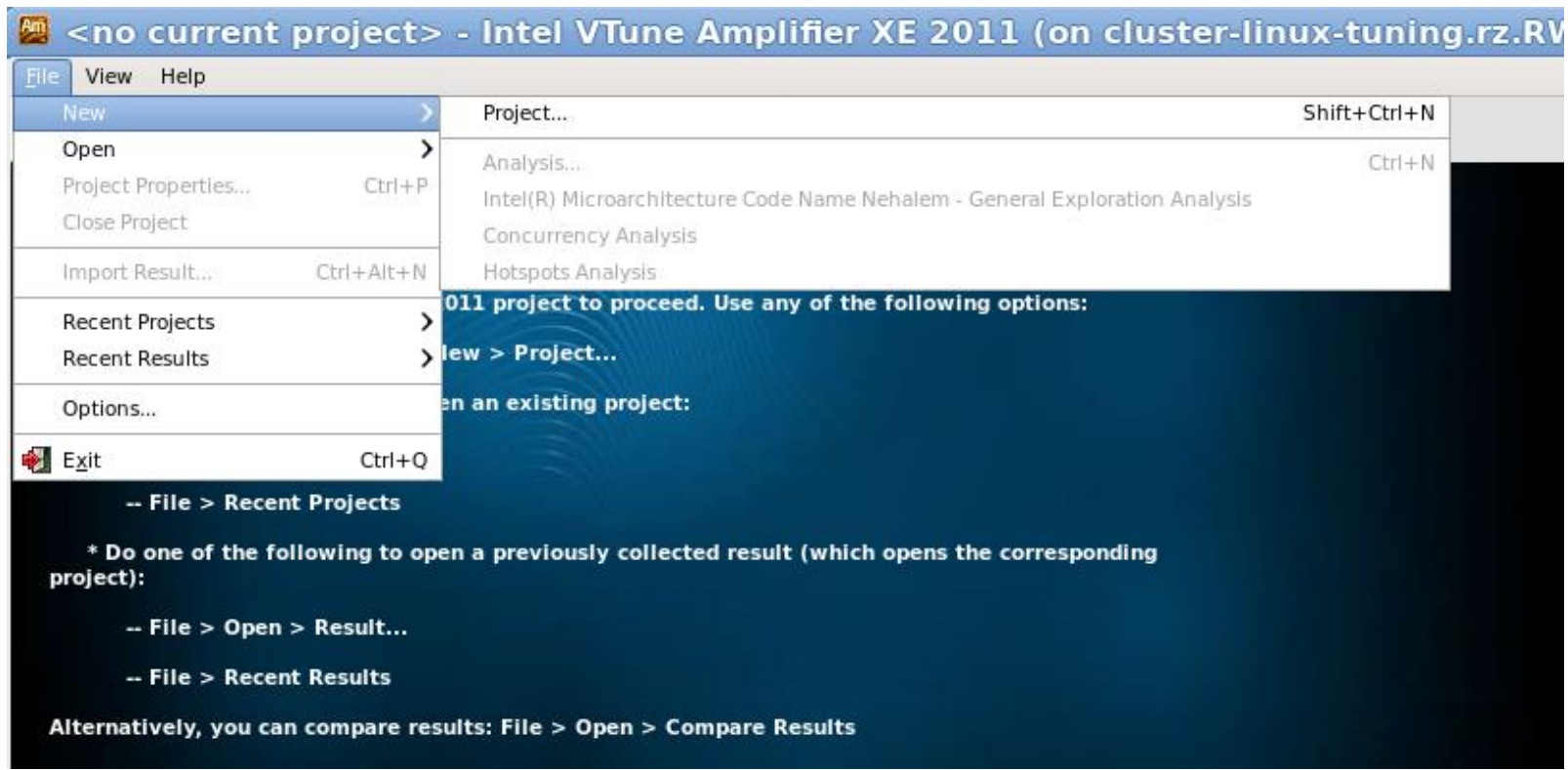
Function	Rate (MB/s)	Avg time	Min time	Max
Copy:	33237.0185	0.0050	0.0048	0.0055
Scale:	33304.6471	0.0049	0.0048	0.0059
Add:	35456.0586	0.0070	0.0068	0.0073
Triad:	36030.9600	0.0069	0.0067	0.0072

Live Demo

Intel VTune Amplifier XE

Intel VTune Amplifier XE – Create Project

- Create a Project in the same way as with the inspector.
- Executable should be build with optimization.
- Use a reasonable sized data set.



Amplifier XE – Measurement Runs

- 1 Basic Analysis Types
- 2 Hardware Counter Analysis Types (e.g., Westmere Architecture on cluster-linux-tuning)
- 3 Analysis for Intel Xeon Phi coprocessors, choose this for OpenMP target programs.

The screenshot shows the Intel VTune Amplifier XE 2013 interface. The 'Choose Analysis Type' dialog is open, displaying a tree view of analysis types. The 'Hotspots' option under 'Knights Corner Platform Analysis' is highlighted with a red circle and a yellow '3'. Other options are marked with yellow circles '1' and '2'.

Hotspots - Knights Corner Platform

Identify time-consuming code in your application. Advanced Hotspots analysis (formerly, Lightweight Hotspots) uses the kernel driver and extends the hotspots analysis by collecting call stacks, context switch and statistical call count data and analyzing the CPI (Cycles Per Instruction) metric. At the default level this analysis uses higher frequency sampling at lower overhead compared to Basic Hotspots...

List of Intel Xeon Phi coprocessor cards: 0

Analyze user tasks

Details

Events configured for CPU: Intel(R) Xeon(R) E5 processor

NOTE: For analysis purposes, Intel VTune Amplifier XE 2013 may adjust the Sample After values in the table below by a multiplier. The multiplier depends on the value of the Duration time estimate option specified in the Project Properties dialog.

Event Name	Sample After	Event Description
CPU_CLK_UNHALTED	10000000	
INSTRUCTIONS_EXECUTED	10000000	

Buttons: Start, Start Paused, Project Properties

Amplifier XE – Hotspot Analysis

The screenshot shows the 'Hotspots - Hotspots' window with the following sections:

- Elapsed Time: 1.294s** (marked with a yellow circle 1)
 - Total Thread Count: 12
 - CPU Time: 14.840s
 - Paused Time: 0s
- Top Hotspots** (marked with a yellow circle 2)

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in better performance.

Function	CPU Time
main	10.672s
__kmp_wait_sleep	2.438s
__kmp_x86_pause	1.100s
__kmp_execute_tasks	0.400s
__kmp_yield	0.120s
[Others]	0.110s
- Collection and Platform Info** (marked with a yellow circle 3)

This section provides information about this collection, including result set size and collection platform data.

Command Line: /rwthfs/rz/cluster/home/ds534486/PPCES2012/stream/stream.exe

Environment Variables: OMP_NUM_THREADS=12;

Frequency: 3.07 GHz

Logical CPU Count: 24

Operating System: Linux

Computer Name: cluster-linux-tuning.rz.RWTH-Aachen.DE

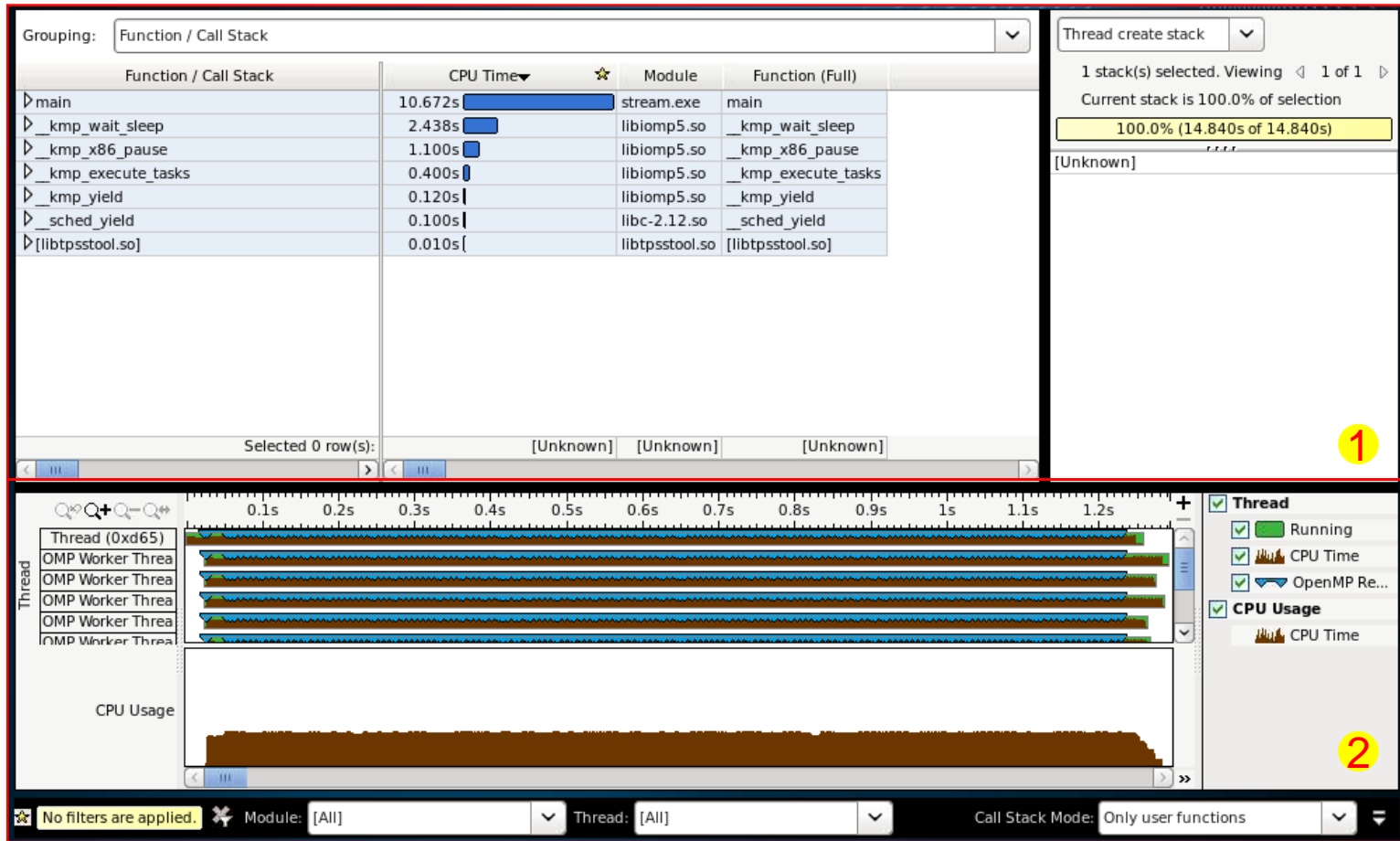
Result Size: 71 KB

Summary:

- 1 General Timing Information
- 2 Top Hotspots
- 3 Platform Information

Amplifier XE – Hotspot Analysis

- 1 Function Summary
- 2 Timeline View



Amplifier XE – Hotspot Analysis

Double clicking on a function opens source code view.

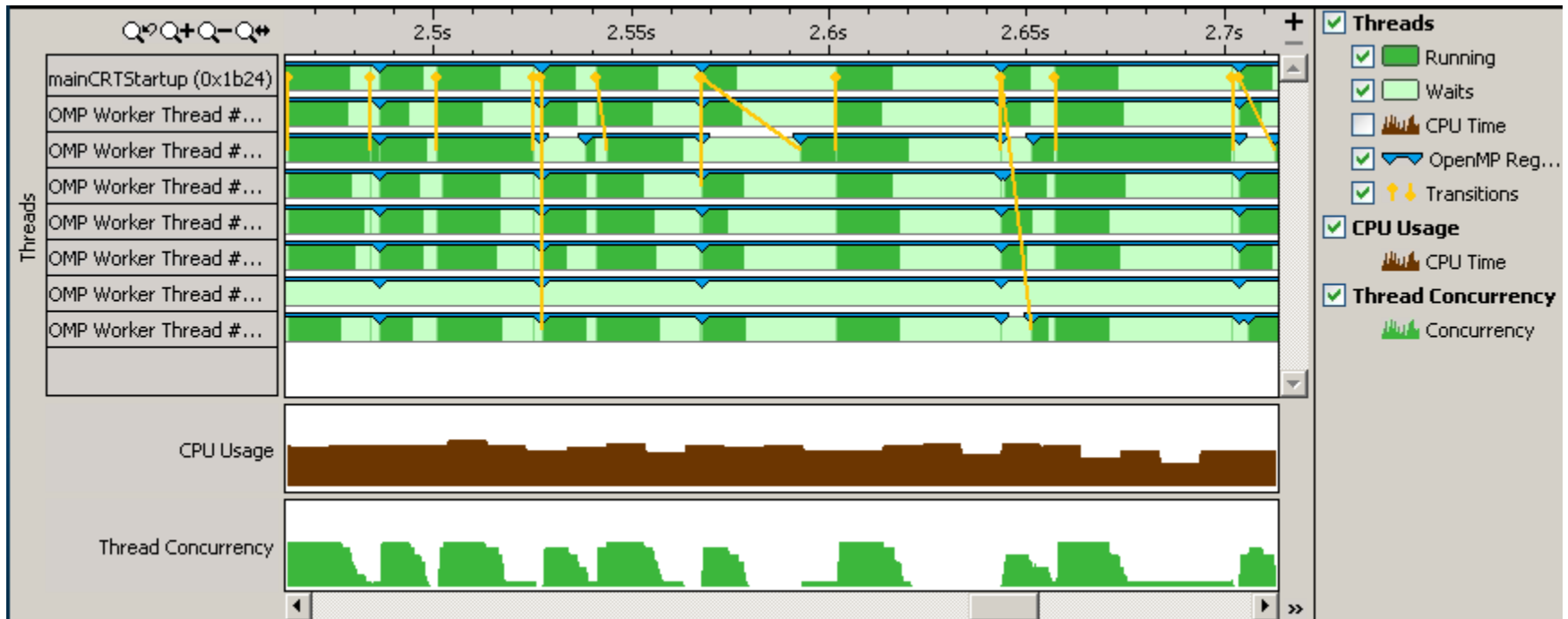
- 1 Source Code View (only if compiled with -g)
- 2 Hotspot: Add Operation of Stream
- 3 Metrics View

The screenshot displays the Amplifier XE interface with two main panels. The left panel shows the source code with line numbers 238 to 254. Line 241, containing the code `c[j] = a[j]+b[j];`, is highlighted in blue and marked with a yellow circle '2'. The right panel shows a 'CPU Time' view with a vertical bar chart. The bar for line 241 is significantly longer than others, indicating a hotspot, and is marked with a yellow circle '3'. Other bars are labeled with times like 0.010s, 0.140s, 0.160s, and 2.751s. A label 'Hotspots' with arrows points to the bars in the right panel. At the bottom of the right panel, it says 'Selected 1 row(s): 2.790s'. A yellow circle '1' is located at the bottom of the source code panel.

Line	Source	CPU Time
238	<code>#else</code>	
239	<code>#pragma omp parallel for</code>	0.010s
240	<code>for (j=0; j<N; j++)</code>	0.140s
241	<code>c[j] = a[j]+b[j];</code>	2.790s
242	<code>#endif</code>	
243	<code>times[2][k] = mysecond() - times[2][k];</code>	
244	<code>times[3][k] = mysecond();</code>	
245	<code>#ifdef TUNED</code>	
246	<code>tuned_STREAM_Triad(scalar);</code>	
247	<code>#else</code>	
248	<code>#pragma omp parallel for</code>	0.160s
249	<code>for (j=0; j<N; j++)</code>	2.751s
250	<code>a[j] = b[j]+scalar*c[j];</code>	
251	<code>#endif</code>	
252	<code>times[3][k] = mysecond() - times[3][k];</code>	
253	<code>}</code>	
254		

Amplifier XE – Locks and Waits Analysis

- Waiting time is shown in light green.
- Execution time is shown in dark green.
- CPU Usage and Thread Concurrency differ because waiting threads utilize a CPU.



Derived Metrics

■ **Clock cycles per Instructions (CPI)**

- CPI indicates if the application is utilizing the CPU or not
- Take care: Doing “something” does not always mean doing “something useful”.

■ **Floating Point Operations per second (FLOPS)**

- How many arithmetic operations are done per second?
- Floating Point operations are normally really computing and for some algorithms the number of floating point operations needed can be determined.

Amplifier XE – Hardware Counter

1 CPI rate (Clock cycles per instruction): In theory modern processors can finish 4 instructions in 1 cycle, so a CPI rate of 0.25 is possible. A value between 0.25 and 1 is often considered as good for HPC applications.

Elapsed Time: 1.872s

Hardware Event Count:	125,574,000,000
CPU_CLK_UNHALTED.THREAD:	6.3462e+10
INST_RETIRED.ANY:	6.2112e+10

1 CPI Rate: 1.022

The CPI may be too high. This could be caused by issues such as memory access stalls or branch mispredicts. Explore the other hardware-related metrics to identify what is causing the high CPI.

Retire Stalls: 0.570s

A high number of retire stalls is detected. This may result from branch mispredicts or memory access stalls. Use this metric to find where you have stalled instructions. Once you have identified the cause, you can optimize your code to reduce the number of retire stalls.

LLC Miss: 0.013s

LLC Load Misses Serviced By Remote DRAM: 0.001s

Instruction Starvation: 0.098s

Branch Mispredict: 0.001s

Execution Stalls: 0.288s

Summary

Correctness:

- **Data Races are very hard to find, since they do not show up every program run.**
- **Intel Inspector XE helps a lot in finding these errors.**
- **Use really small datasets, since the runtime increases significantly.**
- **If possible use non optimized code.**

Performance:

- **Start with simple performance measurements like hotspots analyses and then focus on these hot spots.**
- **In OpenMP applications analyze the waiting time of threads. Is the waiting time balanced?**
- **Hardware counters might help for a better understanding of an application, but they are hard to interpret.**