Introduction to OpenMP

Christian Terboven <terboven@itc.rwth-aachen.de>
Dirk Schmidl <schmidl@itc.rwth-aachen.de>

18.03.2015 / Aachen, Germany

Stand: 12.03.2015

Version 2.3

Agenda of the OpenMP Part

- **WED**
  - → 09:00h – 10:30h: Introduction to Parallel Programming with OpenMP I
  - → 11:00h – 12:30h: Introduction to Parallel Programming with OpenMP II
  - → 14:00h – 15:30h: Getting OpenMP up to Speed
  - → 16:00h – 17:30h: Advanced OpenMP Programming
- **WED evening: social event**
- **THU**
  - → 09:00h – 10:30h: Intel Xeon Phi Coprocessor
    OpenMP for Accelerators
  - → 14:00h – 15:30h: Vectorization with OpenMP
    Performance Analysis with LIKWID
  - → 16:00h – 17:30h: One (single) kernel for CPU, GPU and Xeon Phi

Introduction

History

- De-facto standard for Shared-Memory Parallelization.

- 1997: OpenMP 1.0 for FORTRAN
- 1998: OpenMP 1.0 for C and C++
- 1999: OpenMP 1.1 for FORTRAN (errata)

- 2000: OpenMP 2.0 for FORTRAN
- 2002: OpenMP 2.0 for C and C++
- 2005: OpenMP 2.5 now includes both programming languages.

- 05/2008: OpenMP 3.0 release
- 07/2011: OpenMP 3.1 release

- 07/2013: OpenMP 4.0 release

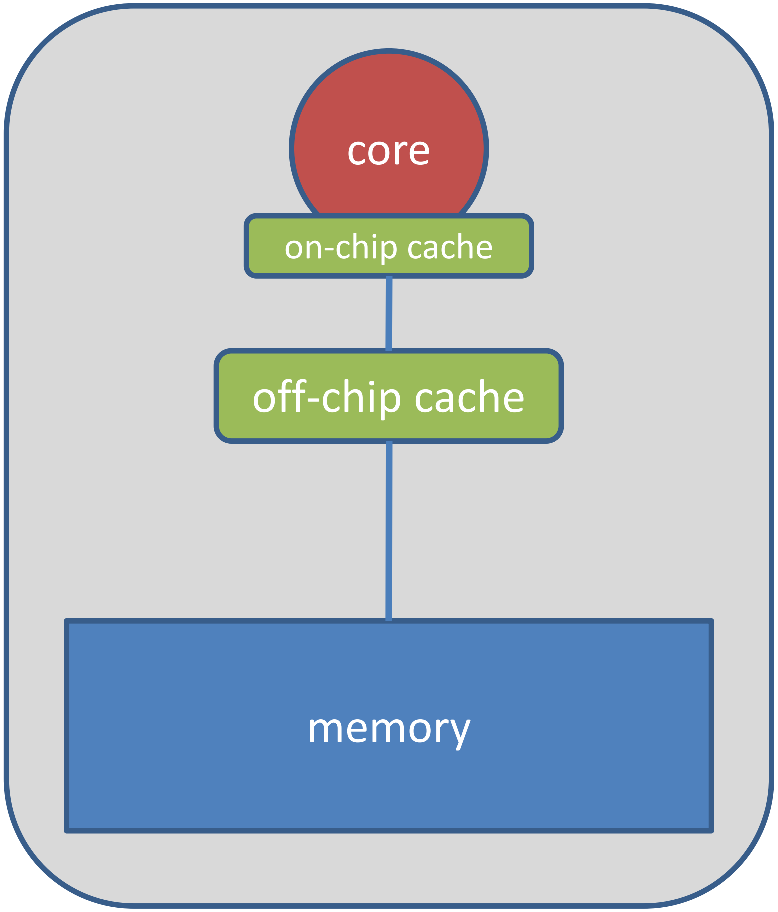
OpenMP™

http://www.OpenMP.org

RWTH Aachen University is a member of the OpenMP Architecture Review Board (ARB) since 2006.

Single Processor System (dying out)

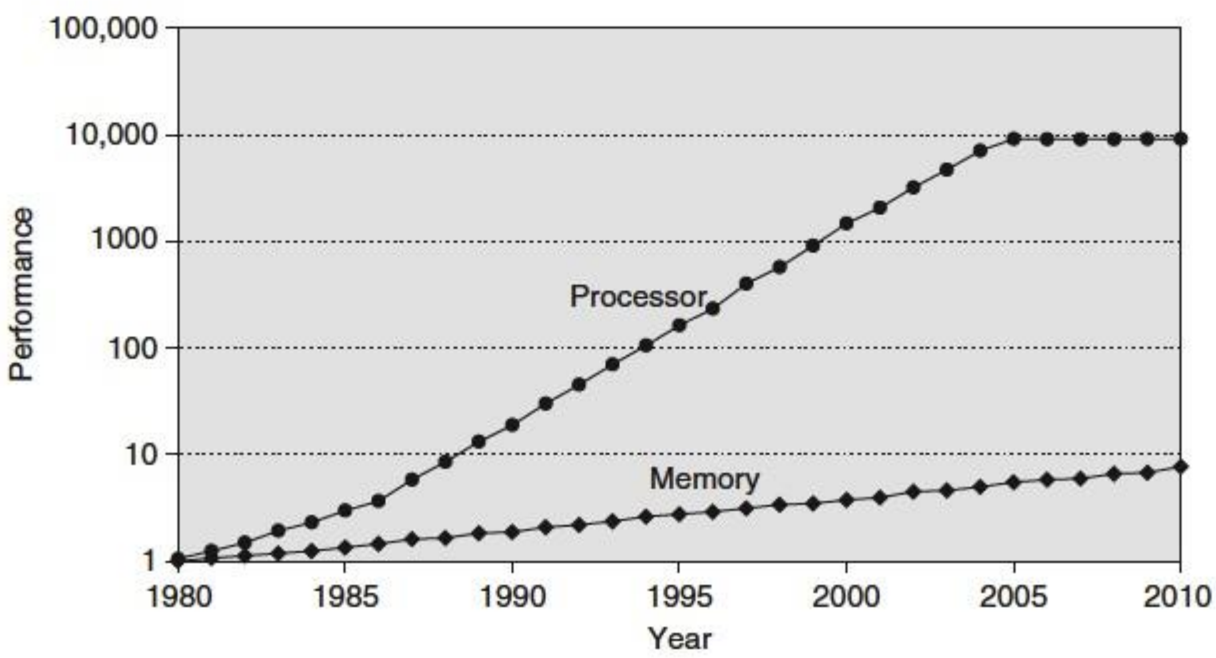
- **CPU is fast**
  - → Order of 3.0 GHz
- **Caches:**
  - → Fast, but expensive
  - → Thus small, order of MB
- **Memory is slow**
  - → Order of 0.3 GHz
  - → Large, order of GB



- **A good utilization of caches is crucial for good performance of HPC applications!**

Memory Bottleneck

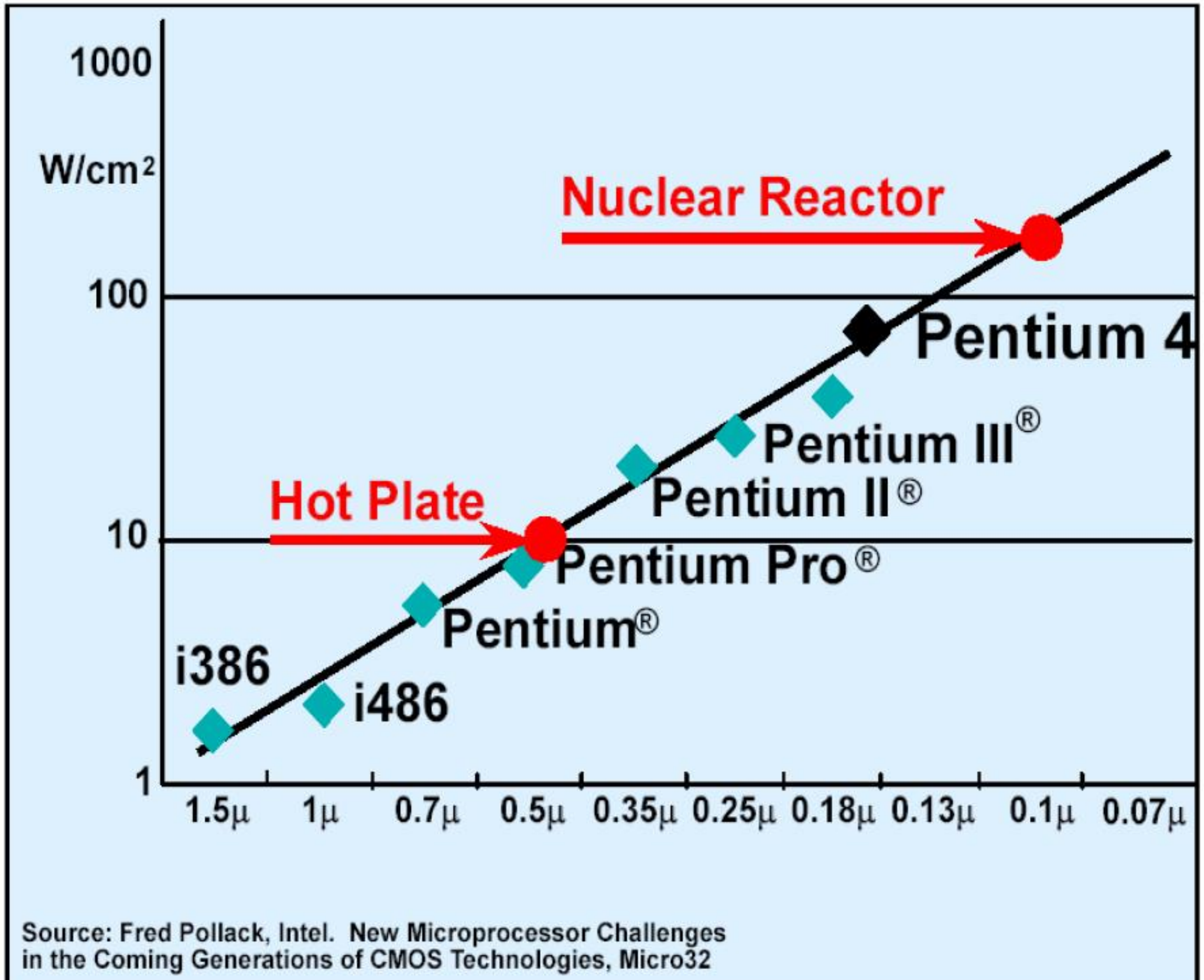
- **There is a growing gap between core and memory performance:**
  - → memory, since 1980: 1.07x per year improvement in latency
  - → single core: since 1980: 1.25x per year until 1986, 1.52x p. y. until 2000, 1.20x per year until 2005, then no change on a *per-core* basis



→ Source: John L. Hennessy, Stanford University, and David A. Patterson, University of California, September 25, 2012

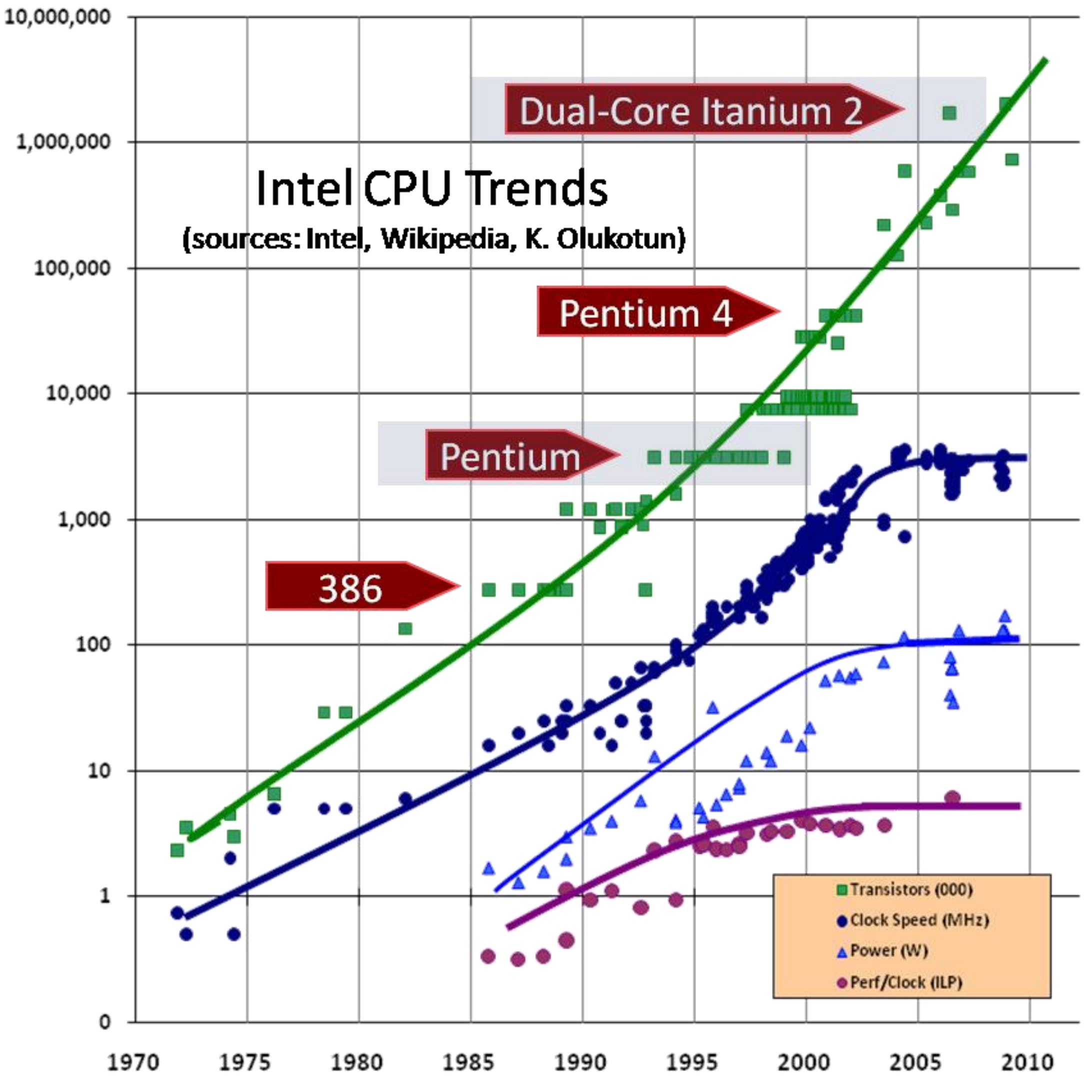
Why is there no 4.0 GHz x86 CPU?

- **Because that beast would get too hot!**



Fast clock cycles make processor chips more expensive, hotter and more power consuming.

Moore‘s Law still holds!



The number of transistors on a chip is still doubling every 24 months …

… but the clock speed is no longer increasing that fast!

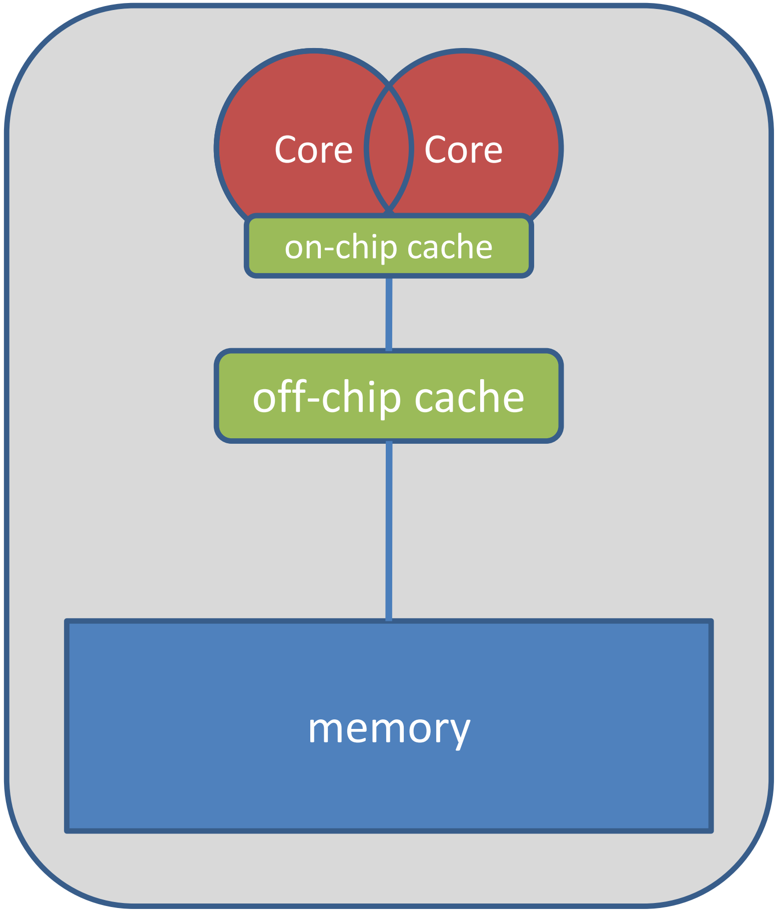
Instead, we will see many more cores per chip!

Source: Herb Sutter

www.gotw.ca/publications/concurrency-ddj.htm

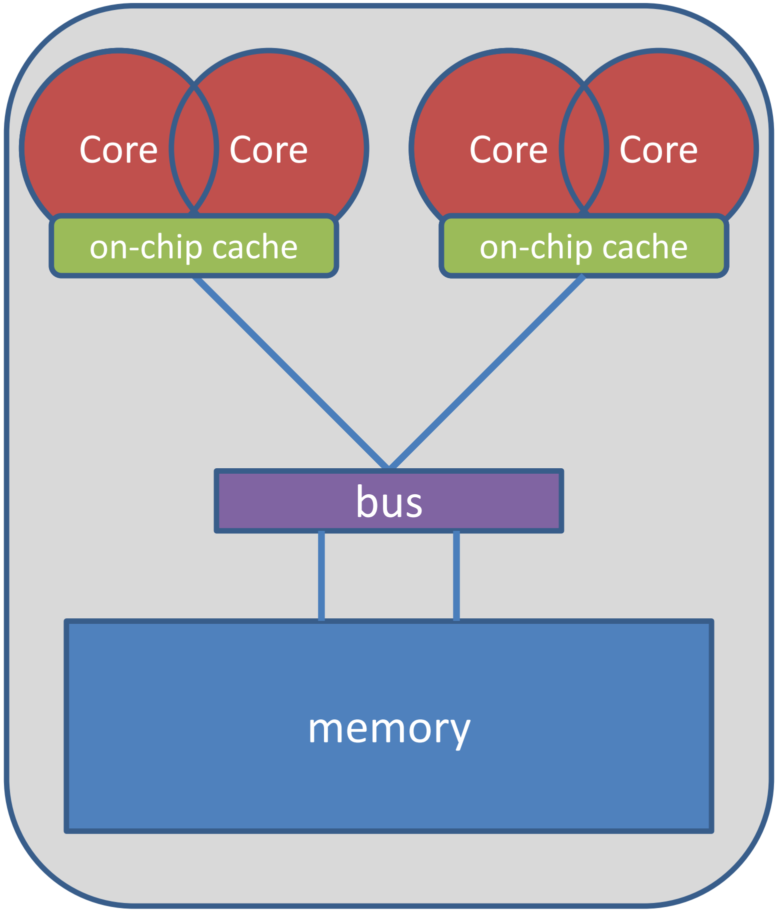
Dual-Core Processor System

- **Since 2005/2006 Intel and AMD are producing dual-core processors for the mass market!**
- **In 2006/2007 Intel and AMD introduced quad-core processors.**
- **→ Any recently bought PC or laptop is a multi-core system already!**



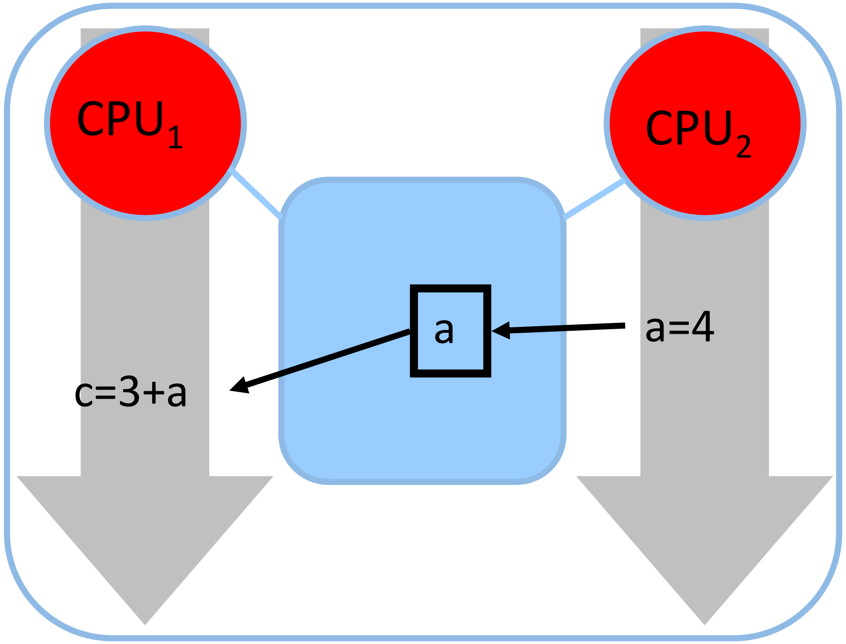
Example for a SMP system

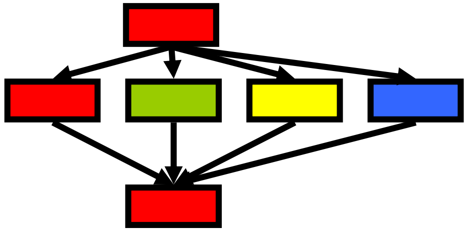
- **Dual-socket Intel Woodcrest (dual-core) system**
  - → Two cores per chip, 3.0 GHz
  - → Each chip has 4 MB of L2 cache on-chip, shared by both cores
  - → No off-chip cache
  - → Bus: Frontsidebus
- **SMP: Symmetric Multi Processor**
  - → Memory access time is uniform on all cores
  - → Limited scalabilty



Shared Memory Parallelization

- **Memory can be accessed by several threads running on different cores in a multi-socket multi-core system:**





Look for tasks that can be executed simultaneously (task parallelism)

OpenMP Overview

&

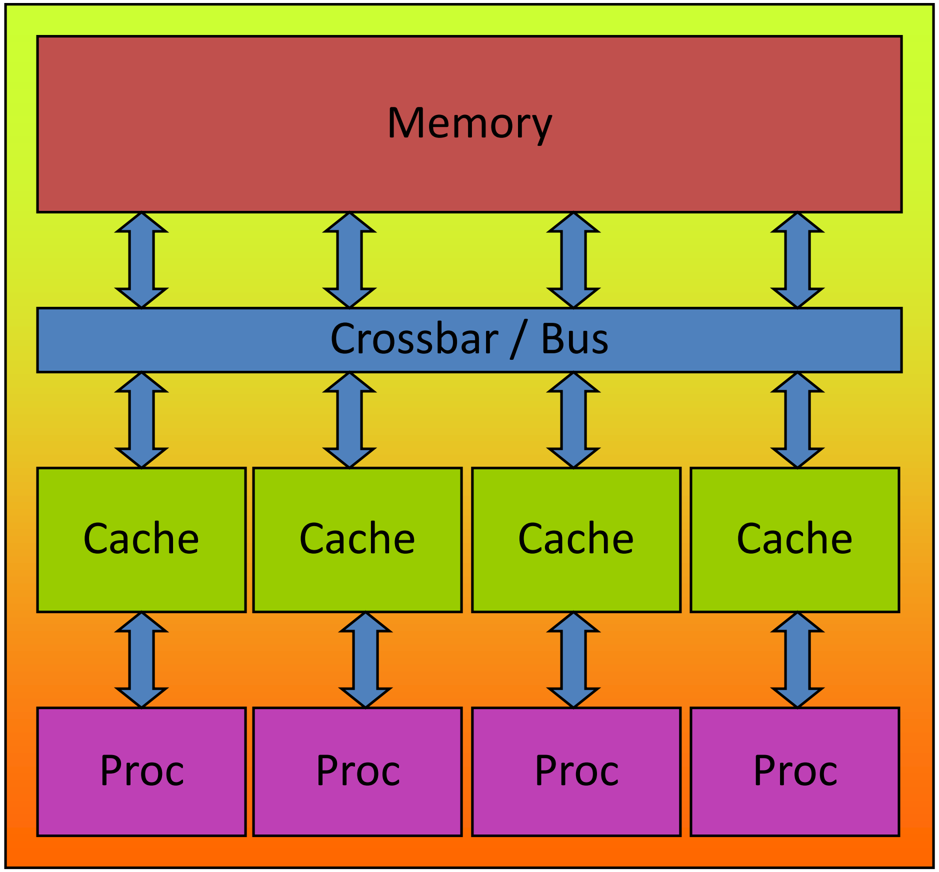
Parallel Region

&

Basic Worksharing

OpenMP's machine model

- **OpenMP: Shared-Memory Parallel Programming Model.**



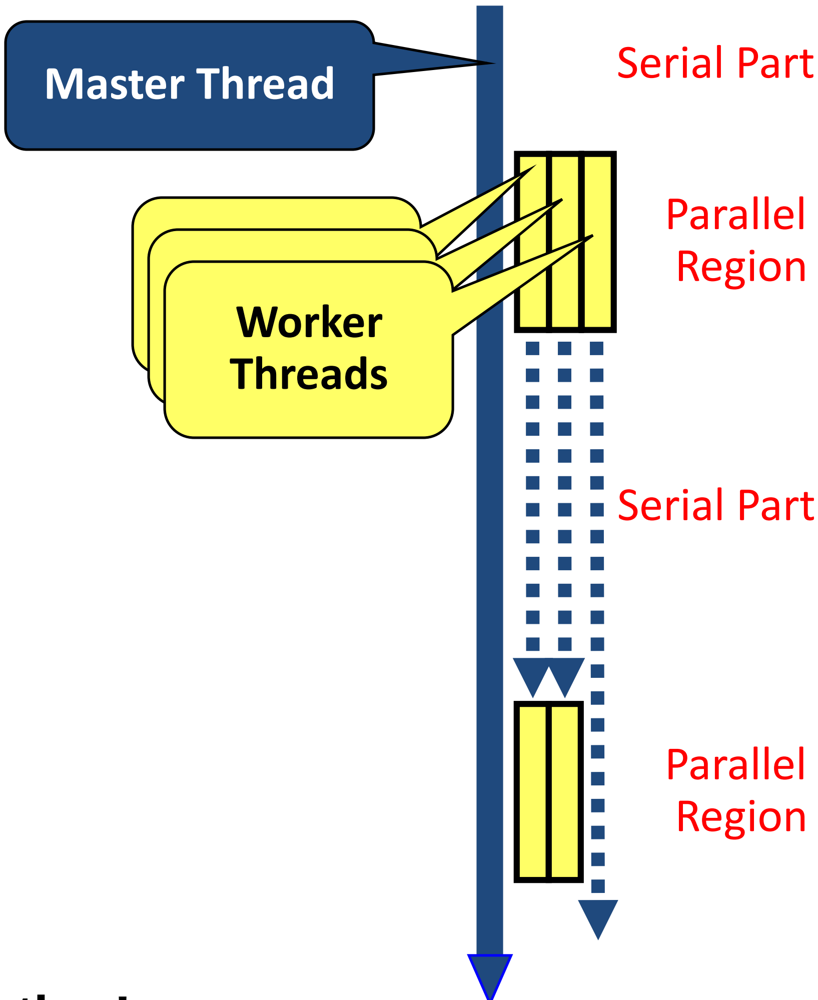
All processors/cores access a shared main memory.

Real architectures are more complex, as we will see later / as we have seen.

Parallelization in OpenMP employs multiple threads.

OpenMP Execution Model

- **OpenMP programs start with just one thread: The *Master*.**
- ***Worker* threads are spawned at *Parallel Regions*, together with the Master they form the *Team* of threads.**
- **In between Parallel Regions the Worker threads are put to sleep. The OpenMP *Runtime* takes care of all thread management work.**
- **Concept: *Fork-Join*.**
- **Allows for an incremental parallelization!**



Parallel Region and Structured Blocks

- **The parallelism has to be expressed explicitly.**

C/C++

```
#pragma omp parallel
{
    ...
    structured block
    ...
}
```

Fortran

```
!$omp parallel
   ...
   structured block
   ...
$!omp end parallel
```

- ***Structured Block***
  - → Exactly one entry point at the top
  - → Exactly one exit point at the bottom
  - → Branching in or out is not allowed
  - → Terminating the program is allowed (abort / exit)

- ***Specification of number of threads:***
  - Environment variable:
    `OMP_NUM_THREADS=…`
  - Or: Via `num_threads` clause:
    add `num_threads(num)` to the parallel construct

Hello OpenMP World

Hello orphaned OpenMP World

Starting OpenMP Programs on Linux

- **From within a shell, global setting of the number of threads:**

```
export OMP_NUM_THREADS=4
./program
```

- **From within a shell, one-time setting of the number of threads:**

```
OMP_NUM_THREADS=4   ./program
```

- **Intel Compiler on Linux: ask the runtime for more information:**

```
export KMP_AFFINITY=verbose
export OMP_NUM_THREADS=4
./program
```

For Construct

For Worksharing

- **If only the *parallel* construct is used, each thread executes the Structured Block.**
- **Program Speedup: *Worksharing***
- **OpenMP‘s most common Worksharing construct: *for***

C/C++

```
int i;
#pragma omp parallel for
for (i = 0; i < 100; i++)
{
   a[i] = b[i] + c[i];
}
```

Fortran

```
INTEGER :: i
!$omp parallel do
DO i = 0, 99
   a[i] = b[i] + c[i];
END DO
```

→ Distribution of loop iterations over all threads in a Team.

→ Scheduling of the distribution can be influenced.

- **Loops often account for most of a program‘s runtime!**

Worksharing illustrated

Pseudo-Code
Here: 4 Threads

Serial

```
do i = 0, 99
    a(i) = b(i) + c(i)
end do
```

Thread 1

```
do i = 0, 24
    a(i) = b(i) + c(i)
end do
```

Thread 2

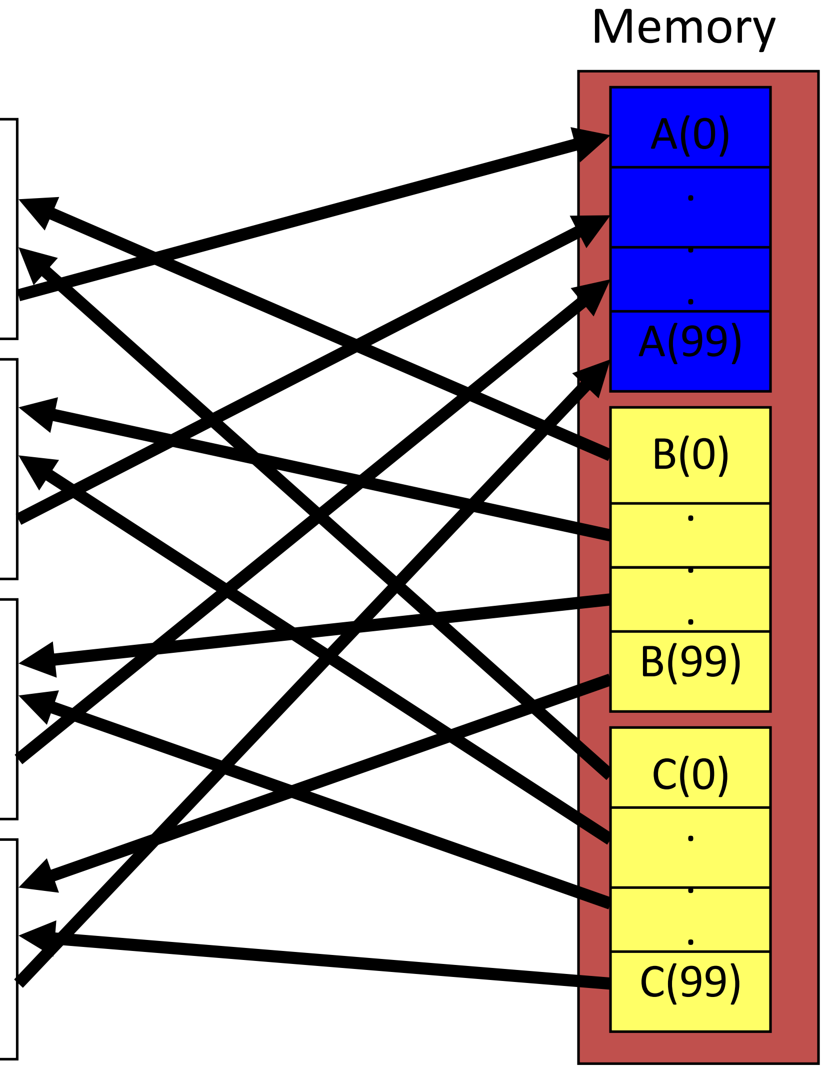
```
do i = 25, 49
    a(i) = b(i) + c(i)
end do
```

Thread 3

```
do i = 50, 74
    a(i) = b(i) + c(i)
end do
```

Thread 4

```
do i = 75, 99
    a(i) = b(i) + c(i)
end do
```



Vector Addition

Synchronization Overview

- **Can all loops be parallelized with `for`-constructs? No!**
  - → Simple test: If the results differ when the code is executed backwards, the loop iterations are not independent. BUT: This test alone is not sufficient:

C/C++

```
int i;
#pragma omp parallel for
for (i = 0; i < 100; i++)
{
   s = s + a[i];
}
```

- ***Data Race*: If between two synchronization points at least one thread writes to a memory location from which at least one other thread reads, the result is not deterministic (race condition).**

Synchronization: Critical Region

- A *Critical Region* is executed by all threads, but by only one thread simultaneously (*Mutual Exclusion*).

C/C++

```
#pragma omp critical (name)
{
    ... structured block ...
}
```

- Do you think this solution scales well?

C/C++

```
int i;
#pragma omp parallel for
for (i = 0; i < 100; i++)
{

#pragma omp critical
    {   s = s + a[i];   }
}
```

It‘s your turn: Make It Scale!

```
#pragma omp parallel
{

#pragma omp for
   for (i = 0; i < 99; i++)
   {

         s  = s   + a[i];

   }

} // end parallel
```

```
do i = 0, 99
   s = s + a(i)
end do
```

```
do i = 0, 24
   s = s + a(i)
end do
```

```
do i = 25, 49
   s = s + a(i)
end do
```

```
do i = 50, 74
   s = s + a(i)
end do
```

```
do i = 75, 99
   s = s + a(i)
end do
```

The Reduction Clause

- **In a *reduction*-operation the operator is applied to all variables in the list. The variables have to be *shared*.**
  - → `reduction(operator:list)`
  - → The result is provided in the associated reduction variable

C/C++

```
#pragma omp parallel for reduction(+:s)
for(i = 0; i < 99; i++)
{
   s = s + a[i];
}
```

  - → Possible reduction operators with initialization value:

`+ (0), * (1), - (0),`

`& (~0), | (0), && (1), || (0),`

`^ (0), min (least number), max (largest number)`

VTune: Detecting Hotspots

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

Stream

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

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

Measures Memory bandwidth for:

y=x (copy)

y=s*x (scale)

y=x+z (add)

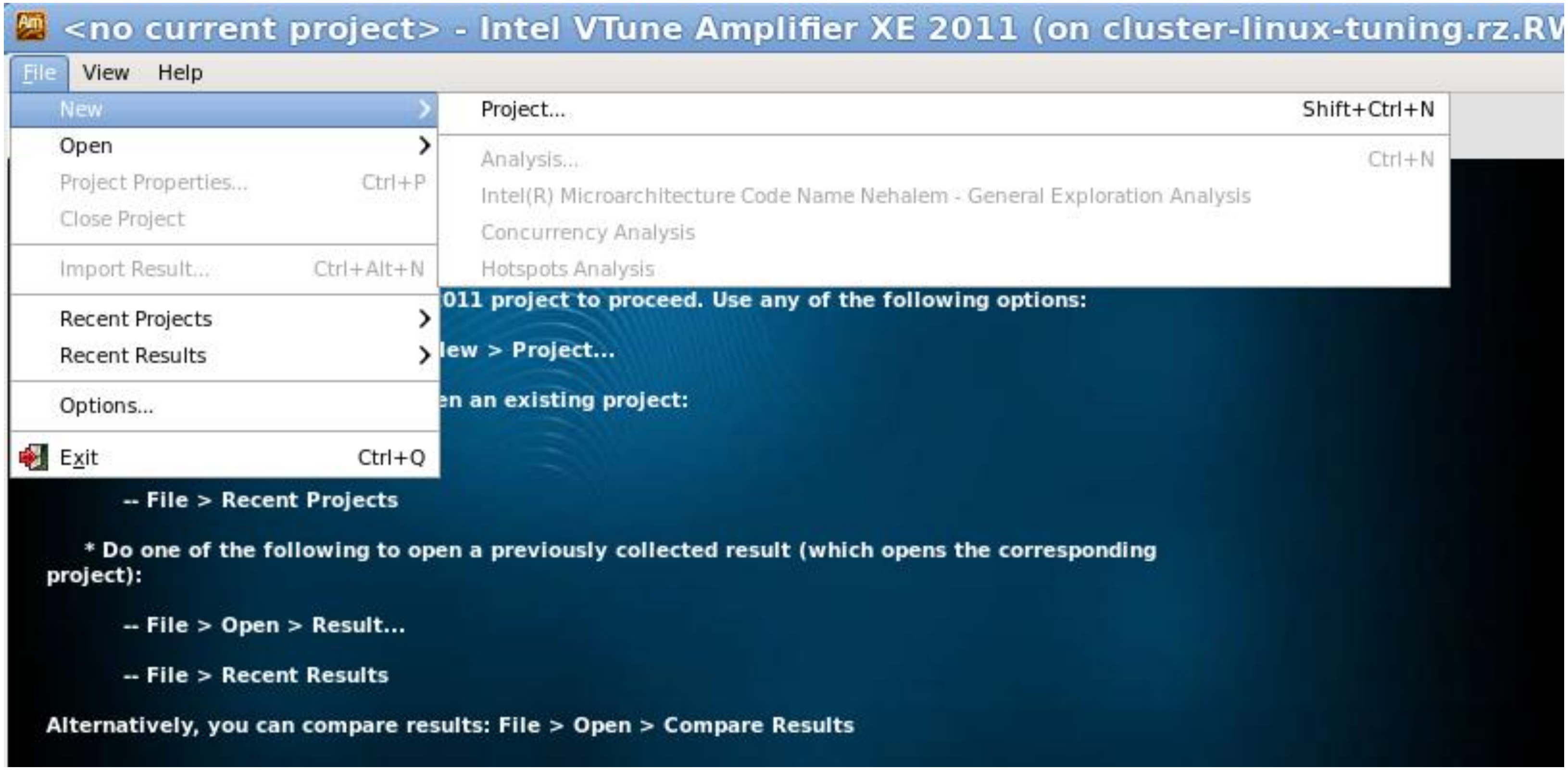
y=x+s*z (triad)

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

```
-------------------------------------------------------------
Function      Rate (MB/s)   Avg time     Min time     Max time
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
```

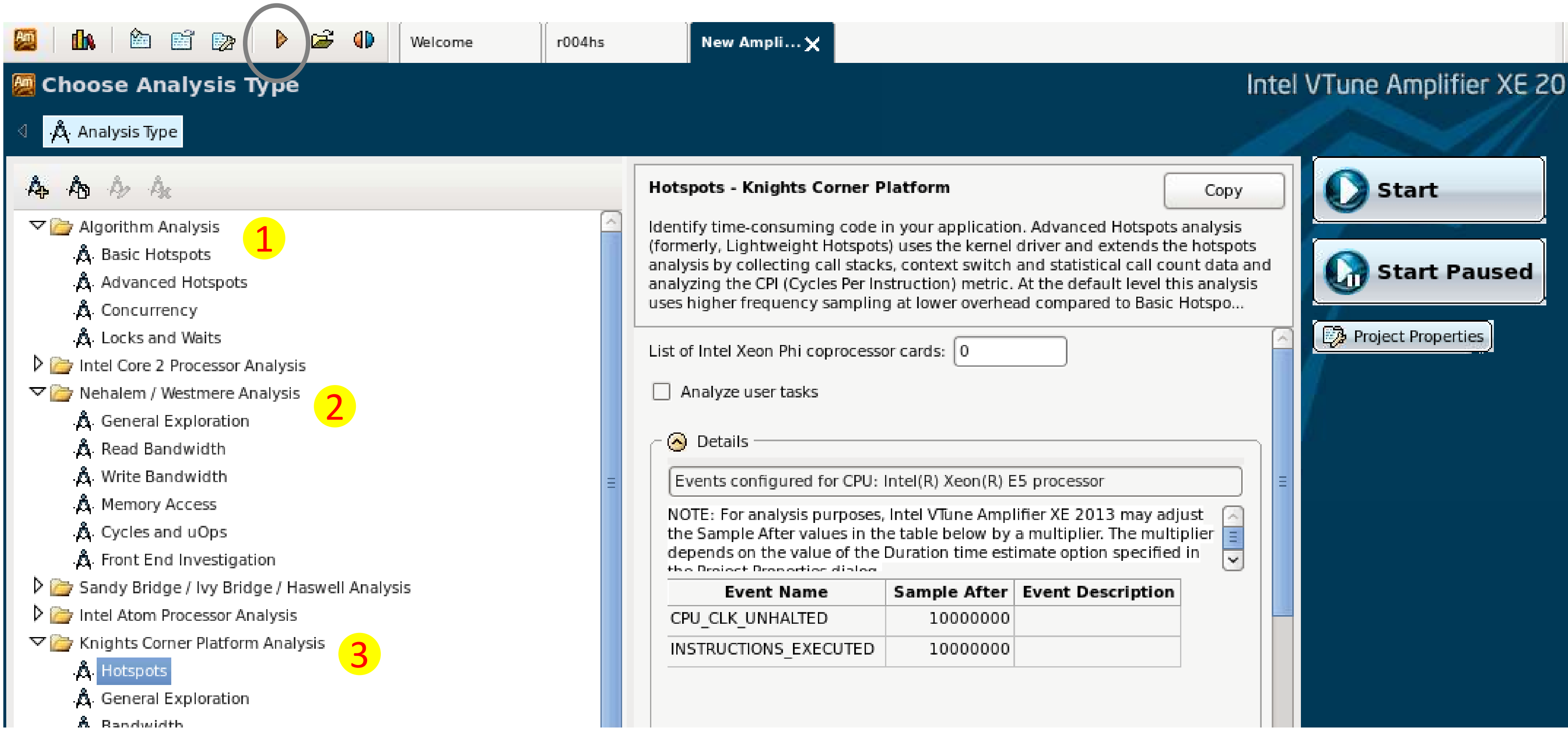
Amplifier XE – Create Project

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

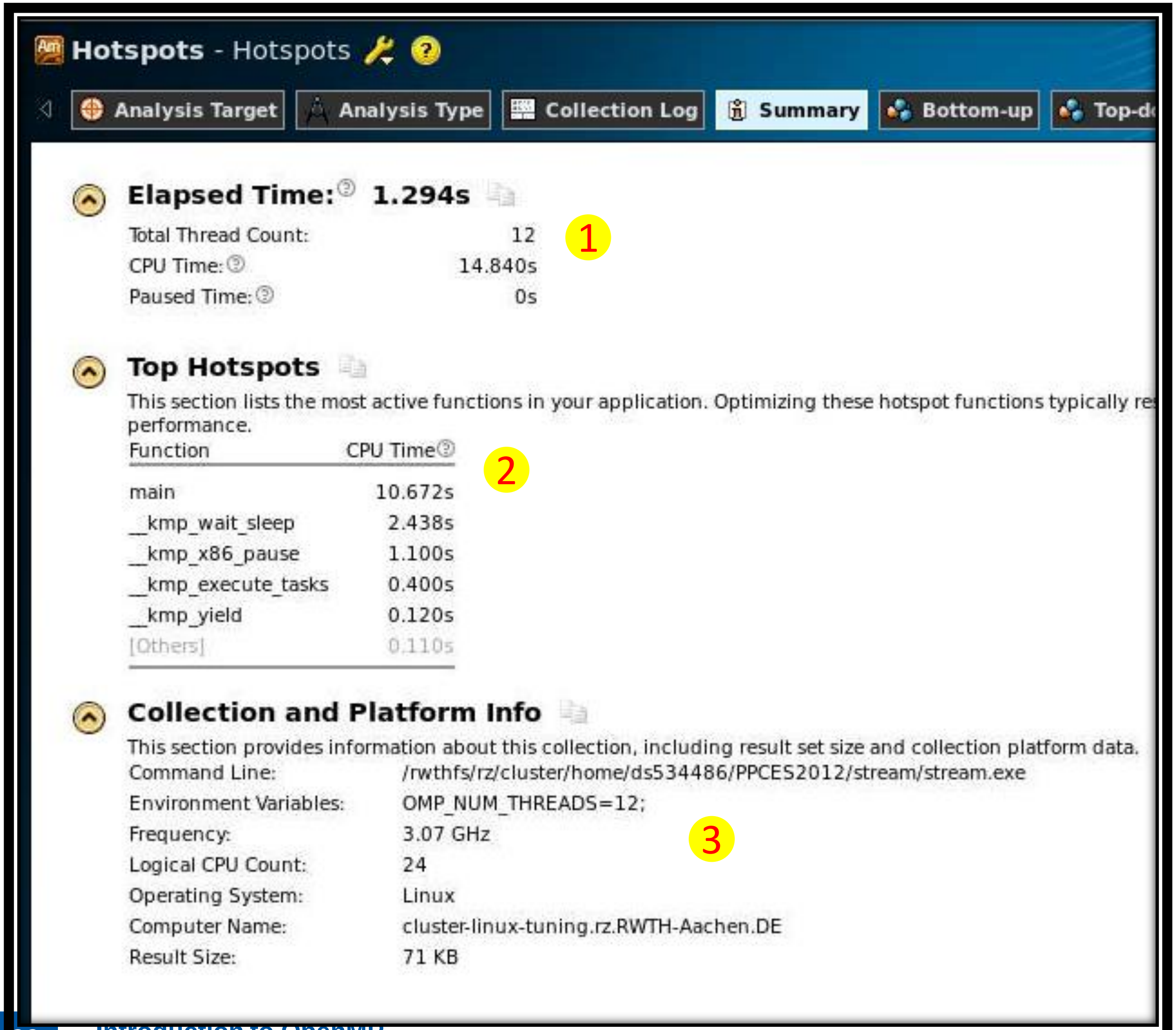


Amplifier XE – Measurement Runs

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



Amplifier XE – Hotspot Analysis

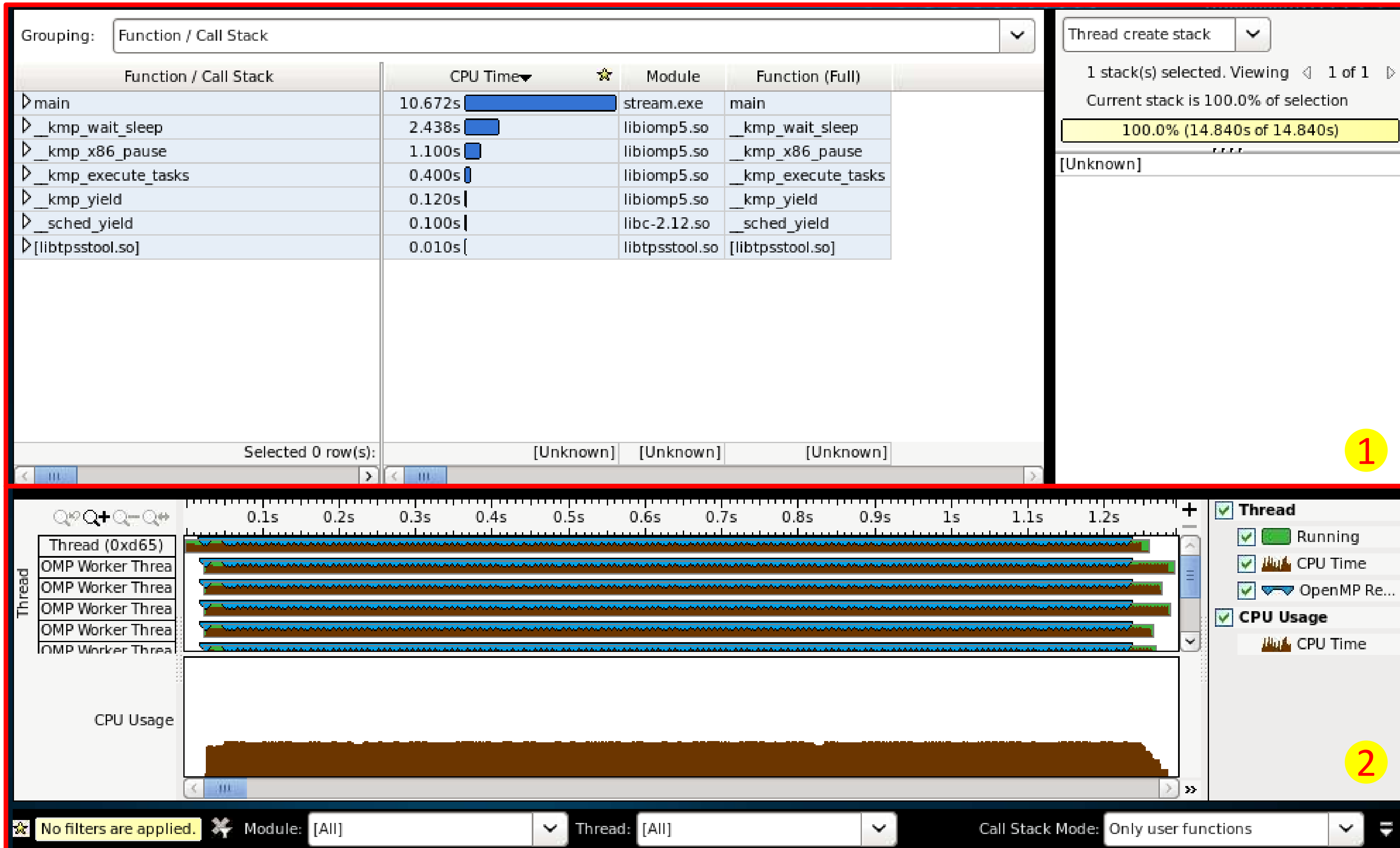


Summary:

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

Amplifier XE – Hotspot Analysis

1. Function Summary
2. Timeline View



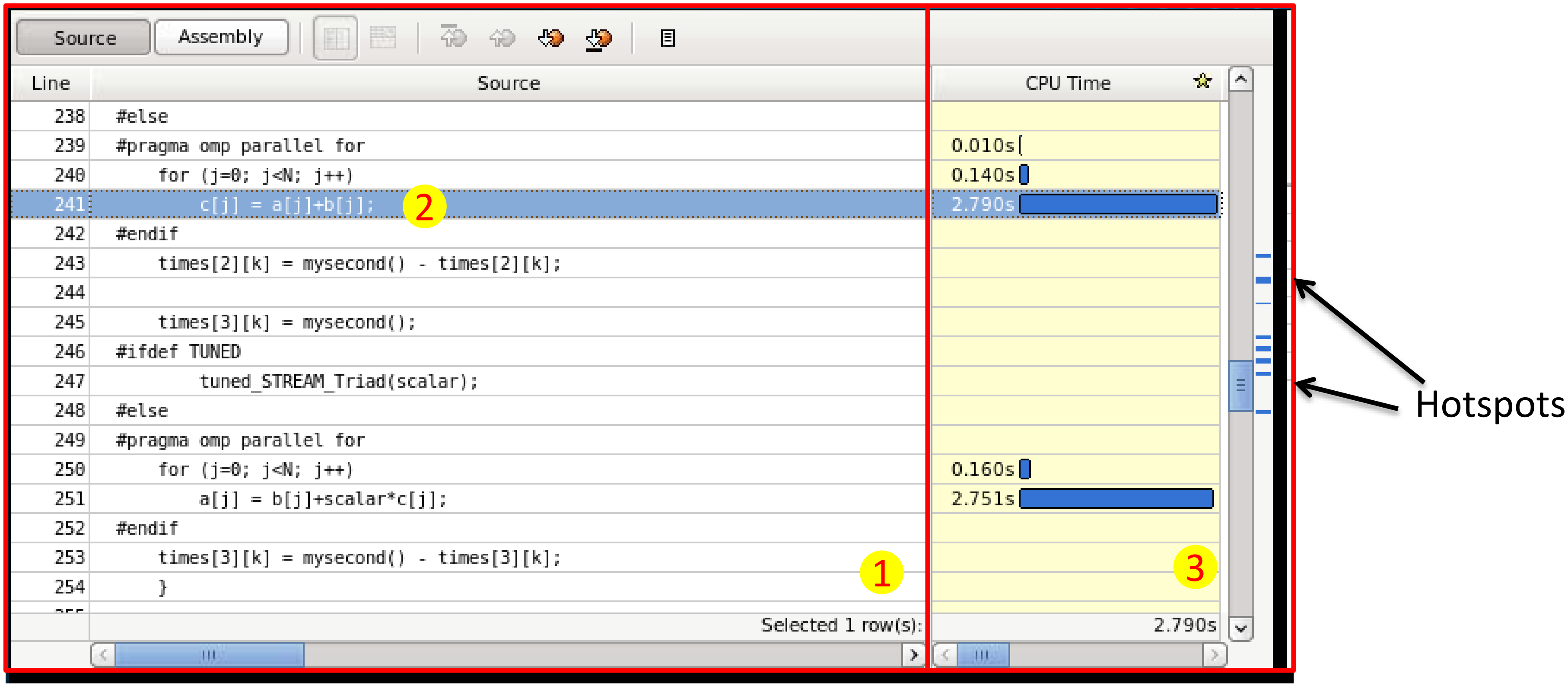
Amplifier XE – Hotspot Analysis

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



Data Scoping

Scoping Rules

- **Managing the Data Environment is the challenge of OpenMP.**

- ***Scoping* in OpenMP: Dividing variables in *shared* and *private*:**
  - → *private*-list and *shared*-list on Parallel Region
  - → *private*-list and *shared*-list on Worksharing constructs
  - → General default is *shared* for Parallel Region, *firstprivate* for Tasks.
  - → Loop control variables on *for*-constructs are *private*
  - → Non-static variables local to Parallel Regions are *private*
  - → *private*: A new uninitialized instance is created for each thread
    - → *firstprivate*: Initialization with Master‘s value
    - → *lastprivate*: Value of last loop iteration is written back to Master
  - → Static variables are *shared*

Privatization of Global/Static Variables

- **Global / static variables can be privatized with the *threadprivate* directive**
  - → One instance is created for each thread
    - → Before the first parallel region is encountered
    - → Instance exists until the program ends
    - → Does not work (well) with nested Parallel Region
  - → Based on thread-local storage (TLS)
    - → TlsAlloc (Win32-Threads), pthread_key_create (Posix-Threads), keyword `__thread` (GNU extension)

Really: try to avoid the use of threadprivate and static variables!

C/C++

```
static int i;
#pragma omp threadprivate(i)
```

Fortran

```
SAVE INTEGER :: i
!$omp threadprivate(i)
```

The Barrier Construct

- **OpenMP `barrier` (implicit or explicit)**
  - → Threads wait until all threads of the current *Team* have reached the barrier

C/C++

```
#pragma omp barrier
```

Exercises 1, 2 and 4

Inspector: Detecting Data Races

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*

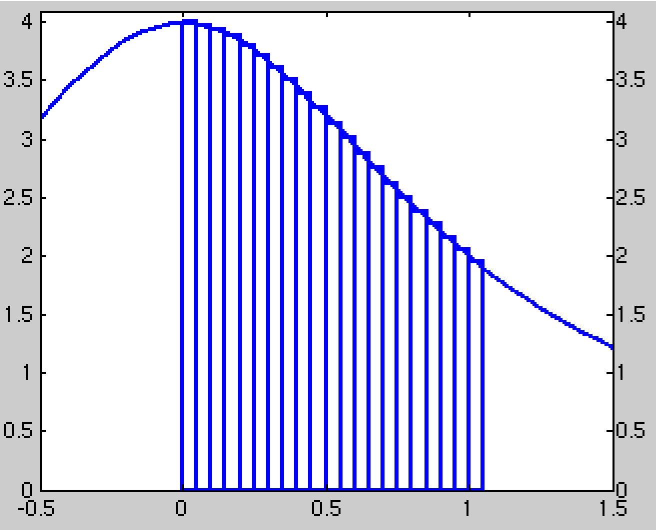
Intel Inspector XE

- **Detection of**
  - → Memory Errors
  - → Dead Locks
  - → Data Races
- **Support for**
  - → Linux (32bit and 64bit) and Windows (32bit and 64bit)
  - → WIN32-Threads, Posix-Threads, Intel Threading Building Blocks and OpenMP
- **New Features (compared to Intel Thread Checker)**
  - → Binary Instrumentation gives full functionality
  - → Independent stand-alone GUI for Windows and Linux
  - → memory error detection
  - → static security analysis (in combination with the Intel 12.X compiler)

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;
}
```

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



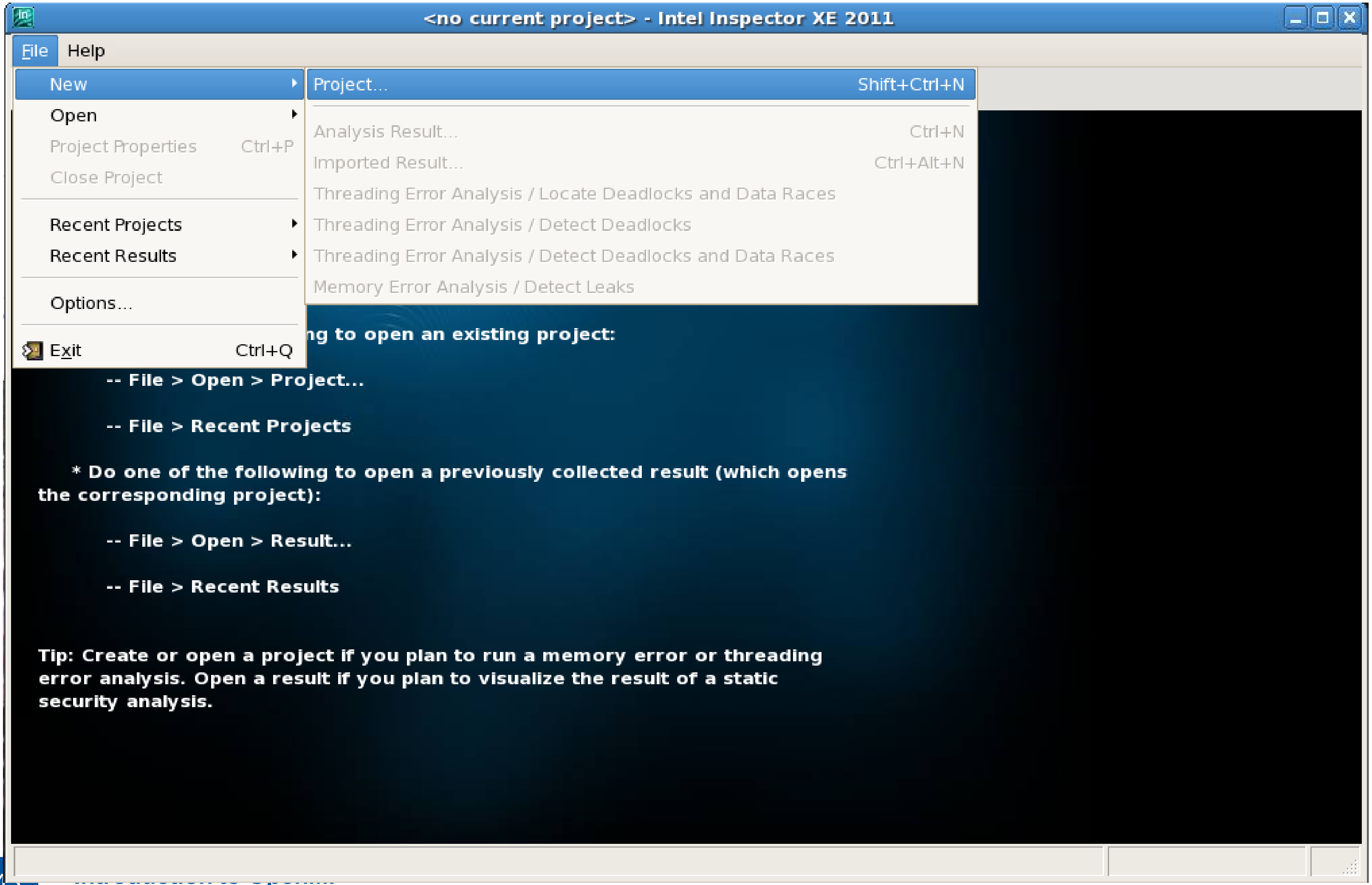
PI Example Code

```
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?

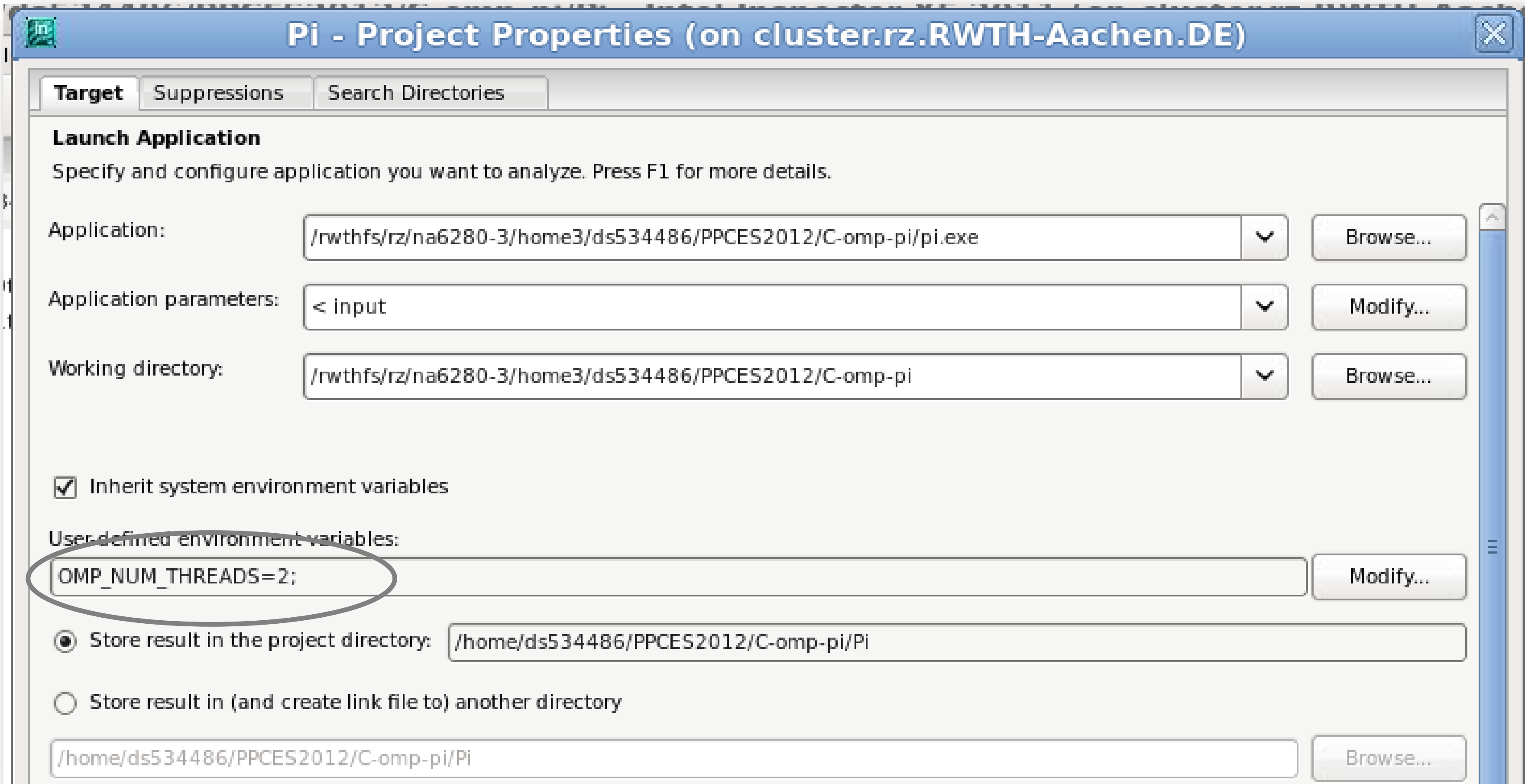
Inspector XE – Create Project

$ module load intelixe ; inspxe-gui



Inspector XE – Create Project

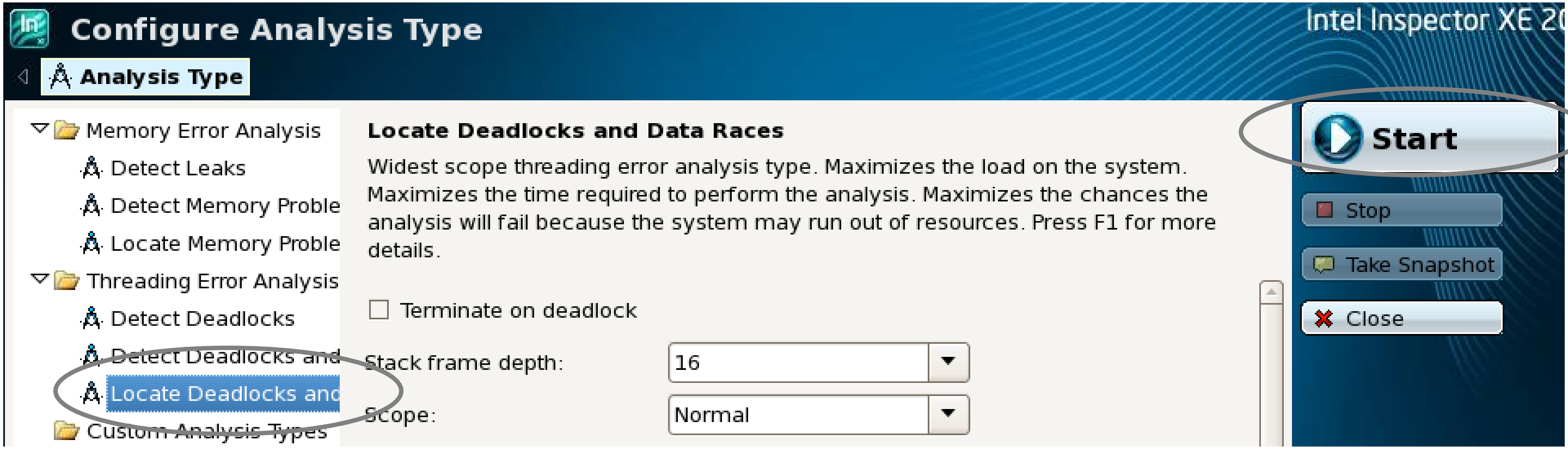
- ensure that multiple threads are used
- choose a real small dataset, execution time can grow 10X – 1000X



Threading Error Analysis Modes

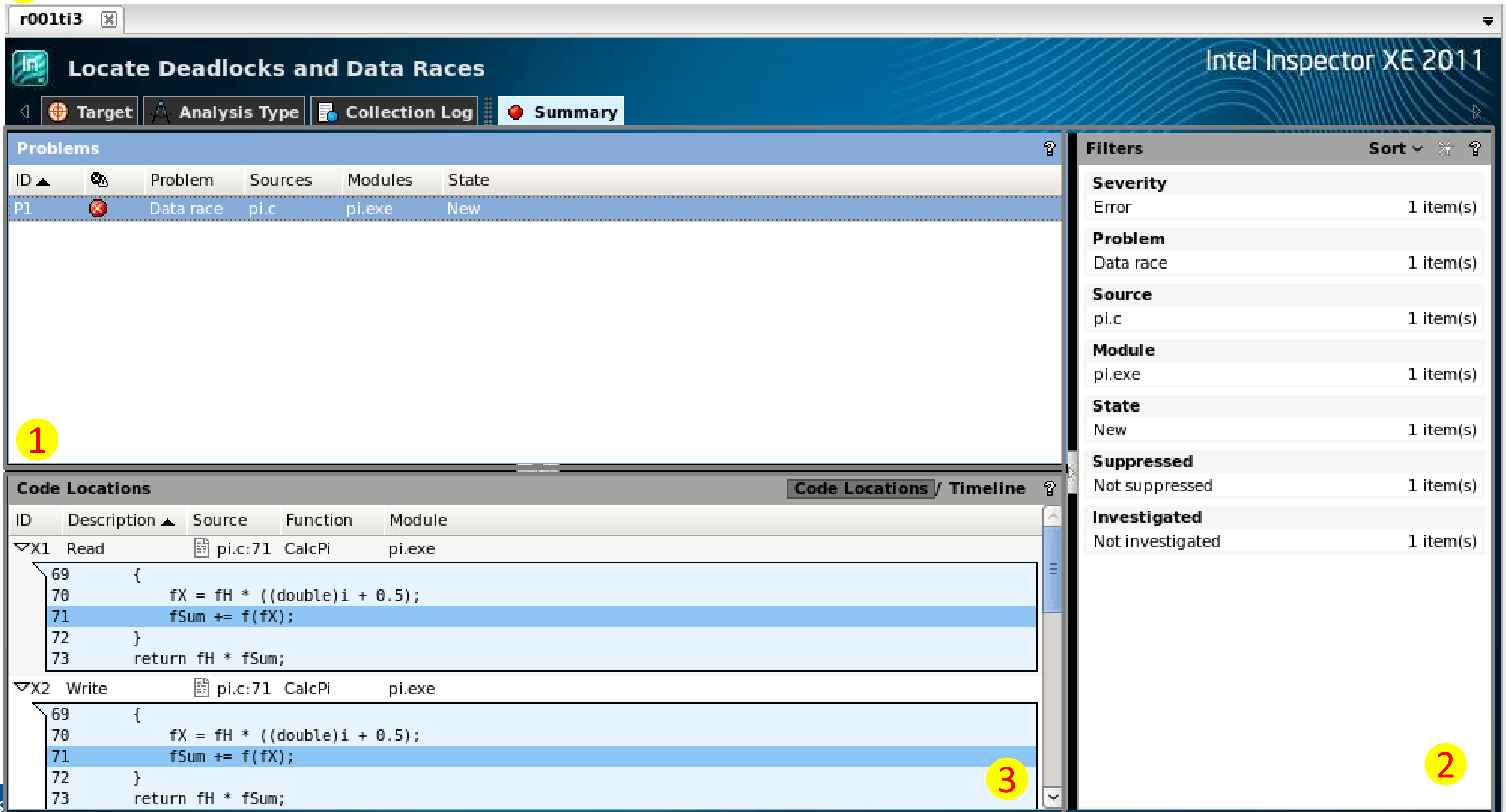
1. Detect Deadlocks
2. Detect Deadlocks and Data Races
3. Locate Deadlocks and Data Races

more details,
more overhead



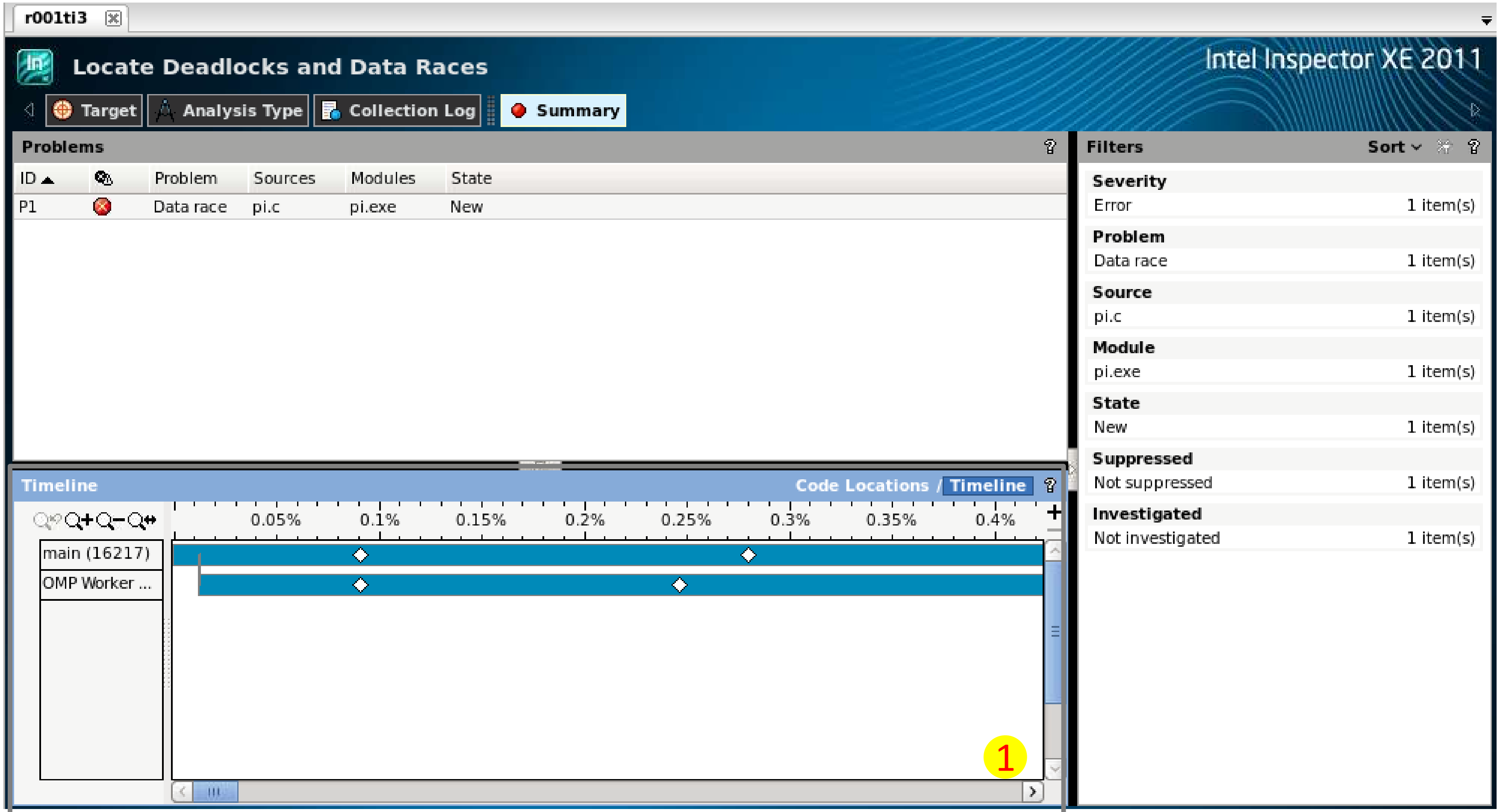
Inspector XE – Results

1 detected problems
2 filters
3 code location



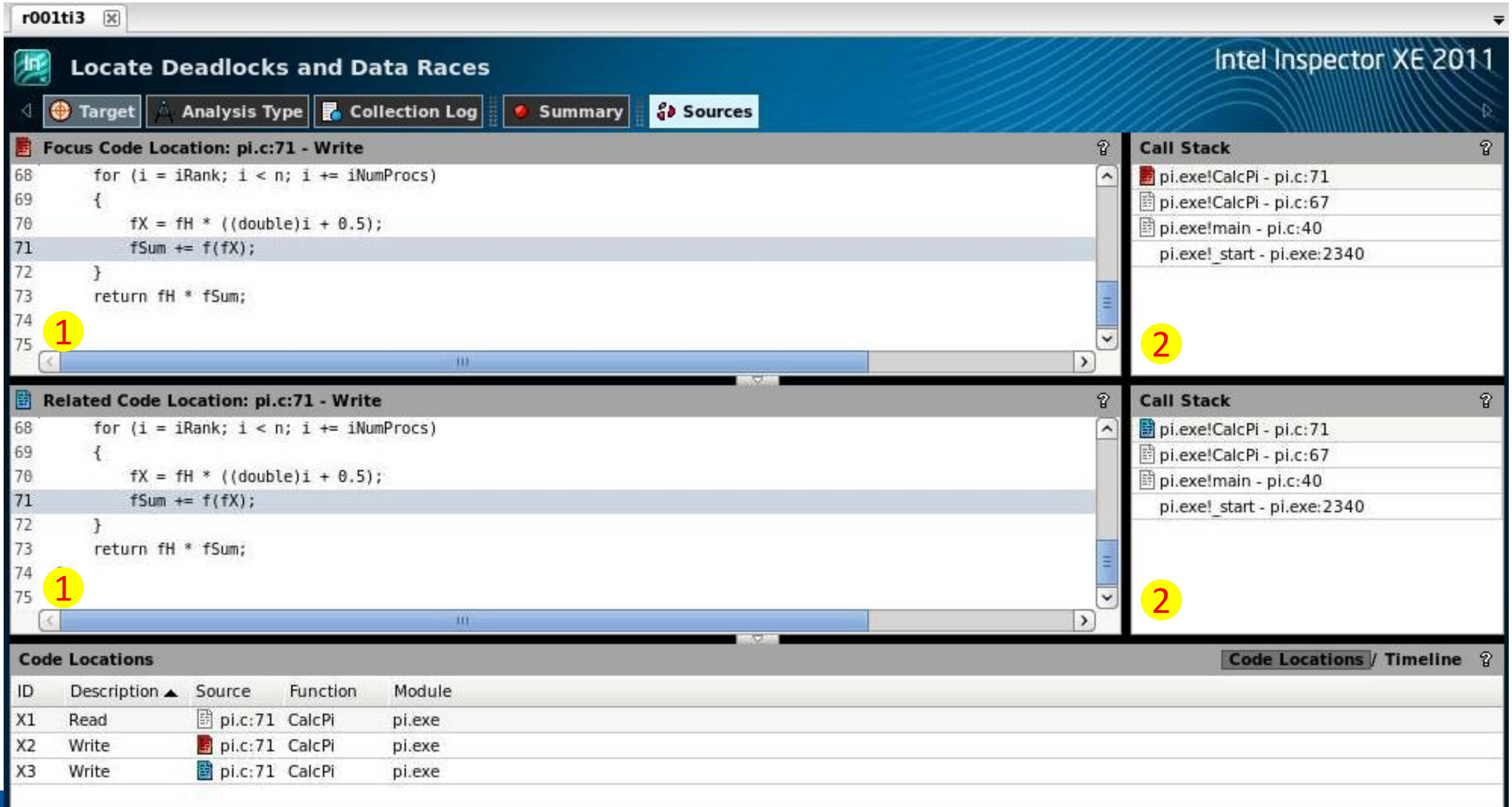
Inspector XE – Results

1 Timeline view



Inspector XE – Results

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

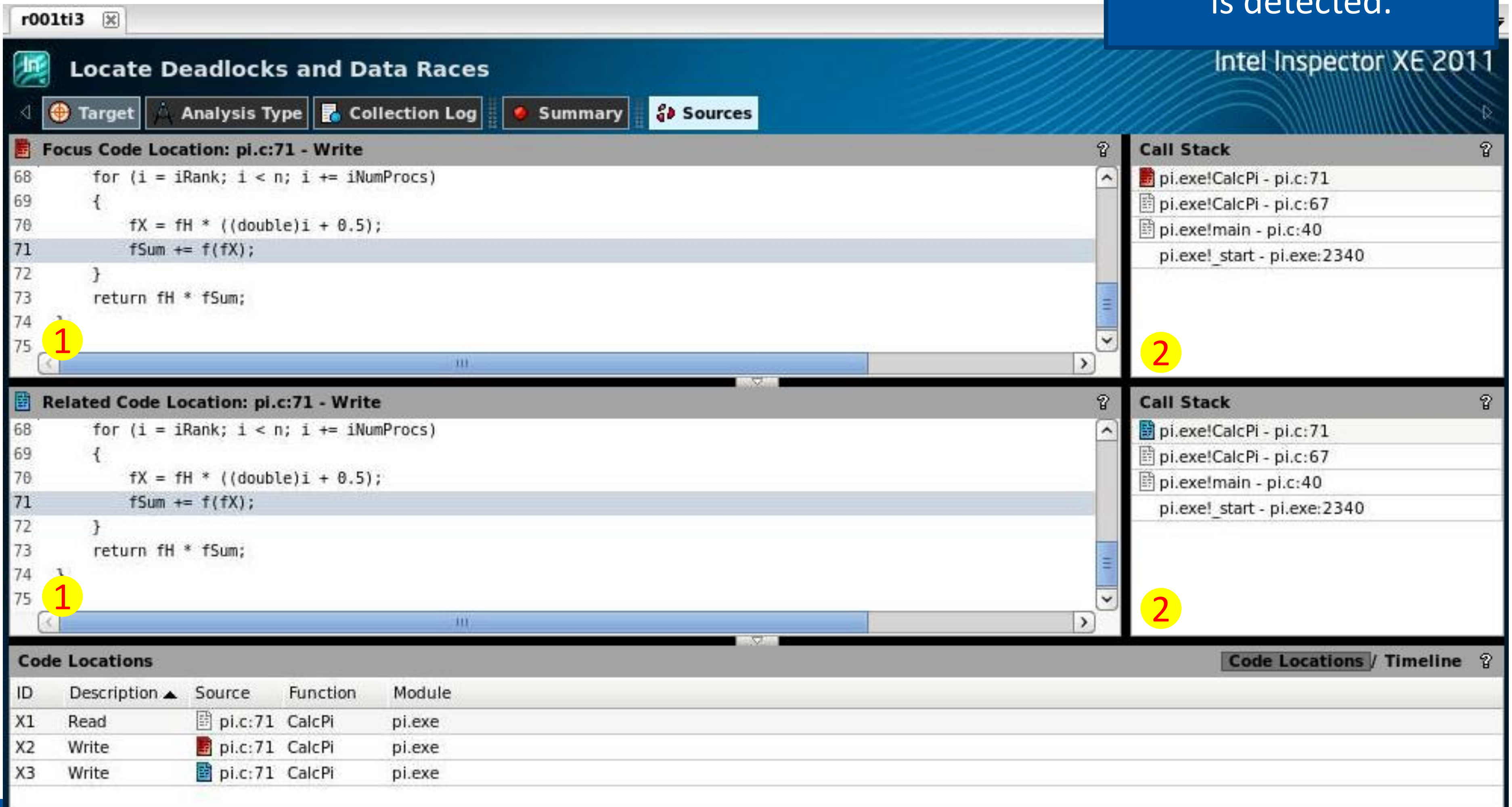


Inspector XE – Results

1 Source Code producing the issue – double click opens an editor

2 Corresponding Call Stack

The missing reduction is detected.



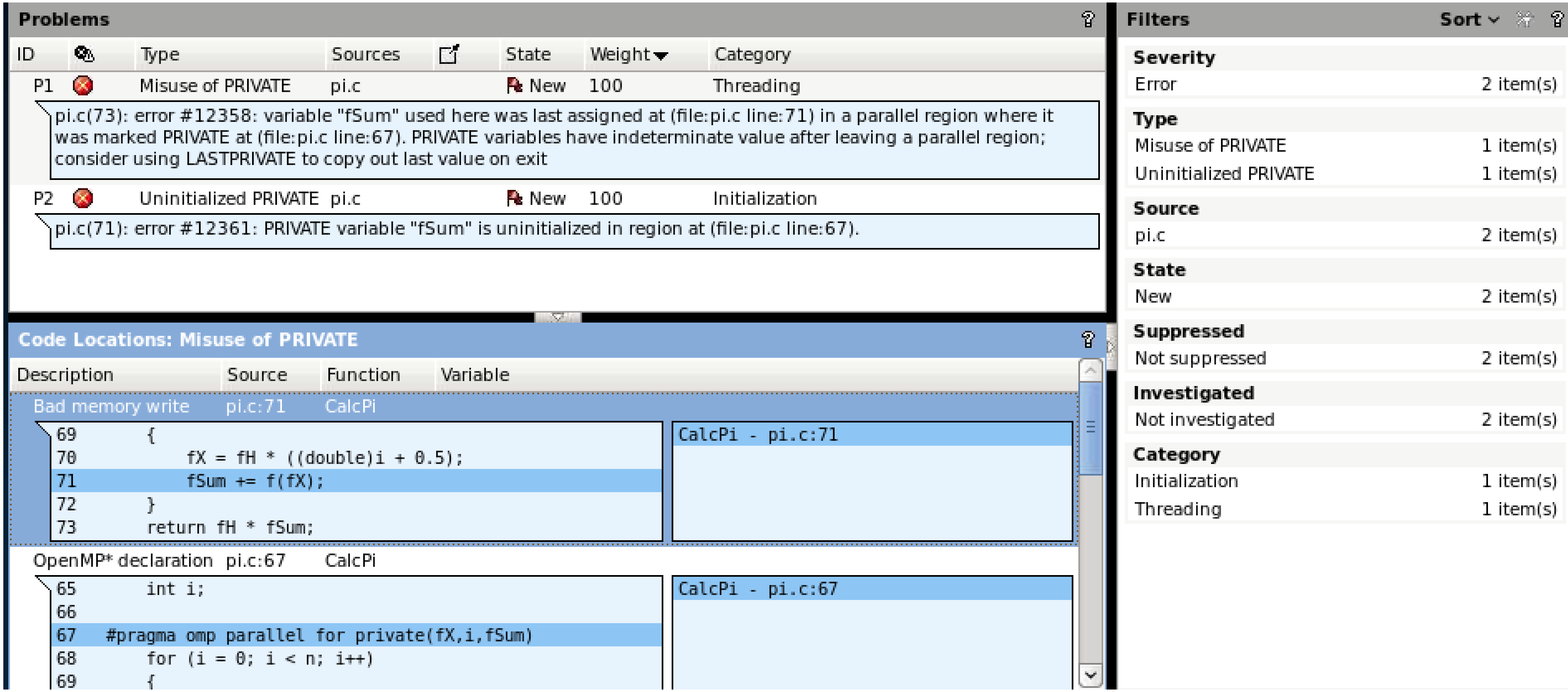
PI Example Code

```
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,fSum)
   for (i = 0; i < n; i++)
   {
      fX = fH * ((double)i + 0.5);
      fSum += f(fX);
   }
   return fH * fSum;
}
```

What if we just made the variable private?

Inspector XE – Static Security Analysis

- At runtime no Error is detected!
- Compiling with the argument “-diag-enable sc-full” delivers:



- At compile-time this error can be found!