Tools for OpenMP Programming

OpenMP Tools

- Correctness Tools
  - → ThreadSanitizer
  - → Intel Inspector XE (or whatever the current name is)

- Performance Analysis
  - → Performance Analysis basics
  - → Overview on available tools

Data Race

- 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
  - → In many cases *private* clauses, *barriers* or *critical regions* are missing
- Data races are hard to find using a traditional debugger

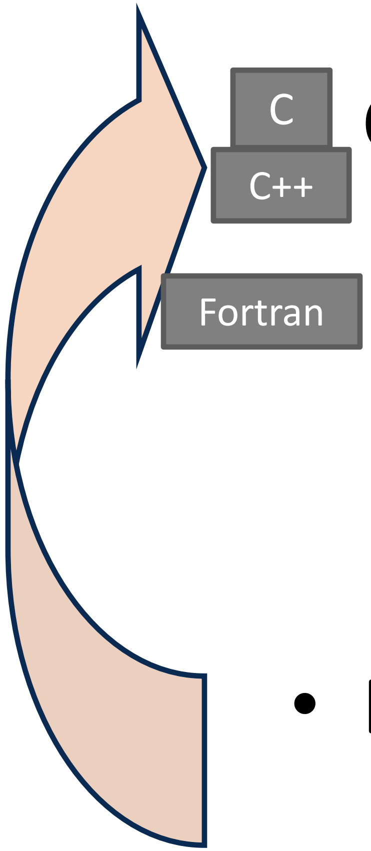
ThreadSanitizer: Overview

- Correctness checking for threaded applications
- Integrated in clang and gcc compiler
- Low runtime overhead: 2x – 15x
- Used to find data races in browsers like Chrome and Firefox

ThreadSanitizer: Usage

```
module load clang
```

Module in Aachen.
https://pruners.github.io



Compile the program with clang compiler:

```
clang -fsanitize=thread -fopenmp -g myprog.c -o myprog
clang++ -fsanitize=thread -fopenmp -g myprog.cpp
          -o myprog
gfortran -fsanitize=thread -fopenmp -g myprog.f -c
clang -fsanitize=thread -fopenmp -lgfortran myprog.o
          -o myprog
```

- Execute:

```
OMP_NUM_THREADS=4 ./myprog
```

- Understand and correct the detected threading errors

ThreadSanitizer: Example

```c
1  #include <stdio.h>
2
3  int main(int argc, char **argv) {
4      int a = 0;
5      #pragma omp parallel
6      {
7          if (a < 100) {
8              #pragma omp critical
9              a++;
10         }
11     }
12 }
```

WARNING: ThreadSanitizer: data race

Read of size 4 at 0x7fffffffdcdc by thread T2:

```
  #0 .omp_outlined. race.c:7
(race+0x0000004a6dce)
  #1 __kmp_invoke_microtask <null>
(libomp_tsan.so)
```

Previous write of size 4 at 0x7fffffffdcdc by main thread:

```
  #0 .omp_outlined. race.c:9
(race+0x0000004a6e2c)
  #1 __kmp_invoke_microtask <null>
(libomp_tsan.so)
```

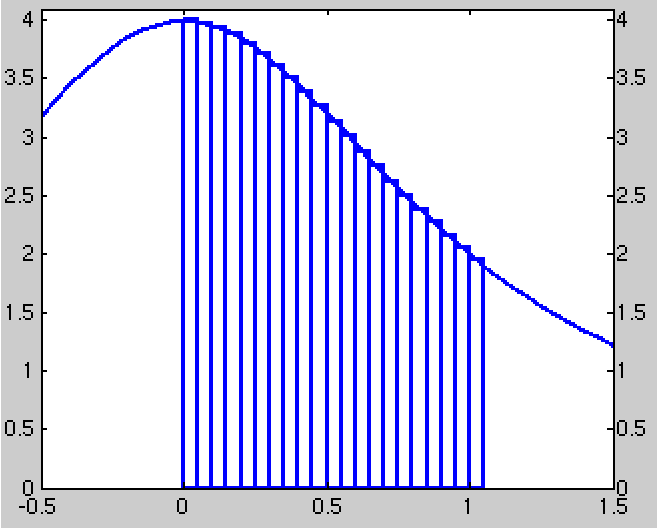
Intel Inspector XE

- Detection of
  - → Memory Errors
  - → Deadlocks
  - → Data Races
- Support for
  - → WIN32-Threads, Posix-Threads, Intel Threading Building Blocks and OpenMP
- Features
  - → Binary instrumentation gives full functionality
  - → Independent stand-alone GUI for Windows and Linux

PI example / 1

```
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;
}
```

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



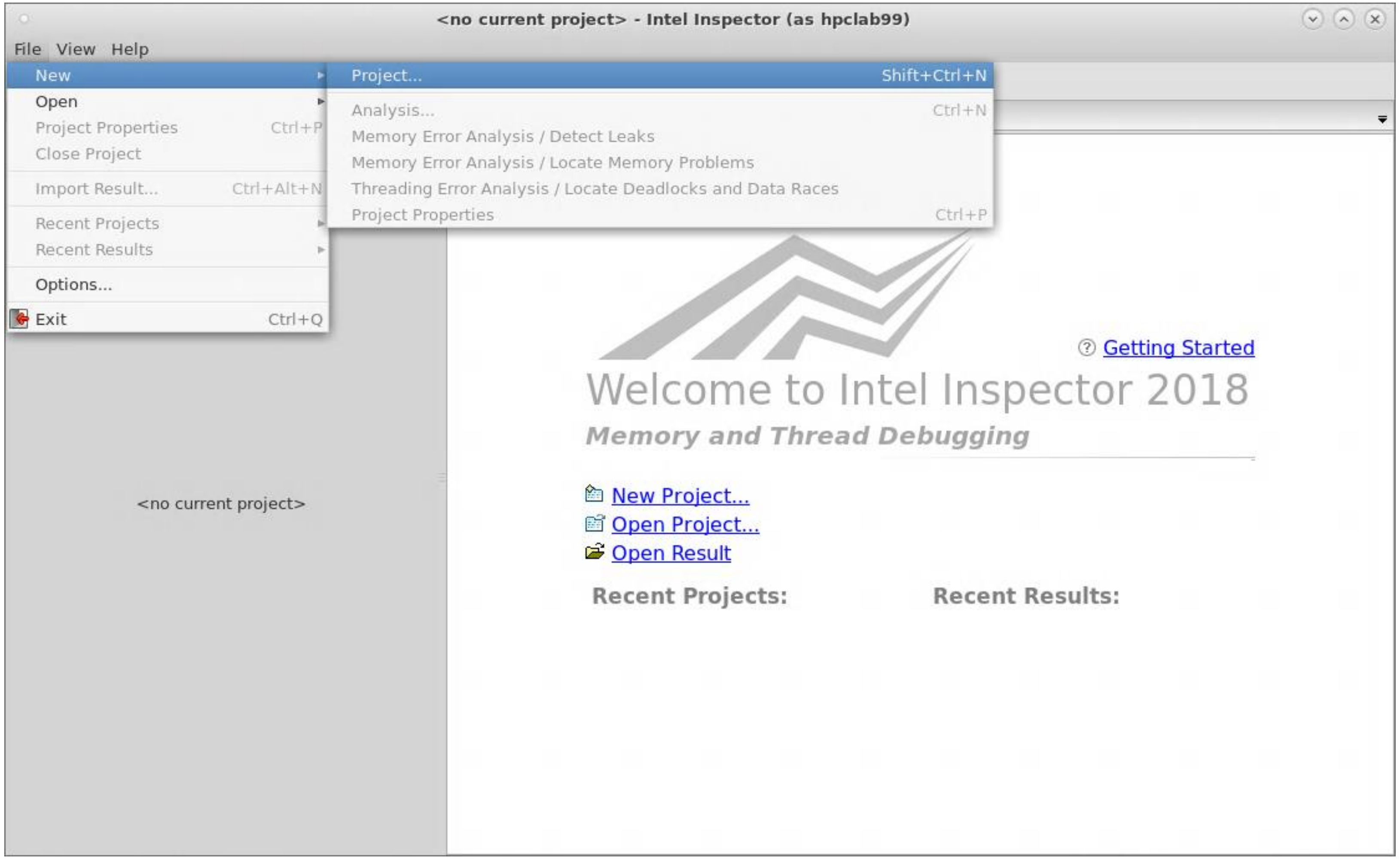
PI example / 2

```
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?

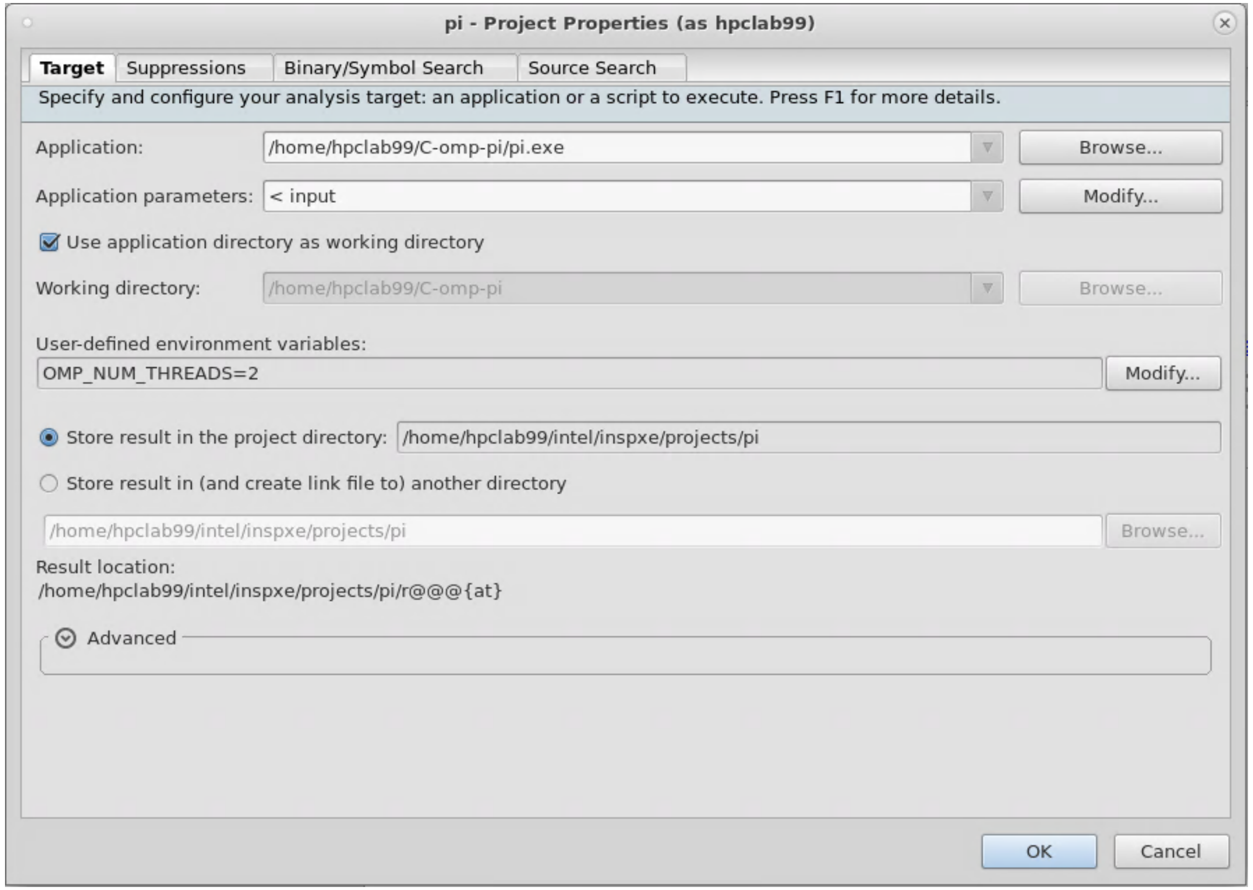
Inspector XE: create project / 1

```
$ module load Inspector ; inspxe-gui
```



Inspector XE: create project / 2

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

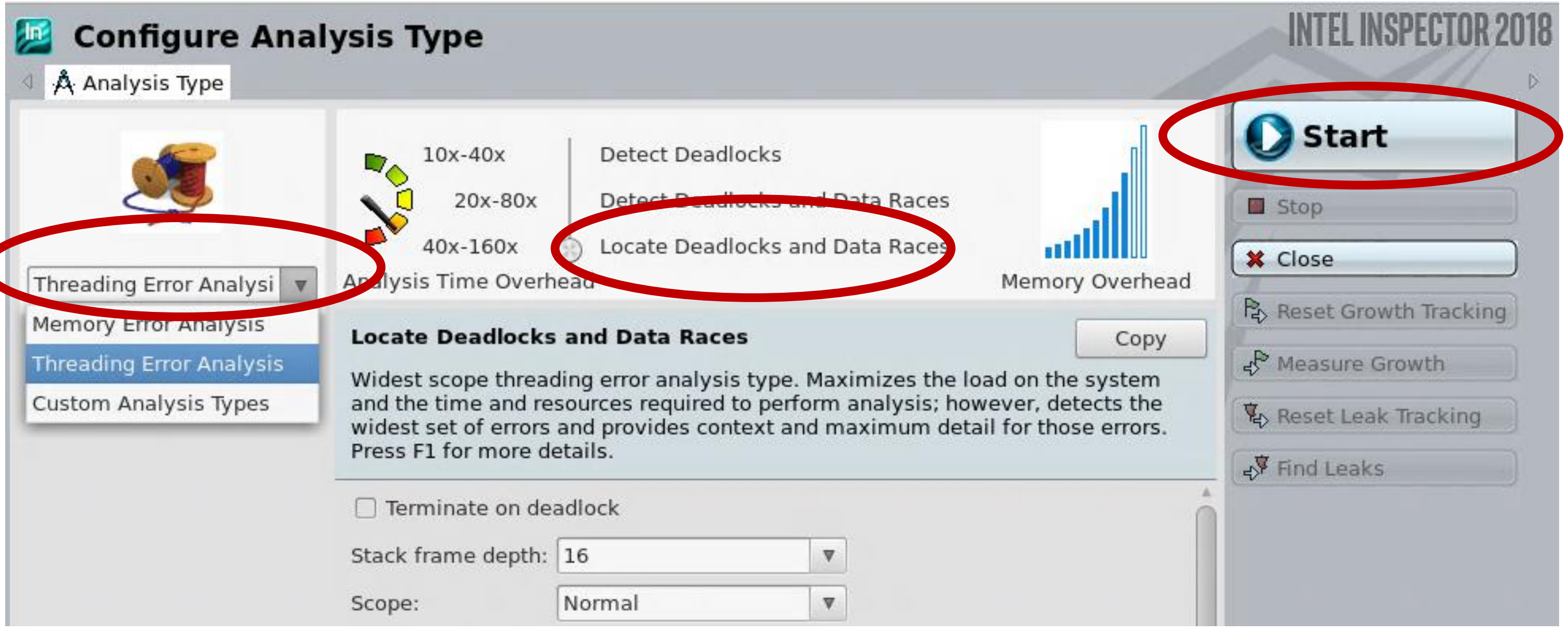


Inspector XE: configure analysis

Threading Error Analysis Modes

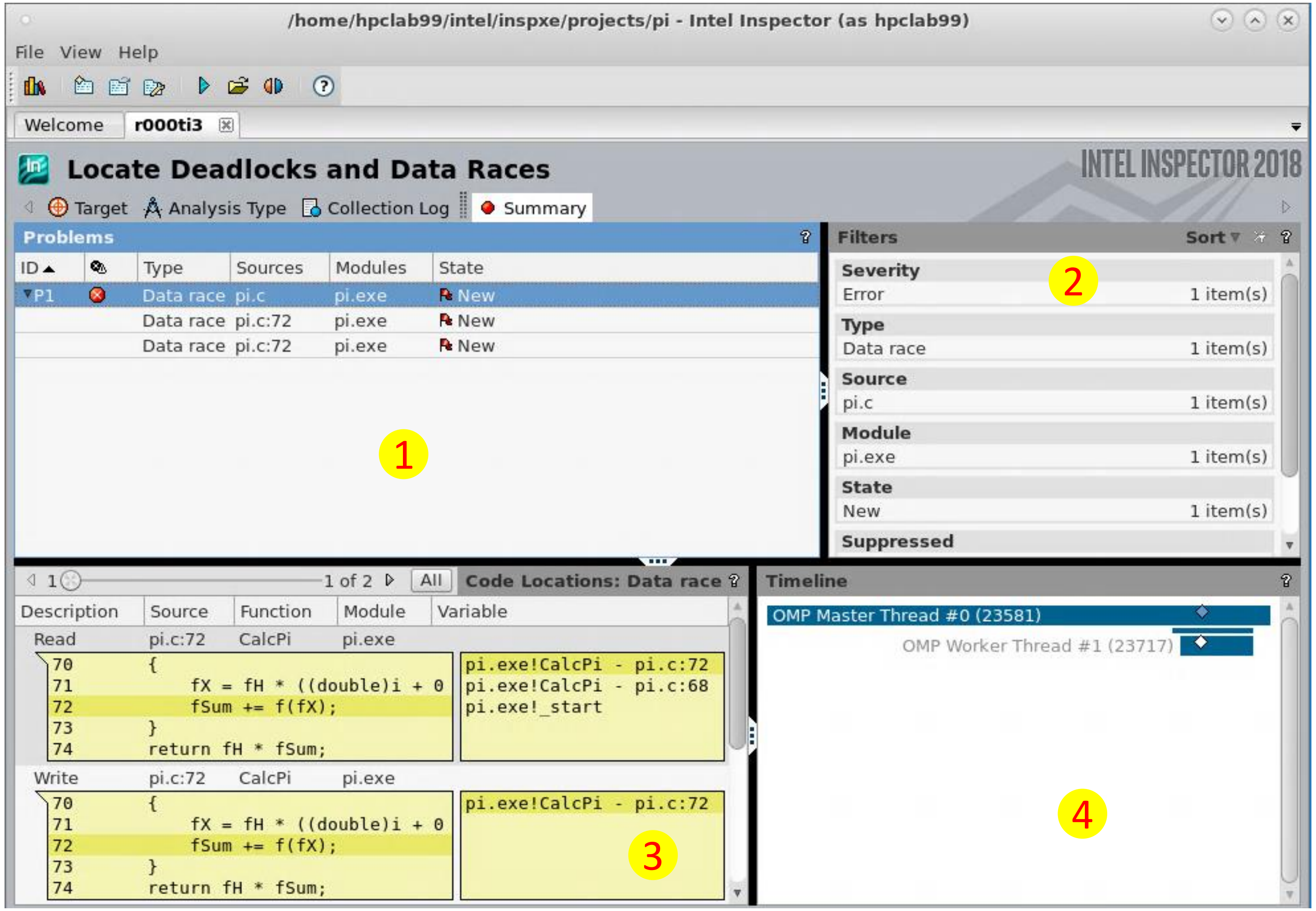
1. Detect Deadlocks
2. Detect Deadlocks and Data Races
3. Locate Deadlocks and Data Races

more details,
more overhead



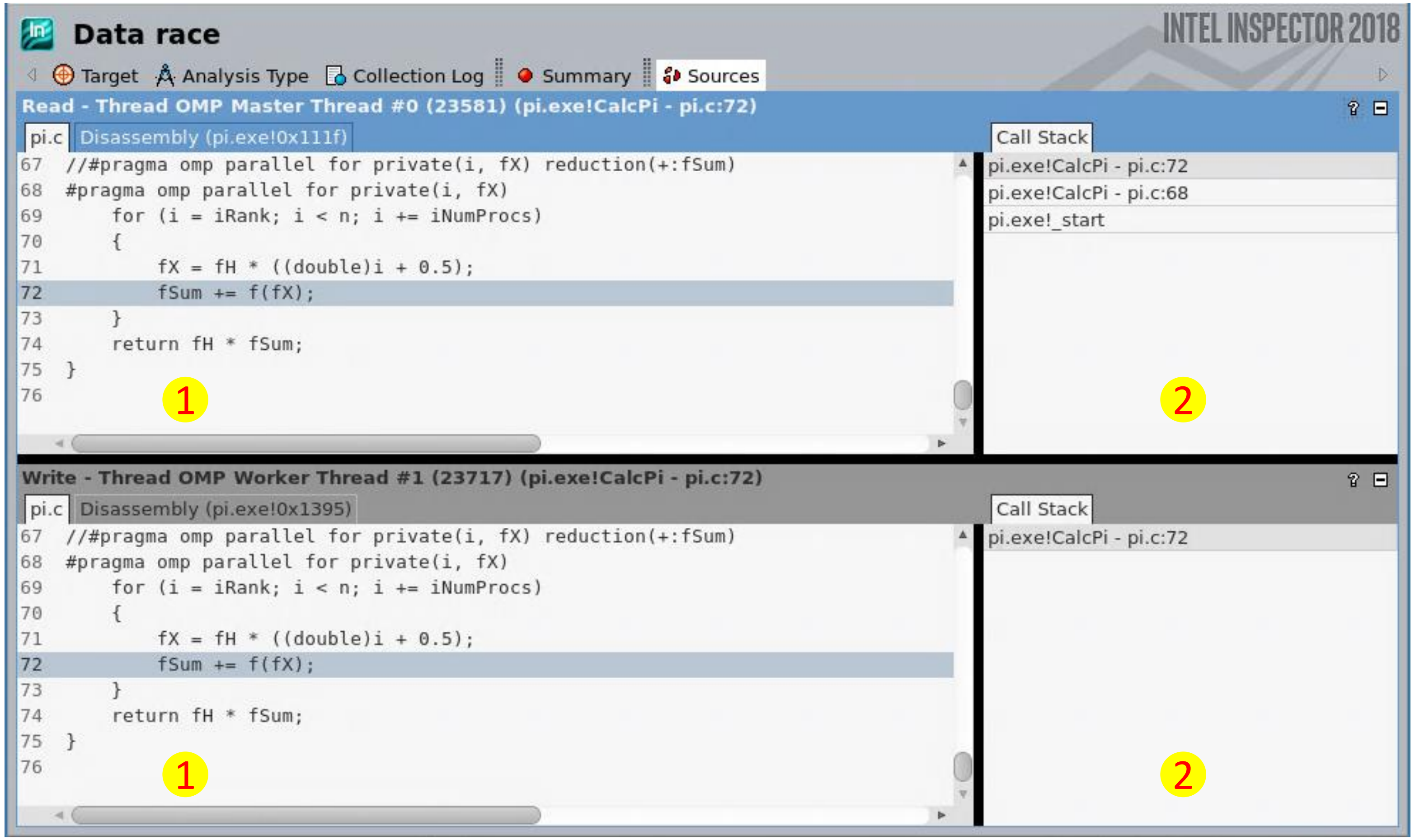
Inspector XE: results / 1

1 detected problems
2 filters
3 code location
4 Timeline



Inspector XE: results / 2

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

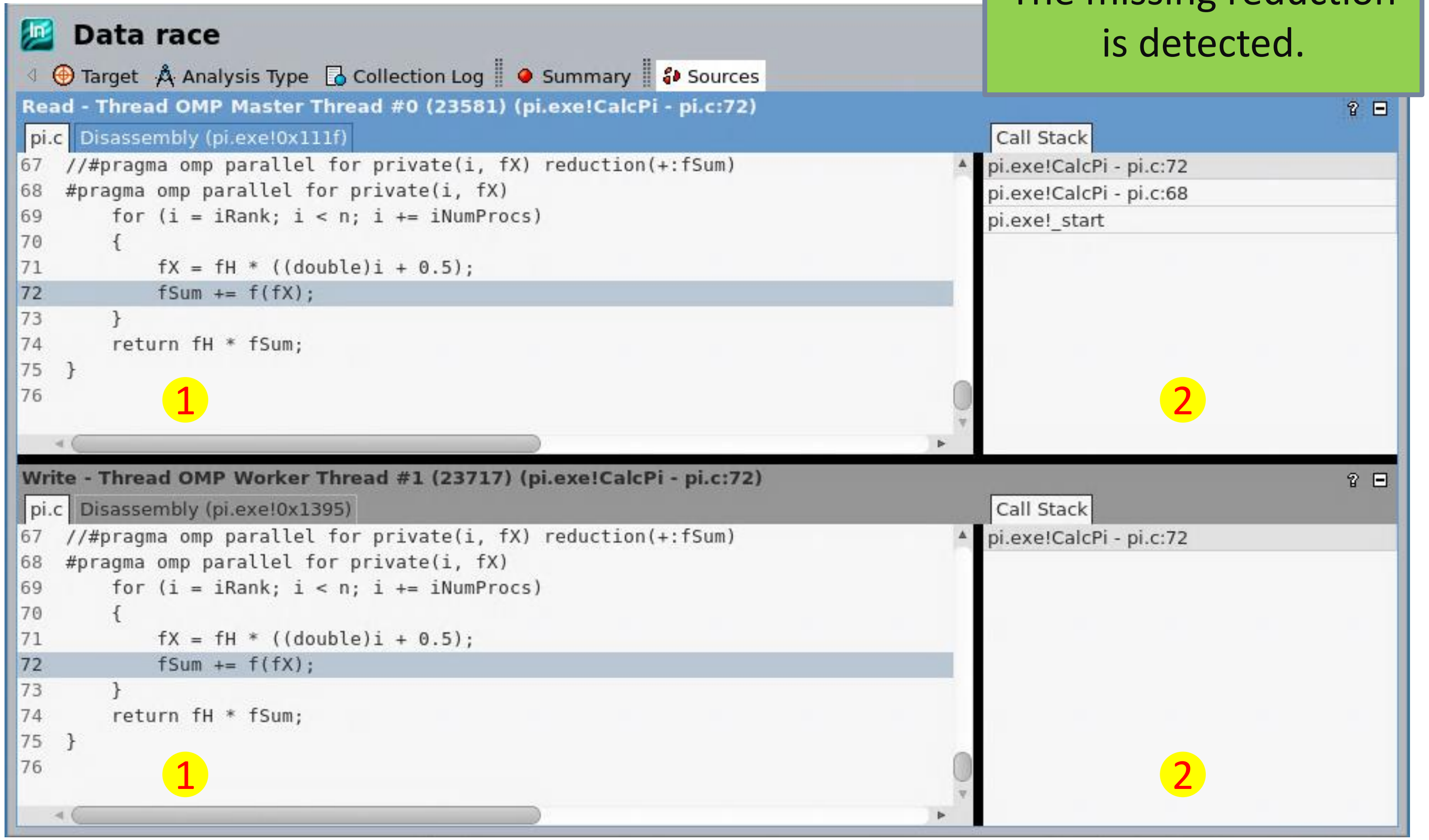


Inspector XE: results / 3

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

2 Corresponding Call Stack

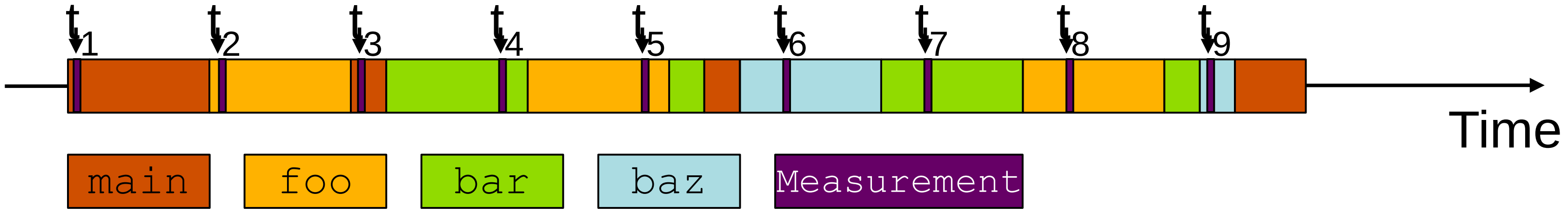
The missing reduction is detected.



Sampling vs. Instrumentation

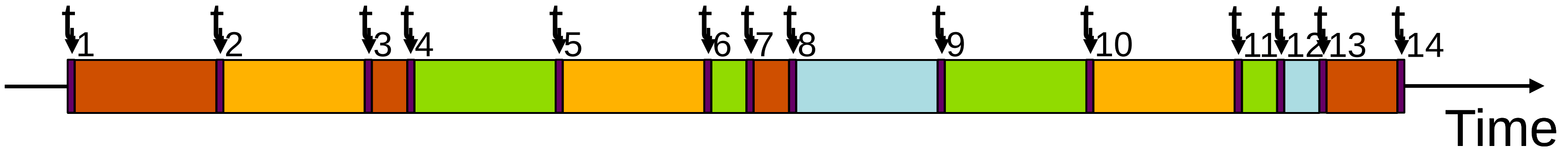
Sampling

- Running program is periodically interrupted to take measurement
- *Statistical* inference of program behavior
- Works with unmodified executables



Instrumentation

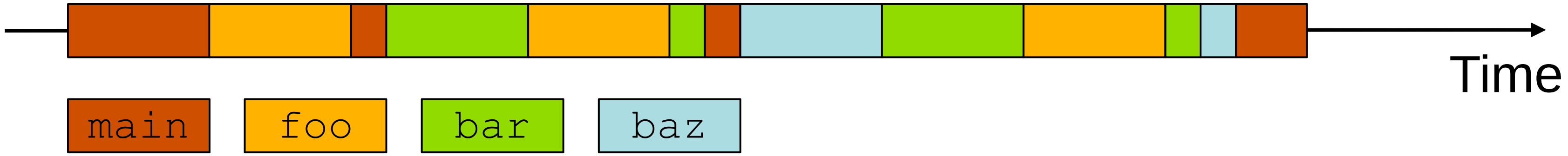
- Every event of interest is captured directly
- More detailed and *exact* information
- Typically: recompile for instrumentation



Tracing vs. Profiling

Trace

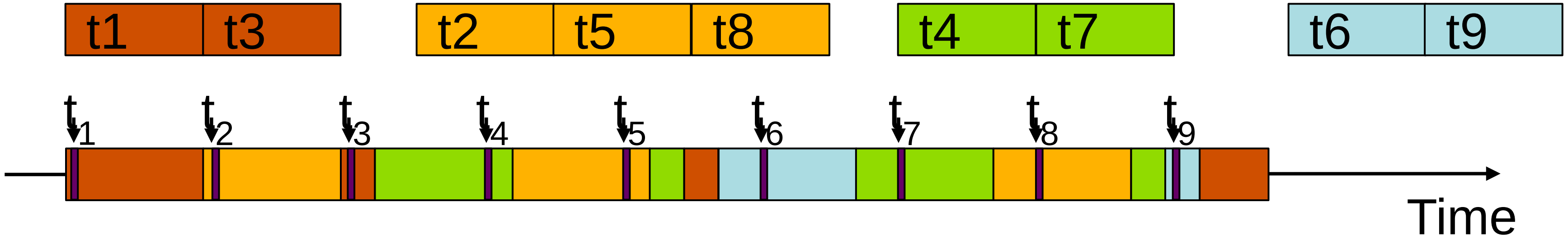
- Chronologically ordered sequence of event records



Profile from instrumentation

- Aggregated information

Profile from sampling

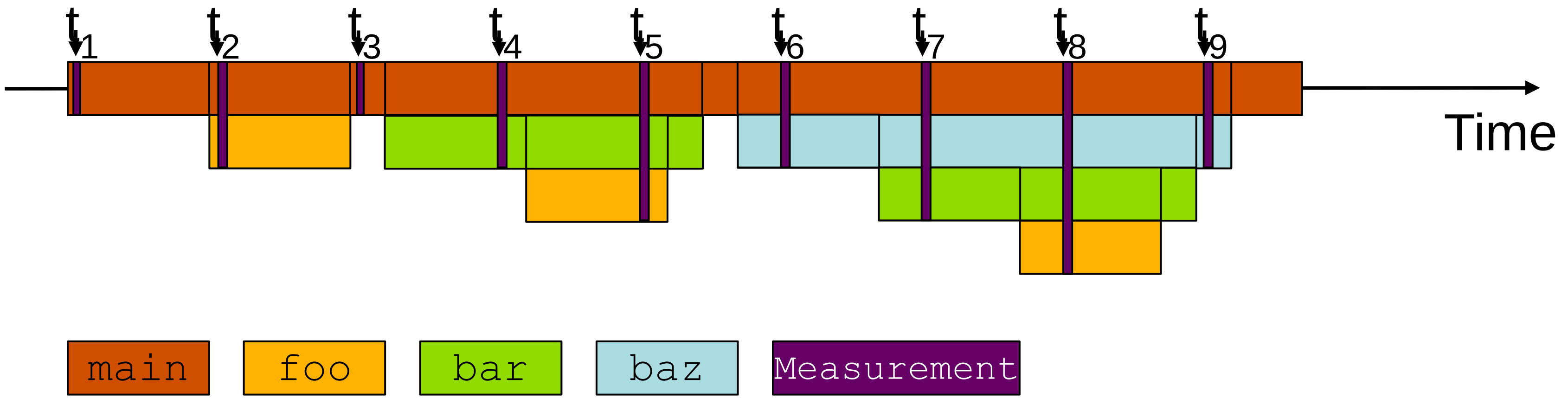


OMPT support for sampling

- OMPT defines states like *barrier-wait*, *work-serial* or *work-parallel*
  - → Allows to collect OMPT state statistics in the profile
  - → Profile break down for different OMPT states

```
void foo() {}
void bar() {foo();}
void baz() {bar();}
int main()
{foo();bar();baz();
 return 0;}
```

- OMPT provides frame information
  - → Allows to identify OpenMP runtime frames.
  - → Runtime frames can be eliminated from call trees



OMPT support for instrumentation

- OMPT provides event callbacks
  - → Parallel begin / end
  - → Implicit task begin / end
  - → Barrier / taskwait
  - → Task create / schedule
- Tool can instrument those callbacks
- OpenMP-only instrumentation might be sufficient for some use-cases

```
void foo() {}
void bar() {
  #pragma omp task
    foo();}
void baz() {
 #pragma omp task
    bar();}
int main() {
#pragma omp parallel sections
{foo();bar();baz();}
 return 0;}
```

VI-HPS Tools / 1

- Virtual institute – high productivity supercomputing
- Tool development
- Training:
  - → VI-HPS/PRACE tuning workshop series
  - → SC/ISC tutorials
- Many performance tools available under vi-hps.org
  - → → tools → VI-HPS Tools Guide
  - → Tools-Guide: flyer with a 2 page summary for each tool

VI-HPS Tools / 2

Data collection

- Score-P : instrumentation based profiling / tracing
- Extrae : instrumentation based profiling / tracing

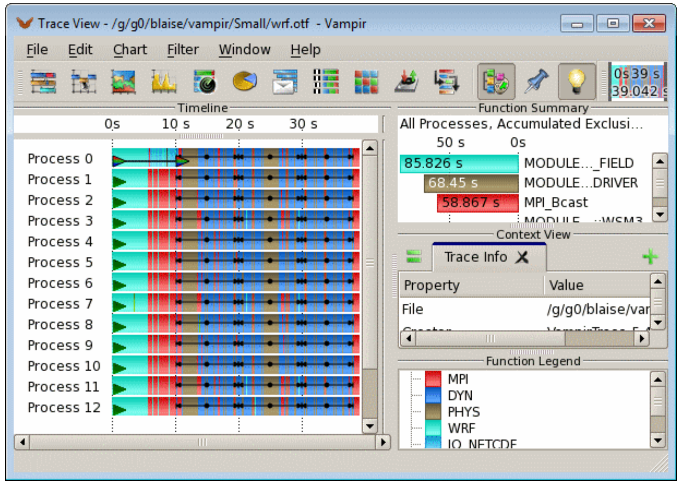
Data processing

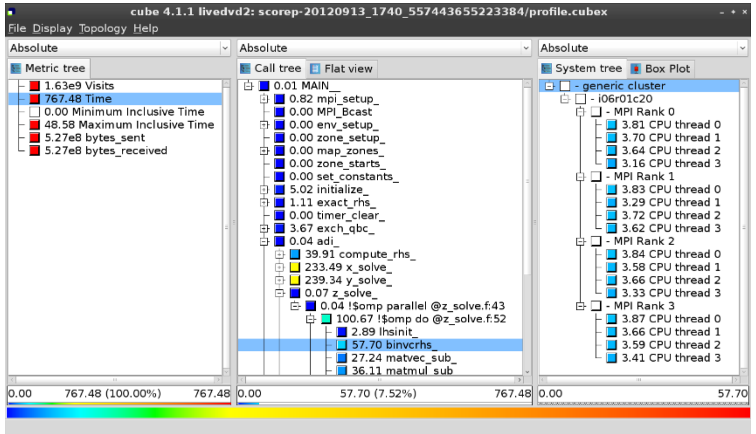
- Scalasca : trace-based analysis

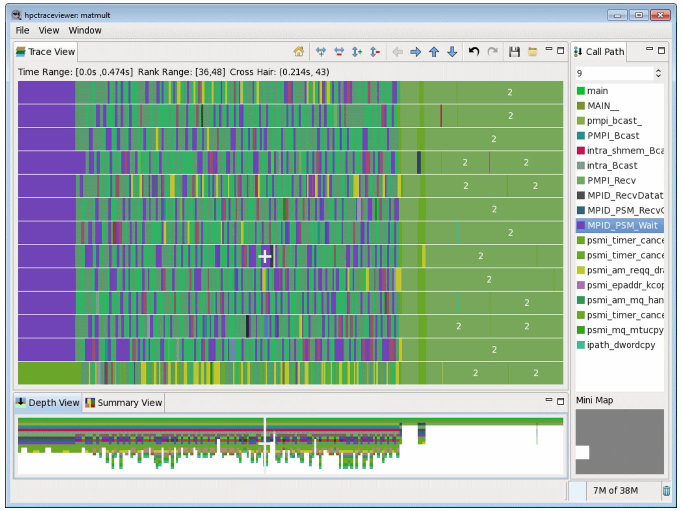
Data presentation

- ARM Map, ARM performance report
- CUBE : display for profile information
- Vampir : display for trace data (commercial/test)
- Paraver : display for extrae data
- Tau : visualization

Performance tools GUI







HPC Toolkit

Summary

Correctness:

- Data Races are very hard to find, since they do not show up every program run.
- Intel Inspector XE or ThreadSanitizer help a lot in finding these errors.
- Use really small datasets, since the runtime increases significantly.

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 might be hard to interpret.

OpenMP Parallel Loops

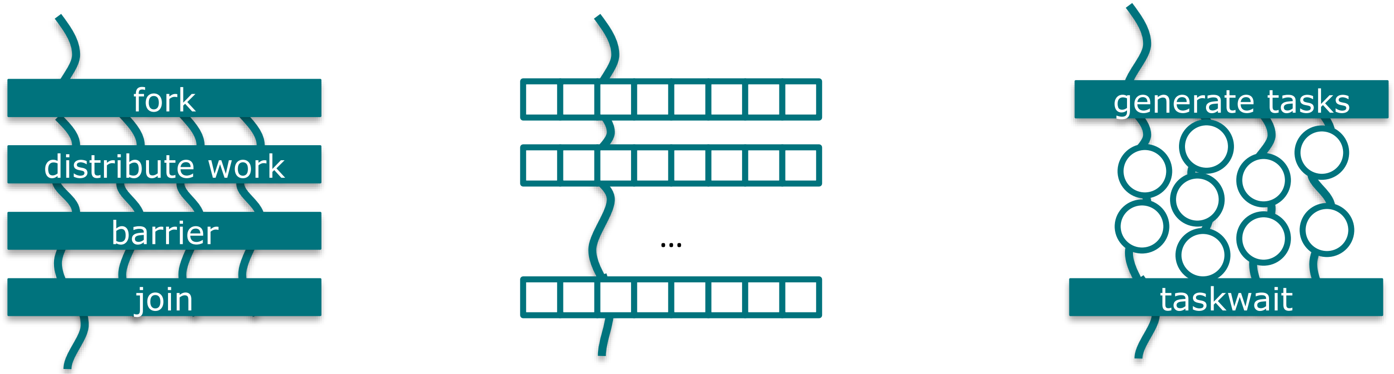
`loop` Construct

- Existing loop constructs are tightly bound to execution model:

```
#pragma omp parallel for
for (i=0; i<N;++i) {…}
```

```
#pragma omp simd
for (i=0; i<N;++i) {…}
```

```
#pragma omp taskloop
for (i=0; i<N;++i) {…}
```



- The **`loop`** construct is meant to tell OpenMP about truly parallel semantics of a loop.

OpenMP Fully Parallel Loops

```
int main(int argc, const char* argv[]) {
    float *x = (float*) malloc(n * sizeof(float));
    float *y = (float*) malloc(n * sizeof(float));
    // Define scalars n, a, b & initialize x, y

#pragma omp parallel
#pragma omp loop
    for (int i = 0; i < n; ++i){
      y[i] = a*x[i] + y[i];
    }
  }
}
```

`loop` Constructs, Syntax

- Syntax (C/C++)

```
#pragma omp loop [clause[[,] clause],…]
for-loops
```

- Syntax (Fortran)

```
!$omp loop [clause[[,] clause],…]
do-loops
[!$omp end loop]
```

`loop` Constructs, Clauses

- `bind(`*`binding`*`)`
  - → Binding region the loop construct should bind to
  - → One of: `teams`, `parallel`, `thread`

- `order(concurrent)`
  - → Tell the OpenMP compiler that the loop can be executed in any order.
  - → Default!

- `collapse(`*`n`*`)`
- `private(`*`list`*`)`
- `lastprivate(`*`list`*`)`
- `reduction(`*`reduction-id`*`:`*`list`*`)`

Extensions to Existing Constructs

- Existing loop constructs have been extended to also have truly parallel semantics.

- C/C++ Worksharing:

```
#pragma omp [for|simd] order(concurrent) \
                              [clause[[,] clause],…]
for-loops
```

- Fortran Worksharing:

```
!$omp [do|simd] order(concurrent) &
                  [clause[[,] clause],…]
do-loops
[!$omp end [do|simd}]
```

DOACROSS Loops

DOACROSS Loops

- "DOACROSS" loops are loops with special loop schedules
  - → Restricted form of loop-carried dependencies
  - → Require fine-grained synchronization protocol for parallelism

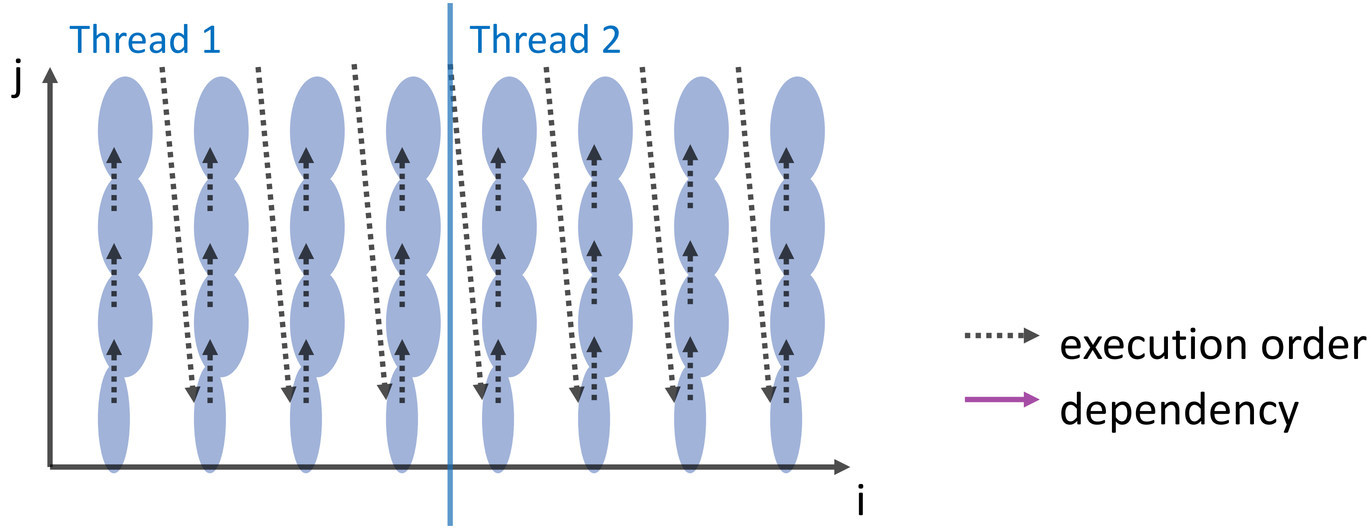
- Loop-carried dependency:
  - → Loop iterations depend on each other
  - → Source of dependency must scheduled before sink of the dependency

- DOACROSS loop:
  - → Data dependency is an invariant for the execution of the whole loop nest

Parallelizable Loops

- A parallel loop cannot not have any loop-carried dependencies (simplified just a little bit!)

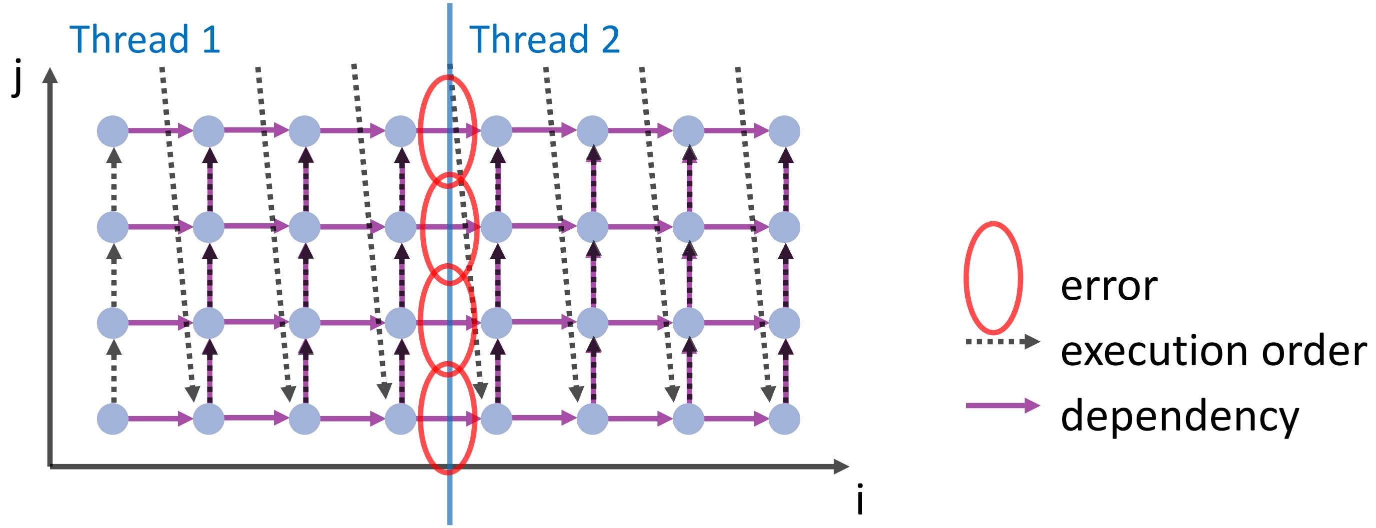
```
for (int i = 1; i < N; ++i) {
    for (int j = 1; j < M; ++j) {
        b[i][j] = f(b[i][j],
                    b[i][j], a[i][j]);
    }
}
```



Non-parallelizable Loops

- If there is a loop-carried dependency, a loop cannot be parallelized anymore ("easily" that is)

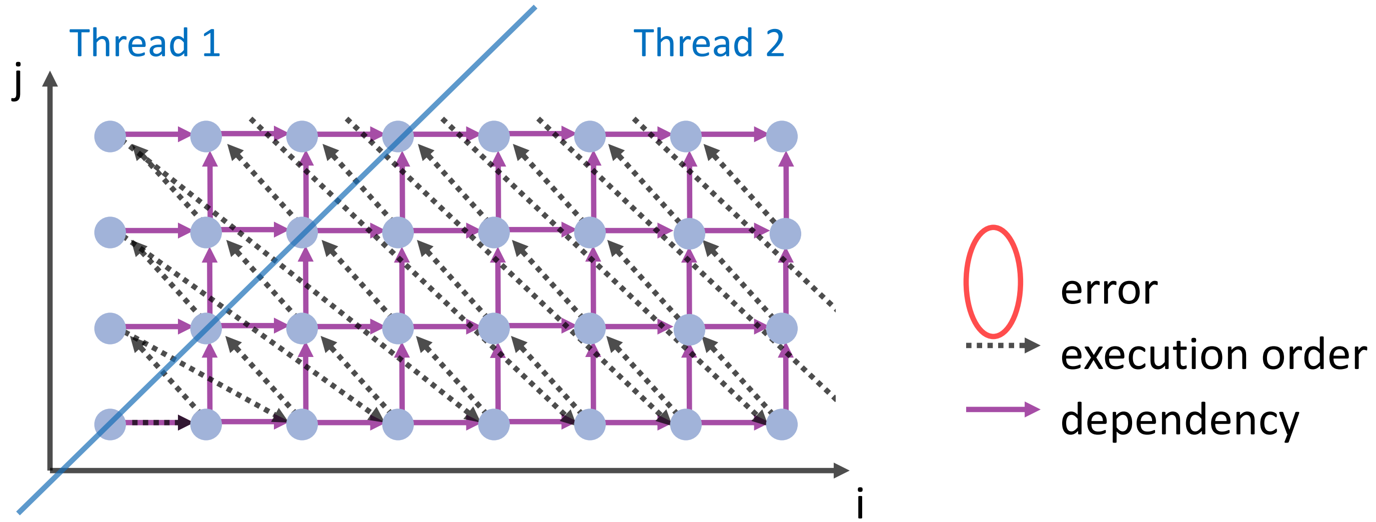
```
for (int i = 1; i < N; ++i) {
    for (int j = 1; j < M; ++j) {
        b[i][j] = f(b[i-1][j],
                    b[i][j-1], a[i][j]);
    }
}
```



Wavefront-Parallel Loops

- If the data dependency is invariant, then skewing the loop helps remove the data dependency

```
for (int i = 1; i < N; ++i) {
    for (int j = i+1; j < i+N; ++j) {
        b[i][j-i] = f(b[i-1][j-i],
                      b[i][j-i-1], a[i][j]);
    }
}
```



DOACROSS Loops with OpenMP

- OpenMP 4.5 extends the notion of the ordered construct to describe loop-carried dependencies
- Syntax (C/C++):

```
#pragma omp for ordered(d) [clause[[,] clause],…]
for-loops
```

and

```
#pragma omp ordered [clause[[,] clause],…]
```

where clause is one of the following:

```
depend(source)
depend(sink:vector)
```

- Syntax (Fortran):

```
!$omp do ordered(d) [clause[[,] clause],…]
do-loops
!$omp ordered [clause[[,] clause],…]
```

Example

- The ordered clause tells the compiler about loop-carried dependencies and their distances

```
#pragma omp parallel for ordered(2)
for (int i = 1; i < N; ++i) {
    for (int j = 1; j < M; ++j) {
#pragma omp ordered depend(sink:i-1,j) depend(sink:i,j-1)
        b[i][j] = f(b[i-1][j],
                    b[i][j-1], a[i][j]);
    }
#pragma omp ordered depend(source)
}
```

Example: 3D Gauss-Seidel

```
#pragma omp for ordered(2) private(j,k)
for (i = 1; i < N-1; ++i) {
  for (j = 1; j < N-1; ++j)   {
#pragma omp ordered depend(sink: i-1,j-1) depend(sink: i-1,j) \
                    depend(sink: i-1,j+1) depend(sink: i,j-1)
    for (k = 1; k < N-1; ++k) {
      double tmp1 = (p[i-1][j-1][k-1] + p[i-1][j-1][k] + p[i-1][j-1][k+1]
                   + p[i-1][j][k-1] + p[i-1][j][k] + p[i-1][j][k+1]
                   + p[i-1][j+1][k-1] + p[i-1][j+1][k] + p[i-1][j+1][k+1]);
      double tmp2 = (p[i][j-1][k-1] + p[i][j-1][k] + p[i][j-1][k+1]
                   + p[i][j][k-1] + p[i][j][k] + p[i][j][k+1]
                   + p[i][j+1][k-1] + p[i][j+1][k] + p[i][j+1][k+1]);
      double tmp3 = (p[i+1][j-1][k-1] + p[i+1][j-1][k] + p[i+1][j-1][k+1]
                   + p[i+1][j][k-1] + p[i+1][j][k] + p[i+1][j][k+1]
                   + p[i+1][j+1][k-1] + p[i+1][j+1][k] + p[i+1][j+1][k+1]);
      p[i][j][k] = (tmp1 + tmp2 + tmp3) / 27.0;
    }
#pragma omp ordered depend(source)
  }
}
```

OpenMP API Version 5.1
State of the Union

OpenMP

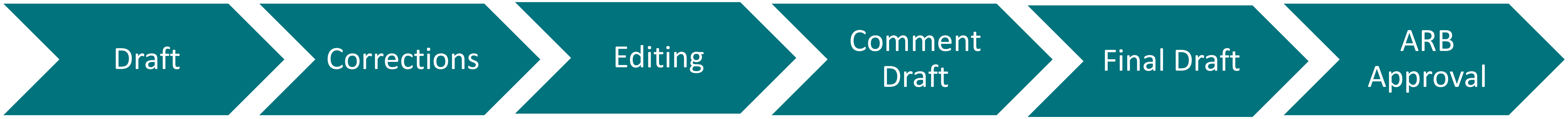
Architecture Review Board

The mission of the OpenMP ARB (Architecture Review Board) is to standardize directive-based multi-language high-level parallelism that is performant, productive and portable.

Development Process of the Specification

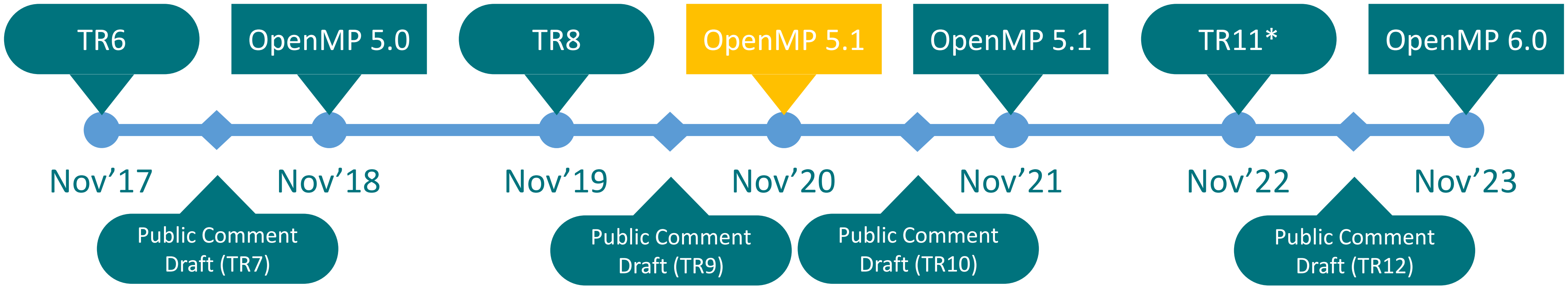
- Modifications of the OpenMP specification follow a (strict) process:

- Release process for specifications:



OpenMP Roadmap

- OpenMP has a well-defined roadmap:
  - 5-year cadence for major releases
  - One minor release in between
  - (At least) one Technical Report (TR) with feature previews in every year

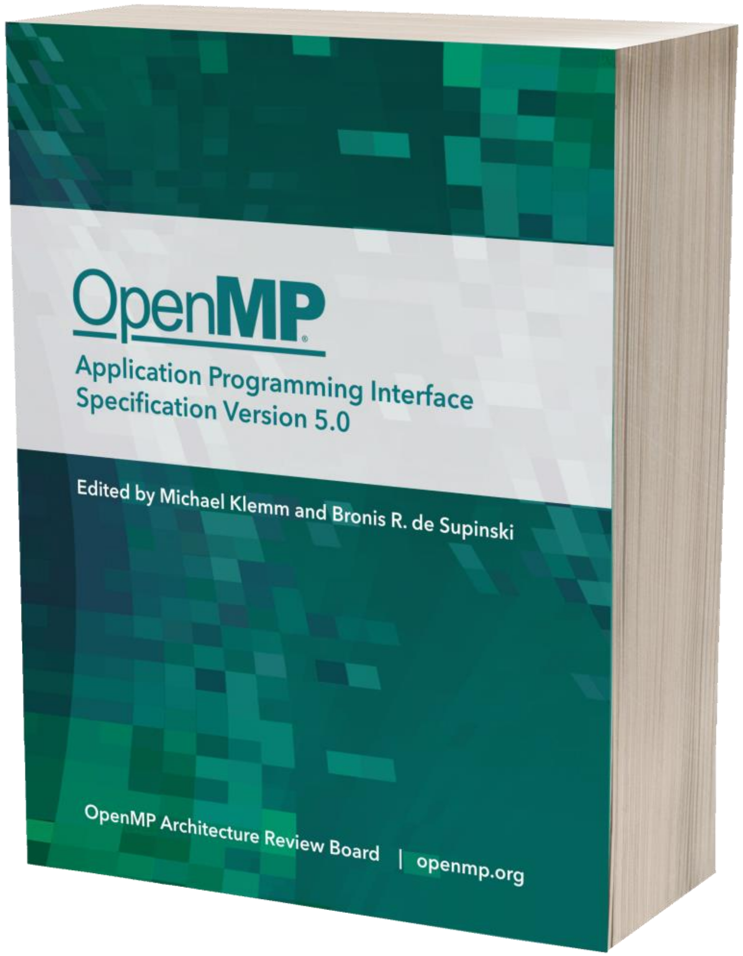


* Numbers assigned to TRs may change if additional TRs are released.

OpenMP API Version 6.0 Outlook – Plans

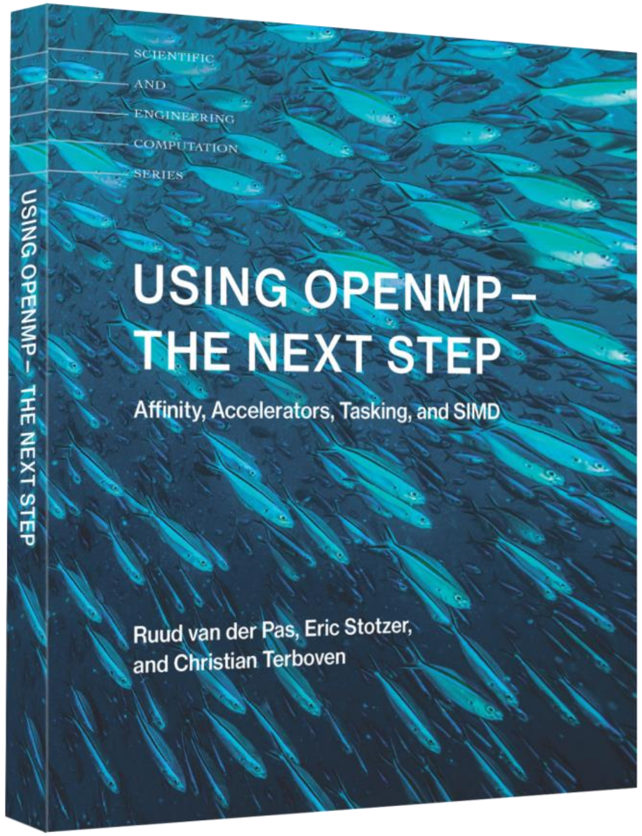
- Better support for descriptive and prescriptive control
- More support for memory affinity and complex memory hierarchies
- Support for pipelining, other computation/data associations
- Continued improvements to device support
  - Extensions of deep copy support (serialize/deserialize functions)
- Task-only, unshackled or free-agent threads
- Event-driven parallelism

Printed OpenMP API Specification

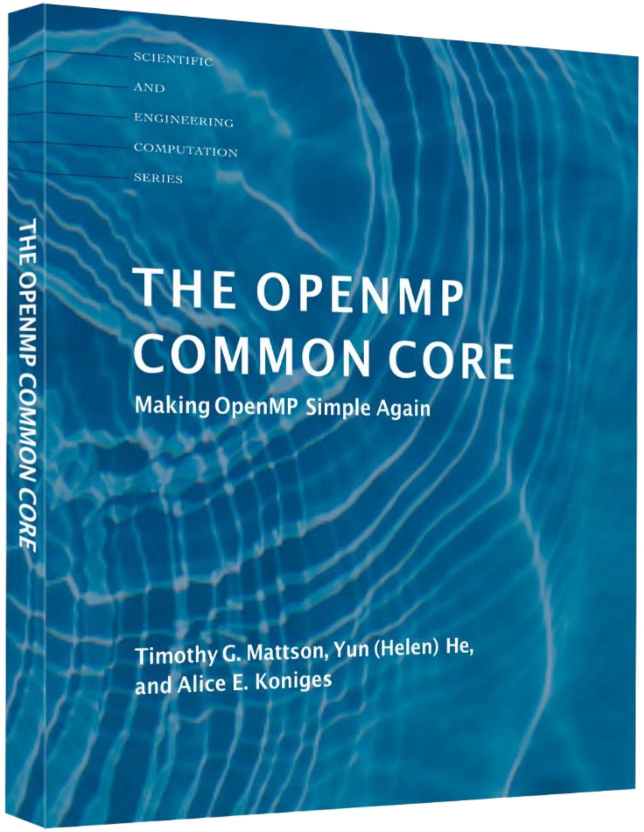


- Save your printer-ink and get the full specification as a paperback book!
  - Always have the spec in easy reach.
  - Includes the entire specification with the same pagination and line numbers as the PDF.
  - Available at a near-wholesale price.
- Get yours at Amazon at https://link.openmp.org/book51

Recent Books about OpenMP



Covers all of the
OpenMP 4.5 features, 2017



Introduces the
OpenMP Common Core, 2019

Help Us Shape the Future of OpenMP

- OpenMP continues to grow
  - 33 members currently
- You can contribute to our annual releases
- Attend IWOMP, become a cOMPunity member
- OpenMP membership types now include less expensive memberships
  - Please get in touch with me if you are interested

OpenMP®

Enabling HPC since 1997

Visit www.openmp.org for more information